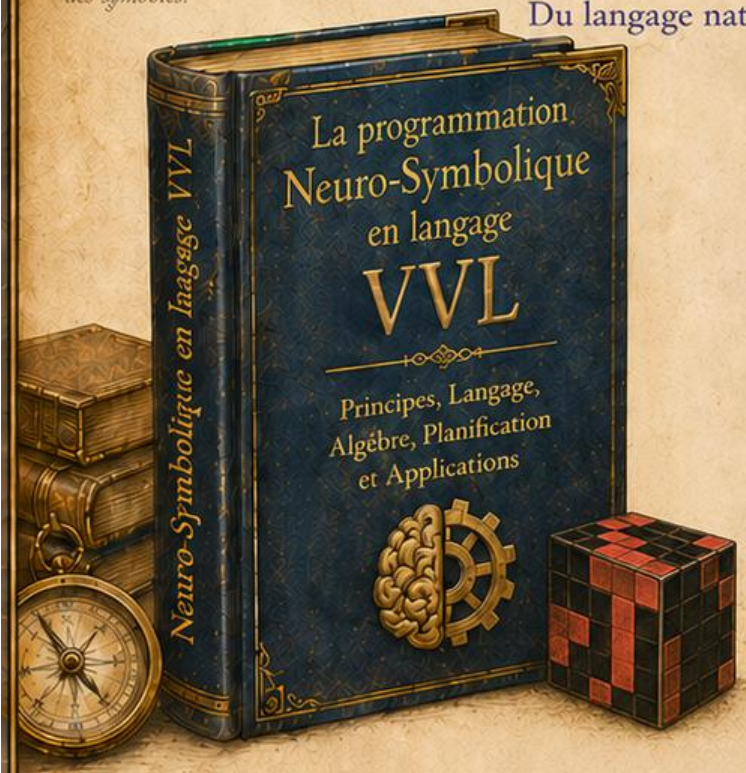
La programmation Neuro-Symbolique en langage VVL

L'intelligence du langage, la rigueur des symboles.

Comprendre, formaliser, raisonner, exécuter.

Du langage naturel au planning vérifié



Un livre pour bâtir l'intelligence des systèmes hybrides de demain

Ce livre présente VVL, un langage formel et exécutable conçu pour relier la compréhension du langage naturel à la puissance des planificateurs et solveurs.
Il expose une approche neuro-symbolique complète, du texte libre jusqu'au planning vérifié.

Neuro
Comprendre le langage naturel grâce aux LLM et extraire l'intention.

Symbolique
Formaliser l'intention en VVL et la traduire en algèbre exécutable.

Algébrique
Raisonner, contraindre et planifier avec des solveurs déterministes (CP-SAT, Prolog).

Vérifiable
Obtenir un résultat audité, réutilisable et améliorable sans le LLM.

Le pipeline Neuro-Symbolique avec VVL

1. Intention NL

« Annule la mission Delorme vendredi »

NEURO

2. VVL (compilé)

```
[$p.apply~event
[new Event
"Cancellation"
$delorme]]
```

SYMBOLIQUE

3. Algèbre exécutable

λ

Contraintes + Règles
+ Faits (Prolog/Zalta)
+ Variables
→ Modèle algébrique

4. Exécution déterministe

CP-SAT / OR-Tools
Re-solve incrémental
→ Planning + Delta

5. Résultat audité

Planning mis à jour
Delta minimal
Hypothèses tracées

Comprendre, formaliser, exécuter, exécuter.

VAILS

VDOM AI LEARNING SYSTEM

L'intelligence du langage, la rigueur des symboles.

Version: 4.5.0 | Engine: Neuro-Symbolic | Mode: Interactive REPL

```
VAILS 4.5.0 - VVL Language 3.3
:q quit    :t trace    :h help    :vvl / :prolog switch language
:rest start API server    :ai-driven start AI session
:genius <question> ask docs    :agent <task> generate code
:gateway on/off enable ngrok  public tunnels
:log on  real-time log monitor  (TUI with tabs)
VVL>
```





Un langage. Un paradigme. Une nouvelle intelligence.

Programmation Neuro-Symbolique en langage VVL

Du paradigme symbolique aux opérateurs de Zalta

Nicolas Korboulewsky

Avril 2026

TABLE DES MATIERES

Introduction — comment lire ce livre 27
La carte de l'ouvrage 27
Trois parcours de lecture 27
Conventions 28
1. Histoire des paradigmes de l'intelligence artificielle 29
1.1. L'IA symbolique : Lisp, Prolog et les systèmes experts (1956–1980) 31
1.2. L'hiver de l'IA et la critique de Dreyfus 32
1.3. La renaissance connexionniste : du perceptron au deep learning 33
1.4. L'ère des transformers : GPT et le raisonnement statistique émergent 34
1.5. Les limites des LLM purs : hallucinations, opacité, non-vérifiabilité 36
1.6. Le retour du symbolique et la fusion neuro-symbolique 38
2. Fondements théoriques du neuro-symbolique 40
2.1. Raisonnement déductif : logique du premier ordre et chaînage 40
2.2. Raisonnement inductif : de l'exemple à la règle 43
2.3. Raisonnement abductif : la meilleure explication d'une observation 44
2.4. La théorie des objets abstraits d'Edward Zalta 45
2.5. Double prédication : exemplification (prop/3) et encodage (enc/3) 46
2.6. Panorama des architectures hybrides existantes 48
DeepProbLog : Prolog différenciable 48
NeurASP : Answer Set Programming et perception neuronale 48
Logic Tensor Networks : la logique floue dans les réseaux 48
La position de VVL 49
3. VAILS et VVL : un langage conçu pour le neuro-symbolique 51
3.1. La vision : un seul runtime, un seul espace de représentation 51
3.2. L'homoiconicité : code, données et langage naturel partagent la même syntaxe 52
3.3. Le LLM comme citoyen de première classe du langage 54
3.4. L'architecture VAILS en sept sous-systèmes 56
3.5. Comparaison avec les approches assemblées (LangChain, LlamaIndex) 58
4. Syntaxe et types fondamentaux 61
4.1. Tout est commande entre crochets 61
4.2. Variables, types primitifs et opérateurs 62
4.3. Listes, JSON et XML : trois syntaxes natives 64
Listes natives 65
JSON natif 65
XML natif 66
4.4. Structures de contrôle : if, when, for, while, match 67
Conditionnelles : if, when, not 67
Match : expression de pattern matching 67

Switch : instruction de branchement 68
Gestion d'erreurs : try / catch 68
4.5. Pattern matching algébrique : types Option et Result 69
Le type Option : Some / None 69
Le type Result : Ok / Error 69
5. Programmation fonctionnelle 73
5.1. Lambdas et fermetures 73
5.2. Keywords : fonctions nommées et macros 74
5.3. Fonctions d'ordre supérieur : map, filter, pmap 76
map : transformation un-à-un 76
pmap : map parallèle 77
Composition et pipelines 77
5.4. Générateurs et fibers : évaluation paresseuse 78
Producteur–consommateur découplés 80
6. Programmation orientée objet 82
6.1. Philosophie : tout reste une liste 82
6.2. Classes, champs et méthodes 82
6.3. Héritage et $super 84
6.4. Mixins et résolution MRO 86
6.5. Protocols : interfaces contractuelles 88
6.6. Annotations de validation et surcharge d'opérateurs 89
Annotations de validation 89
Surcharge d'opérateurs 90
7. Les opérateurs sémantiques 93
7.1. Le type VVL_Semantic et la syntaxe [§ "..."] 93
7.2. Comparaisons floues : ==, !=, <, >, <=, >= 94
7.3. Algèbre conceptuelle : +, -, *, /, % 95
7.4. Extraction QA et composition logique : ?, &&, ||, ^^, ~ 96
Extraction question-réponse (?) 96
Composition logique (&&, ||, ^^, ~) 96
7.5. Appartenance et inclusion : in, <|, |> 97
7.6. Garanties de type, cache déterministe et retry automatique 98
Garantie 1 — Typage de sortie 98
Garantie 2 — Cache déterministe persistant 99
Garantie 3 — Retry automatique avec validation 99
Garantie 4 — Spécifications XML avec few-shot 99
7.7. Référence des 20 opérateurs sémantiques 101
8. Génération de code par LLM : l'opérateur @ 103
8.1. Mise au point : les quatre usages du caractère @ en VVL 103

8.2. L'opérateur @ : du langage naturel à la lambda 104
8.3. Pipeline interne : génération, validation, retry, cache 105
8.4. Sandbox des builtins autorisés dans le code généré 108
8.5. Patterns d'utilisation et bonnes pratiques 108
Pattern 1 — Génération unique, application multiple 108
Pattern 2 — Composition avec map 109
Pattern 3 — Description précise plutôt que vague 109
Pattern 4 — Échec de validation : que faire ? 109
8.6. L'opérateur => : génération libre de texte, JSON, code 110
8.7. Synthèse et transition 111
9. Réification de concepts : l'opérateur [@ "..."] 113
9.1. Du nom au modèle paramétrique 113
9.2. Le cycle de vie en trois phases : bootstrap, lazy load, manipulation 115
Phase 1 — Bootstrap 115
Phase 2 — Découverte paresseuse 116
Phase 3 — Manipulation formelle 116
9.3. Le cache à trois niveaux : session, objet, disque 117
Niveau 1 — Mémoire de session 117
Niveau 2 — Champs de l'objet 117
Niveau 3 — Disque 117
9.4. Typage automatique des propriétés 119
9.5. Désambiguïsation contextuelle par auto-réification 119
9.6. Le concept comme citoyen de première classe 121
10. L'opérateur §§ : compilation du langage naturel 123
10.1. De la question éphémère au programme réutilisable 123
10.2. Les opérateurs §§ et §§? : porte d’entrée universelle 124
10.3. Le router neuro-symbolique : classification et dispatch 125
Le rôle du classifieur 125
Le gate de clarification 125
10.4. Les huit familles de questions 126
10.4.1. arithmetic_closed — calcul direct 126
10.4.2. constraint_optim — CP-SAT 126
10.4.3. relational_query — base Prolog 126
10.4.4. compound — multi-solveur avec data-flow 127
10.4.5. rule_induction — induction de règle 127
10.4.6. factual_retrieval — lookup factuel 127
10.4.7. structured_extraction — JSON typé 127
10.4.8. expert_system — dispatch dynamique (Phase ROUTE-EXPERT) 128
10.5. L’écosystème des experts dynamiques 128

Exemple complet d'un expert métier 129
10.6. L'Approximation : wrapper d'audit en profondeur 129
L'API publique de l'Approximation 129
Les hypothèses (assumptions) et leur pondération 130
Les bandes de confiance 130
10.7. Configuration, comportement défensif, caches 130
Variables d'environnement et seuils 130
Builtins de debug du routage 131
Comportement défensif — fallbacks 131
Caches 131
10.8. Opérateurs et builtins de support 132
L'opérateur de puissance ^ 132
Builtins mathématiques 132
10.9. Mode trace `:t` et debug du routage 132
10.10. [generic] : capitaliser les solutions résolues 133
10.11. L'écosystème du catalogue [generic] 135
10.12. Limites connues 136
Exemples complets de bout en bout 137
10.13. Synthèse : les quatre rôles du LLM dans VVL 137
11. La base de connaissances Prolog 140
11.1. Le triangle neuro-symbolique de VVL 140
11.2. Faits, règles et requêtes : la grammaire de base 141
Faits : assertions déclaratives 142
Règles : déductions conditionnelles 142
Requêtes : interrogation du monde déclaré 142
11.3. Intégration depuis VVL : kb, facts, rules 143
11.4. Six modes de requête depuis VVL 144
11.5. Le pont OO ↔ Prolog : obj-assert 145
11.6. Cycle de vie complet d'un objet à travers les couches 147
12. Le système ontologique 151
12.1. Du fait nu au concept typé 151
12.2. Définition de schéma : ontology:define 152
12.3. Extraction par LLM : fromText et fromImage 153
12.4. Règles métier : ontology:rule 154
12.5. Inférence : infer et infer-fixpoint 155
infer : un passage 155
infer-fixpoint : boucle jusqu'à stabilité 155
12.6. Mondes éphémères : ontology:in-world 157
12.7. Synthèse : la pyramide symbolique 158

13. L'algèbre des opérateurs Zalta 159
13.1. Une algèbre pour les concepts 159
13.2. Architecture : dispatch via mixin et MRO 160
13.3. Satisfaction (== et ~~) 163
13.4. Intersection (* et **) 164
13.5. Enrichissement (+ et ++) 165
13.6. Projection (/ et //) 166
13.7. Synthèse : un objet, trois couches symboliques 167
14. RAG et recherche sémantique 170
14.1. Pourquoi RAG : les limites du LLM seul 170
14.2. Le pipeline RAG en trois phases 171
Phase 1 — Indexation (offline) 171
Phase 2 — Retrieval (recherche) 171
Phase 3 — Generation (génération) 171
14.3. Insertion de documents : construire la base de connaissances 172
Linéarisation des données structurées 173
14.4. Quatre modes de requête 174
14.5. CRUD documentaire : maintenir la base 176
14.6. Genius : la documentation VVL comme RAG 177
14.7. Synergie RAG + ontologie 178
15. Extraction structurée par LLM 182
15.1. Le LLM comme moteur d'OCR sémantique 182
15.2. Les trois opérateurs sémantiques d'extraction 183
15.3. Pipeline de scraping web 185
15.4. Extraction depuis images et PDF 186
15.5. Extraction typée par schéma ontologique 187
15.6. Pattern : extraction → typage → action 188
15.7. Synthèse : RAG vs extraction 189
16. Gherkin et ontologie dynamique 191
16.1. Inversion paradigmatique : observer pour inférer 191
16.2. Le format Gherkin : Background et Scenarios 192
Background : définition de schéma 192
Scenarios : règles métier 193
16.3. ontology:loadGherkin : la transpilation 194
16.4. ontology:context : déduction de schéma depuis des échantillons 195
16.5. ontology:learn : induction de règles métier 196
16.6. Pipeline complet : 5 phases neuro-symboliques 197
Auto-apprentissage continu 199
16.7. Synthèse : un changement de paradigme 201

17. Apprentissage inductif de règles ... 202
17.1. Le défi épistémologique de l'induction ... 202
17.2. Anatomie d'un dataset d'apprentissage ... 203
Format canonique : couples request/result ... 203
Trois axes de qualité ... 204
17.3. Induction libre : ontology:learn sans cible ... 205
17.4. Induction ciblée : ontology:learn avec target_feature ... 206
17.5. Validation et consolidation des règles induites ... 208
Validation croisée ... 208
Détection et consolidation ... 208
17.6. Conflits entre règles induites et règles écrites ... 209
17.7. Pattern d'induction continue ... 210
17.8. Synthèse de la Partie V : les quatre outils neuronaux ... 211
18. L'oscillation neuro-symbolique formalisée ... 213
18.1. Pourquoi formaliser l'oscillation ... 213
18.2. Anatomie d'une oscillation ... 214
18.3. Les quatre patterns canoniques d'oscillation ... 215
Pattern 1 — Définir / Extraire / Inférer / Présenter ... 216
Pattern 2 — Découvrir / Compiler / Apprendre / Déployer ... 216
Pattern 3 — Indexer / Chercher / Synthétiser ... 216
Pattern 4 — Mesurer / Hypothétiser / Tester / Itérer ... 216
18.4. Étude de cas : le diagnostic différentiel ... 217
18.5. L'objet comme pont entre les couches ... 219
18.6. Métriques d'oscillation : coût, latence, fidélité ... 220
18.7. Synthèse : VVL comme oscillation programmable ... 221
19. Diagnostic médical différentiel ... 223
19.1. Pourquoi le médical est l'archétype de l'oscillation neuro-symbolique ... 223
19.2. Le mapping Zalta-médical ... 224
19.3. Le schéma médical : classe Patient et cinq archétypes ... 225
19.4. Extraction des notes cliniques ... 227
19.5. Diagnostic différentiel par scoring ... 227
19.6. Règles métier : alertes et facteurs de risque ... 229
19.7. Simulation thérapeutique par mondes possibles ... 230
19.8. Synthèse : un système médical neuro-symbolique authentique ... 232
20. Système de devisage BTP de bout en bout ... 234
20.1. Contexte : un système de devisage industriel ... 234
20.2. Architecture en quatre fichiers ... 234
20.3. Configuration : les archétypes Zalta en mode lazy ... 235
20.4. Phase d'apprentissage : 01_learn.vvl ... 236

20.5. Génération à la demande : 02_generate.vvl 238
20.6. Validation batch et qualité 240
20.7. Les quatre niveaux de compréhension à l'œuvre 241
20.8. Synthèse : démonstration industrielle 243
21. Détection de fraude transactionnelle 244
21.1. Les contraintes du temps réel financier 244
21.2. L'ontologie bancaire : Transaction et attributs 245
21.3. Règles métier en cascade par priorité 245
21.4. Pipeline de bout en bout 247
21.5. Performance : cache sémantique et concurrence 248
Cache sémantique 248
Concurrence par pmap 249
21.6. Résilience : circuit-breaker et patterns avancés 250
21.7. Synthèse de la Partie VI : quatre cas industriels 251
22. Agents autonomes : [agent] et [architect] 254
22.1. De l'utilisateur à l'agent autonome 254
22.2. Rappel : l'opérateur @ pour la génération ponctuelle 255
22.3. La commande [agent] : boucle ReAct multi-étapes 257
Syntaxe et options 257
Le cycle ReAct interne 258
Exemples d'utilisation 259
22.4. La commande [architect] : orchestration single-shot 260
Pattern 1 — Agent pour le difficile, VVL pour le déterministe 261
Pattern 2 — Agent + opérateur @ pour le post-traitement 262
Pattern 3 — Agent dans une boucle (automatisation de tâches variées) 262
Pattern 4 — Architect qui délègue et que l'on capture 262
22.5. Étude de cas : analyse de contrat juridique 262
22.6. L'homoiconicité comme clé de voûte 264
22.7. Synthèse : trois niveaux pour une autonomie programmable 264
23. Le protocole d’atomes : taxonomie unifiée du runtime 266
23.1. Pourquoi un protocole transversal 266
23.2. Trois familles ontologiques mutuellement exclusives 268
Le cas hybride : VVL_ConceptEntity 268
23.3. Les huit axes métadonnées 269
23.4. Capabilities composables (anti fat-interface) 270
23.5. Trois niveaux d’identité orthogonaux 271
23.6. Builtins d’introspection 272
[atom:family $x] 272
[atom:axes $x] 273

[atom:capabilities $x] ... 273
[atom:trace $x] ... 273
23.7. Le DSL de policy par axes ... 274
[policy:define] ... 274
[policy:check $policy $atom] ... 275
23.8. Cas d'usage opérationnels ... 276
Cas 1 — Audit trail GDPR pour un concept réifié ... 276
Cas 2 — Sandbox multi-tenant déclarative ... 277
Cas 3 — Identité content-based pour cross-validation ... 277
Cas 4 — Filtre de déterminisme pour hot path ... 278
23.9. Synthèse : un protocole, dix optimisations émergentes ... 278
Statut et perspectives ... 280
24. Système expert et logique floue ... 281
24.1. Motivation : du programme écrit au programme décrit ... 281
24.2. Opérateurs flous et familles T-normes ... 282
24.3. Hedges linguistiques et agrégateurs ... 284
24.4. fuzzy:ontology — du domaine NL au schéma formel ... 284
* ... 285
24.5. fuzzy:rule — du langage naturel à la règle formelle ... 286
Shape A — règle « si X alors Y » ... 286
Shape B — dispatcher (énumération multi-alternatives) ... 286
24.6. expert — bootstrap one-shot d'un système expert ... 287
24.7. Persistance : expert:save / load / list ... 289
24.8. Étude de cas : triage de tickets support multi-décisions ... 290
24.9. Optimisations sous le capot ... 291
Cache de similarité sémantique ... 291
Extraction enum-aware ... 291
24.10. Synthèse : trois paradigmes d'écriture des règles ... 292
25. Optimisation par contraintes ... 294
25.1. Architecture : VVL conceptuel sur CP-SAT ... 294
25.2. Les formes du corps de problème ... 296
Types de décisions ... 296
Garanties opérationnelles ... 297
25.3. Le sous-ensemble compilable ... 297
Toujours compilable ... 297
Slog 3 — Matching flou ~~ ... 298
Auto-scaling des seuils ... 298
Non compilable — rejeté à validate ou compile ... 298
25.4. Du langage naturel au modèle compilé ... 300

25.5. Opérateurs flous résolus au compile-time ... 302
Composition de l'objectif (mathématique du SCALE) ... 302
25.6. Patterns avancés : multi-objectif, globales, multi-instance, paramétrique ... 303
Multi-objectif lexicographique ... 303
Contraintes globales natives CP-SAT ... 303
Multi-instance ontologique ... 303
Paramétrique : la forme [for] ... 304
Diagnostic d'infaisabilité ... 305
25.7. Routing géographique : geo:distance, geo:haversine, route:matrix ... 305
Repli au compile-time des builtins géographiques purs ... 306
Pattern paramétrique : tournée de techniciens ... 307
Prompt NL spécialisé routing ... 308
25.8. Trois niveaux d'expression ... 309
25.9. Synthèse : l'optimisation au sein de l'oscillation neuro-symbolique ... 310
26. Algorithmes génétiques : optimisation boîte-noire de structures discrètes ... 312
26.1. Pourquoi un solveur évolutionnaire complémentaire à CP-SAT ... 312
26.2. Le modèle évolutionnaire standard ... 314
26.3. Vue d'ensemble : les quatre phases du module ... 315
26.4. Phase 1 — Opérateurs déterministes et fitness holdout ... 316
26.5. Phase 2 — Mutations LLM avec budget et cost tracking ... 317
26.6. Phase 3 — Boucle adversariale par contre-exemples ... 317
26.7. Phase 4 — Multi-objectif et frontière de Pareto ... 319
26.8. Architecture interne et synergie avec le protocole d'atomes ... 320
26.9. Pattern end-to-end de production et synthèse ... 321
Tradeoffs et limites ... 322
Synthèse — la boucle se referme ... 322
27. Programmation logique inductive : apprentissage sound and complete depuis exemples ... 325
27.1. Trois voies d'apprentissage : pourquoi une troisième ... 326
27.2. Le modèle Learning-From-Failures ... 327
27.3. Le bias declaratif : modes, rôles, bornes ... 328
27.4. Phase ILP-1 — La boucle LFF en pure Python ... 329
27.5. θ-subsumption et ConstraintStore — l'audit trail formel ... 331
27.6. Phase ILP-2 — Bootstrap LLM (propose-bias, seed-hypotheses, induce) ... 331
[ilp:propose-bias] — un bias depuis du langage naturel ... 332
[ilp:seed-hypotheses] — N clauses candidates ... 332
[ilp:induce] — le pipeline complet en un appel ... 333
27.7. Pont avec le module genetic — refine et counter-examples ... 333
[ilp:refine-with-ga] — raffinement stochastique des winners ... 334
[ilp:learn-with-counter] — boucle adversariale par contre-exemples ... 334

27.8. Comparé à Popper, synergie avec le protocole d'atomes 335
27.9. Synthèse — la triade des modes d'apprentissage neuro-symbolique 336
Statut, limites, et perspectives 336
28. Méta-programmation, homoiconicité et helpers REBOL like 338
28.1. Le noyau réflexif derrière le livre 338
28.2. Code-comme-donnée : le triangle load / evaluate / dump 339
28.2.1. Du texte à la forme : [load] 340
28.2.2. De la forme à la valeur : [evaluate] et [execute] 341
28.2.3. Élever une expression brute : [quote] 342
28.2.4. Le retour : [dump] 342
28.3. Introspection : type-of, getType, inspectWord 344
28.3.1. Le nom de type stable : [type-of] 344
28.3.2. Inspection token-par-token : [inspectWord] 344
28.4. La frontière import / load / evaluate 345
28.5. [keyword] : la macro utilisateur, approfondissement 345
28.5.1. Anatomie d'une keyword 346
28.5.2. Keyword multi-arguments et calcul interne 346
28.5.3. La limite des quasiquotes imbriquées 346
28.5.4. Quand utiliser keyword, quand utiliser une fonction 347
28.6. Utilitaires de méta et de débogage 347
28.6.1. Aide en ligne : [help] 348
28.6.2. Logging structuré : log, logTable, logImage 348
28.6.3. Paramètres de tâche : [task-param] 348
28.6.4. Exploration du workspace : [file-tree] 349
28.7. [collect] / [keep] : génération de séries par effets de bord 349
28.7.1. Le pattern : un buffer caché 350
28.7.2. Isolation des collects imbriqués 350
28.7.3. Erreur : keep hors collect 351
28.8. [attempt] : la gestion d'erreur sérialisable 351
28.8.1. Anatomie d'un retour [attempt] 352
28.8.2. attempt vs try : la matrice de décision 352
28.8.3. La compatibilité ascendante : try:json 352
28.9. [object] : l'objet de données léger 353
28.9.1. object n'est pas une classe 353
28.9.2. Composition avec le triangle homoiconique 353
28.10. Synthèse : le noyau réflexif et le style de programmation 354
28.10.1. L'homoiconicité comme principe d'économie 354
28.10.2. Les trois niveaux d'évaluation 354
28.10.3. Les helpers REBOL comme parachèvement ergonomique 355

28.10.4. Vers le chapitre 29 355
29. Dialectes : parsing déclaratif et DSL embarqués 356
29.1. Du triangle homoiconique aux dialectes 356
29.1.1. match, load, parse : trois rôles distincts 356
29.1.2. Grammaire-comme-donnée 357
29.1.3. La filiation REBOL et la déviation VVL 358
29.2. Anatomie d'un [parse] 358
29.2.1. La signature 359
29.2.2. Un premier exemple complet 359
29.2.3. String input vs list input 360
29.3. Les règles primitives 361
29.3.1. Un premier exemple par règle 361
29.3.2. Le rôle pivot de [literal] 362
29.4. Les quantifieurs 362
29.4.1. some : un ou plus 363
29.4.2. any : zéro ou plus 363
29.4.3. opt : zéro ou un 364
29.4.4. repeat : compte exact ou borné 364
29.5. Les alternatives : [alt ...] 364
29.5.1. Pourquoi pas le pipe | 365
29.5.2. Ordre des alternatives et avidité 365
29.6. Captures : nommer les portions consommées 366
29.6.1. La syntaxe : [capture name rule] 366
29.6.2. Le type de la capture : string vs liste 366
29.6.3. Captures multiples et imbriquées 367
29.7. Dialectes nommés : un registre d'usage 368
29.7.1. Déclarer un dialecte : [dialect:def] 368
29.7.2. Utiliser un dialecte : [dialect:parse] 369
29.7.3. Inspecter le registre : [dialect:list] 369
29.7.4. Le pattern de bibliothèque de dialectes 370
29.8. Cas d'usage 371
29.8.1. Validation d'une sortie LLM 371
29.8.2. Décomposition d'une notation métier 372
29.8.3. Mini-DSL embarqué 372
29.9. parse vs match : la matrice de décision 373
29.10. Limites du MVP et perspectives 374
29.10.1. Pas de captures arborescentes nativement 374
29.10.2. Pas de backtracking sur les alternatives 374
29.10.3. Pas de | pour les alternatives 374

29.10.4. Pas encore de générateurs de parseurs depuis EBNF 374
29.10.5. Performance et taille des entrées 374
29.11. Synthèse : la grammaire au statut de donnée 375
29.11.1. Articulation avec les chapitres suivants 375
29.11.2. Un dernier coup d'œil 376
30. Modules : namespaces et packages réutilisables 377
30.1. La nécessité d'une frontière de package 377
30.1.1. Les limites d'[import] 377
30.1.2. Modules vs dialectes : commandes nommées vs données nommées 378
30.1.3. Modules vs plugins : code natif VVL vs code externe signé 378
30.2. Anatomie d'un module 379
30.2.1. Une première déclaration 379
30.2.2. Le scope isolé 380
30.2.3. Le corps s'exécute une fois — pas à chaque appel 381
30.2.4. La déclaration retourne le nom du module 381
30.3. Exports : valeurs et keywords 381
30.3.1. Accès aux valeurs : [module:get] 382
30.3.2. Appel aux keywords : la qualification Module:nom 382
30.3.3. Pourquoi deux mécanismes distincts 383
30.4. La qualification des keywords et la collision-safety 383
30.4.1. Le cas critique : exporter un nom de builtin 383
30.4.2. Priorités de résolution 384
30.4.3. Le pattern d'ouverture multi-modules 384
30.5. Chargement de fichiers : [module:load] et [module:reload] 385
30.5.1. La sémantique de base 385
30.5.2. Le cache et l'idempotence 385
30.5.3. Forcer le rechargement : [module:reload] 386
30.5.4. Pattern de structuration d'une bibliothèque 387
30.6. Modules vs import vs dialectes vs plugins 387
30.6.1. Quand choisir quoi 388
30.7. Cas d'usage 388
30.7.1. Bibliothèque mathématique 388
30.7.2. Logger avec niveaux et contexte 389
30.7.3. Mini-domain library avec composition de modules 390
30.8. Patterns avancés 391
30.8.1. Tests intégrés à la racine du module 391
30.8.2. Modules qui exportent des dialectes 392
30.9. Limites du MVP et perspectives 393
30.9.1. Pas de syntax « from import » 393

30.9.2. Le builtin [use] reste réservé ... 393
30.9.3. Pas de versioning ni de dépendances déclarées ... 393
30.9.4. Les builtins gardent leur priorité absolue ... 393
30.9.5. Pas de modules paramétrés ... 394
30.10. Synthèse : trois échelles de réutilisation ... 394
30.10.1. Les trois échelles ... 394
30.10.2. Frontière formelle au cœur du langage ... 394
30.10.3. Vers le chapitre 31 : valeurs scalaires typées ... 395
31. Valeurs scalaires typées : EMAIL, URL, DATE, UUID ... 396
31.1. Une lacune historique : le scalaire validé ... 396
31.1.1. Le problème des deux strings indistinguables ... 396
31.1.2. Trois stratégies préexistantes — et leurs limites ... 397
31.1.3. La quatrième stratégie : un type natif ... 397
31.2. Le contrat des valeurs typées ... 398
31.2.1. La construction : [TYPE payload] ... 398
31.2.2. La projection humaine : [form] ... 399
31.2.3. La projection canonique : [mold] ... 400
31.2.4. La projection JSON : .json ... 400
31.2.5. Le round-trip — la garantie fondamentale ... 400
31.3. EMAIL : l'adresse de courrier électronique ... 401
31.3.1. Validation ... 401
31.3.2. Le couple form / mold : la spécificité d'EMAIL ... 402
31.3.3. Pourquoi pas une validation RFC complète ... 402
31.4. URL : la ressource web ... 403
31.4.1. Schemas acceptés : http et https uniquement ... 403
31.4.2. Le contenu de l'URL est intégralement préservé ... 403
31.5. DATE : la date civile ... 404
31.5.1. Format strict : YYYY-MM-DD ... 404
31.5.2. Limites du MVP : pas de temps ni de fuseau ... 405
31.6. UUID : l'identifiant universellement unique ... 405
31.6.1. Format et normalisation ... 405
31.6.2. Génération vs construction ... 406
31.7. La grammaire <TYPE:payload> — le littéral préfixé ... 406
31.7.1. Anatomie du littéral ... 407
31.7.2. Pourquoi un préfixe explicite ... 407
31.8. type-of et getType : deux noms pour deux usages ... 407
31.8.1. Quand utiliser quoi ... 408
31.9. Valeurs typées vs ontologie vs dialecte ... 408
31.9.1. La division du travail ... 409

31.10. Cas d'usage 410
31.10.1. Validation d'un formulaire d'inscription 410
31.10.2. Filtrage d'une liste d'identifiants 411
31.10.3. Persistance avec round-trip garanti 411
31.11. Limites du MVP et perspectives 412
31.11.1. Quatre types seulement 412
31.11.2. Pas de types personnalisés utilisateur 412
31.11.3. Pas de paramétrage de validation 413
31.12. Synthèse : la validation au point de création 413
31.12.1. Le principe : valider tôt, valider une fois 413
31.12.2. Trois lieux de validation dans la Partie VIII 414
31.12.3. Vers le chapitre 32 : les tests intégrés 414
32. Tests intégrés et auditabilité formelle 415
32.1. Le quatrième lieu de validation 415
32.1.1. Pourquoi un système de test au cœur du langage 416
32.1.2. La proximité avec le code testé 417
32.2. Le principe non-throwing 417
32.2.1. Ce qu'une assertion classique fait 417
32.2.2. Ce que test:* fait à la place 418
32.2.3. test:* coexiste avec try et attempt 418
32.2.4. Quand l'assertion classique reste appropriée 419
32.3. [test:assert] : l'assertion booléenne 420
32.3.1. Signature et retour 420
32.3.2. La truthiness en VVL 420
32.3.3. Cas d'usage typiques 421
32.4. [test:equals] : l'égalité structurelle 421
32.4.1. Convention de l'ordre des arguments 421
32.4.2. L'égalité par type 422
32.4.3. Ce que test:equals ne fait pas 423
32.5. [test:suite] : grouper et compter 423
32.5.1. Signature et exécution 423
32.5.2. Le couple ok / failed 424
32.5.3. Imbrication des suites 424
32.6. [test:summary] : la photographie du contexte 425
32.6.1. Le format de retour 425
32.6.2. Cas d'usage : le rapport final 425
32.7. Le quartet en synthèse 426
32.8. Patterns d'écriture 426
32.8.1. Le tutoriel auto-validant 426

32.8.2. Le module avec tests collés 427
32.8.3. Le fichier de tests externes 428
32.8.4. Suite par fonctionnalité 428
32.9. Tests des systèmes apprenants 429
32.9.1. Audit d'une règle induite par LLM 429
32.9.2. Audit d'une stratégie évoluée par algorithme génétique 430
32.9.3. Audit d'une clause synthétisée par ILP 431
32.9.4. Test-driven learning 431
32.10. Limites du MVP et perspectives 432
32.10.1. Pas de mocks ni de stubs natifs 432
32.10.2. Pas de paramétrisation de tests 432
32.10.3. Pas d'isolation entre suites 433
32.10.4. Pas de timing ni de tests asynchrones 433
32.10.5. Pas d'assertions négatives spécialisées 433
32.11. Synthèse : le 4e lieu, et vers les plugins 433
32.11.1. Le sens du 4e lieu 433
32.11.2. Articulation avec le chapitre 33 : les plugins signés 434
32.11.3. La gradation complète 434
33. Plugins signés : Ed25519, subprocess/inproc, VScript 436
33.1. La frontière externe : le 5e lieu de validation 436
33.1.1. Modules vs plugins : la division du travail 436
33.1.2. Pourquoi une signature cryptographique 437
33.1.3. Ed25519 : le choix de l'algorithme 438
33.2. Le flux de confiance, vue d'ensemble 438
33.2.1. Les trois acteurs 439
33.2.2. Les deux vérifications cryptographiques 439
33.3. Le flux complet en commandes 440
33.3.1. Côté auteur : générer la clé et signer le bundle 440
33.3.2. Côté admin : importer la clé publique 441
33.3.3. Côté workspace : installer, configurer, charger 441
33.3.4. Persister les liaisons : le pattern init.vvl 442
33.4. Le magasin de clés de confiance 442
33.4.1. Les commandes du magasin 443
33.4.2. Les tiers standard et high 443
33.4.3. La révocation : [plugin-trust-remove] 443
33.5. Cycle de vie d'un plugin 444
33.5.1. Installation : [plugin-install] 444
33.5.2. Chargement de session : [plugin-load] 444
33.5.3. Désactivation : [plugin-unload] 445

33.5.4. Désinstallation : [plugin-remove] 445
33.5.5. Introspection : plugins-list et plugin-info 445
33.6. Modes d'exécution : subprocess vs inproc 446
33.6.1. Mode subprocess : l'isolation par processus 446
33.6.2. Mode inproc : la performance et ses contreparties 447
33.6.3. La double garde du mode inproc 447
33.6.4. Quand choisir quoi 448
33.7. Plugins VScript : le compromis sandboxé 448
33.7.1. Pourquoi un troisième langage 448
33.7.2. L'objet plugin en VScript 449
33.8. Connecteurs API : secrets, login, allowlist réseau 449
33.8.1. Déclaration dans le manifeste 449
33.8.2. Le coffre de secrets 450
33.8.3. Profils d'authentification 450
33.8.4. Audit : qui a fait quoi 451
33.9. Le SDK Python : stdlib-only 451
33.9.1. La structure minimale d'un plugin 451
33.9.2. Les helpers pour APIs externes 452
33.9.3. Le protocole sous-jacent 453
33.10. Limites du MVP et perspectives 453
33.10.1. Pas de sandbox OS pour les plugins Python 454
33.10.2. Windows et la limite d'isolation 454
33.10.3. Pas de mise à jour automatique 454
33.10.4. Renvoi à l'Annexe B pour la table exhaustive 454
33.11. Cas d'usage 455
33.11.1. Pont vers un modèle de langage 455
33.11.2. Connecteur pour base de données vectorielle 455
33.11.3. Export vers un format propriétaire 456
33.12. Synthèse : la chaîne fermée à cinq lieux 456
33.12.1. Une chaîne, pas un empilement 456
33.12.2. Modules + plugins : la complémentarité finale 457
33.12.3. Vers le chapitre 34 : la synthèse de l'ouvrage 457
34. Synthèse — VVL comme plateforme neuro-symbolique mûre 458
34.1. Cartographie de l'ouvrage 458
34.1.1. Trois grands arcs 458
34.1.2. Le rôle particulier de la Partie VIII 459
34.2. Les quatre paradigmes d'écriture des règles 459
34.2.1. Paradigme manuel : un humain écrit les règles 460
34.2.2. Paradigme neural : un LLM synthétise les règles 460

34.2.3. Paradigme symbolique : une recherche logique infère les règles 461
34.2.4. Paradigme bayésien : l'évidence calibre les poids 461
34.2.5. Comparaison synthétique 461
34.2.6. La voie hybride 462
34.3. La plateforme : sept strates en cohérence 462
34.3.1. Lecture verticale des strates 463
34.3.2. Lecture horizontale : ce que chaque strate garantit 463
34.4. Le pipeline canonique 463
34.4.1. Description du pipeline 464
34.4.2. Un exemple concret de bout en bout 464
34.4.3. La densité d'expression 466
34.5. Limites et perspectives 466
34.5.1. Limites du MVP infrastructurel 466
34.5.2. Limites des paradigmes apprenants 466
34.5.3. Limites cognitives plus profondes 467
34.5.4. Directions de recherche 467
34.6. Au lecteur — le mot final 467
34.6.1. Ce que le livre prétend 467
34.6.2. Trois invitations 468
34.6.3. Une dernière note — et un pont vers la Partie IX 468
35. Annotation stochastique bayésienne — la quatrième voie d'apprentissage 471
35.1. Position du module : couche probabiliste additive 471
35.1.1. Ce qu'apporte le module 471
35.1.2. Ce qui reste inchangé 472
35.1.3. Place dans l'ouvrage 472
35.2. Le glissement : du poids scalaire à la croyance Beta 472
35.2.1. La limite de l'approche scalaire 472
35.2.2. La solution : porter une distribution 473
35.2.3. Pourquoi Beta et pas une autre distribution 473
35.2.4. Aléatoire vs épistémique : la distinction enfin nommée 473
35.3. Cinq sources de priors 474
35.3.1. Le tableau des cinq sources 474
35.3.2. La logique d'élicitation 475
35.4. Le modèle génératif 475
35.4.1. Lecture descendante : de la croyance à l'observation 476
35.4.2. La conjugaison Beta-Bernoulli 477
35.4.3. Indépendance des observations 477
35.5. Annotation des clauses 477
35.5.1. La syntaxe préfixée p::clause 477

35.5.2. La syntaxe ?::clause pour l'ignorance totale ... 478
35.5.3. Annotation fine : [bannotate] ... 478
35.6. Inspection des croyances ... 479
35.6.1. [bposterior] : la croyance d'une clause ... 479
35.6.2. [bbeliefs] : la vue complète ... 480
35.7. L'apprentissage par échantillonnage de Gibbs ... 480
35.7.1. Signature et flux global ... 481
35.7.2. Le pas E : conditionnement ... 482
35.7.3. Le pas M : conjugaison Beta ... 482
35.7.4. La pertinence : ne créditer que ce qui a influé ... 483
35.7.5. La propriété clé : récupérer le paramètre latent ... 483
35.8. La négation est exacte ... 483
35.8.1. Pourquoi la négation est-elle un problème ... 483
35.8.2. Apprendre à travers la négation ... 484
35.8.3. Limite : programmes stratifiés ... 484
35.9. Prédiction avec propagation d'incertitude ... 485
35.9.1. [bprob] : propagation Beta et IC épistémique ... 485
35.9.2. Le calcul interne : double échantillonnage ... 486
35.9.3. [bapprox] : intégration arithmétique via VVL_Approximation ... 486
35.9.4. Confidence : exact / high / moderate / low ... 486
35.10. Mise à jour séquentielle et labels ... 487
35.10.1. Posterior → prior : la mise à jour séquentielle ... 487
35.10.2. Labels : variables aléatoires partagées ... 488
35.11. Articulation avec les autres paradigmes d'apprentissage ... 488
35.11.1. vs ILP (chapitre 27) : forme vs poids ... 489
35.11.2. vs LLM (chapitre 17) : bayes calibre les règles synthétisées ... 489
35.11.3. vs tests (chapitre 32) : oracle binaire vs oracle gradué ... 489
35.11.4. Les quatre paradigmes d'apprentissage ... 490
35.12. Architecture interne, rétro-compatibilité, limites ... 490
35.12.1. Cinq modules Python en table latérale ... 490
35.12.2. Rétro-compatibilité : le contrat ... 491
35.12.3. Le piège STEP_COUNTER ... 491
35.12.4. Limites et hypothèses ... 491
35.13. Synthèse — le quatrième paradigme et le mot final ... 492
35.13.1. Ce que le module ajoute à l'écosystème ... 492
35.13.2. Statut et intégration ... 492
35.13.3. Le mot final, à présent ici ... 492
36. Le calcul symbolique comme citoyen de première classe ... 494
36.1. Position du module : couche CAS additive ... 494

36.1.1. Trois utilisations qui justifient un CAS 494
36.1.2. Pourquoi SymPy spécifiquement 495
36.2. La lacune comblée : √2 reste √2 495
36.2.1. Précision préservée par construction 495
36.2.2. Résolution exacte, pas seulement numérique 496
36.3. Le type VVL_Sym, citoyen de première classe 496
36.3.1. Affichage et représentation 497
36.3.2. La frontière automatique entre les types 497
36.4. Les sucres syntaxiques [$ …] et [? …] 498
36.4.1. Forme [$ x] : déclarer un symbole 498
36.4.2. Forme [$ x y z] : déclarer plusieurs symboles 498
36.4.3. Forme [$ a*x^2 + b*x + c] : réifier une expression 499
36.4.4. Forme [? EXPR var?] : la résolution terse 499
36.4.5. Forme [$f arg] : appliquer une lambda symbolique 499
36.5. L'arithmétique native transparente 500
36.5.1. Le mécanisme interne : hook sur VVL_BinOp 500
36.5.2. Conséquence : le code reste lisible 501
36.6. La liaison paresseuse 501
36.6.1. Le mécanisme en action 501
36.6.2. Pourquoi la liaison paresseuse est précieuse 502
36.7. Catalogue des builtins — calcul élémentaire et algèbre linéaire 502
36.7.1. Famille « calcul élémentaire » 503
36.7.2. Famille « algèbre linéaire » 504
36.8. Catalogue des builtins — fonctions, ODE et pont 505
36.8.1. Famille « pont vers les autres types » 506
36.8.2. Le builtin sous-jacent [symbolic:parse] 506
36.9. Propriétés accessibles sur VVL_Sym 507
36.10. Le pont avec VVL_Approximation : la confiance « exact » 508
36.10.1. La sémantique de [symbolic:exact] 508
36.10.2. Pourquoi cette discipline est honnête 509
36.11. Intégration au langage naturel via [resolve] 510
36.11.1. La politique « symbolique par défaut » 510
36.11.2. Le mode estimation explicite 511
36.12. Sécurité du sandbox AST 512
36.12.1. La menace : exécution de code arbitraire 512
36.12.2. La défense : validation AST par liste blanche 512
36.13. Notes de syntaxe, pièges, et articulation finale 513
36.13.1. Synthèse et clôture 513
37. La preuve formelle native : vérifier ce que le LLM propose 515

37.1. Le critère de de Bruijn : trouver n'est pas vérifier ... 516
37.2. Un choix d'ingénierie : pur Python plutôt que Z3 ... 516
37.3. Couche 0 — la proposition comme donnée ... 516
37.4. Couche 2 — la décision automatique et le routeur de fragment ... 517
37.5. Couche 1 — le noyau de confiance ... 518
37.6. Couche 1bis — la boucle noyau : le LLM propose, le noyau dispose ... 519
37.7. Couche 2bis — entailment relationnel et audit de cohérence ... 520
37.8. Couche 3 — contrats de fonction et invariants de classe ... 521
37.9. Couche 4 — la provenance graduée : sortir du tout-ou-rien ... 522
37.10. Synthèse : la réponse au diagnostic du chapitre 1 ... 523
38. miniKanren embarqué : la logique relationnelle comme citoyen du langage ... 525
38.1. Les buts sont inertes, le résolveur résout ... 526
38.2. Unification, variables fraîches et termes ... 527
38.3. Conjonction, disjonction et complétude par entrelacement ... 528
38.4. Relations récursives et réversibles ... 529
38.5. Choix engagés et négation : le cut sans l'impureté ... 531
38.6. Le store de contraintes : diségalité, ordre, types, domaines ... 532
38.7. Contraintes symboliques : la CLP par SymPy ... 534
38.8. Le pont vers le KB Prolog ... 535
38.9. Le pont OO ↔ KB : la classe comme langage de requête ... 536
38.10. Logique relationnelle probabiliste, façon ProbLog ... 538
38.11. Le pont OO ↔ CSV : Active Record et requête sur table ... 540
38.12. eval relationnel : synthèse de programmes et quines ... 542
38.13. Synthèse : ce que la logique relationnelle ajoute à VVL ... 546
39. La VM Rust : le calcul symbolique passe natif ... 547
39.1. Une seule langue, un opérateur explicite ... 548
39.2. Le diagnostic de couverture ... 548
39.3. La campagne de parité et le catalogue des divergences ... 549
39.4. Arithmétique exacte : bignum, promotion, démotion ... 550
39.5. Un moteur Prolog SLD natif — et deux bugs révélés dans l'hôte ... 550
39.6. L'ontologie native, ou l'optimisation annulée volontairement ... 552
39.7. Métaprogrammation native : quote et match ... 554
39.8. Objet natif, données mutables et requêtes CSV ... 555
39.9. Fonctions compilées récursives, frames et optimisation d'appel terminal ... 556
39.10. La frontière hôte assumée ... 557
39.11. Versionnement double et durcissement ... 558
39.12. Synthèse : ce que prouve ce chantier ... 559
40. Le cluster de calcul : distribuer du VVL sur plusieurs machines ... 560
40.1. Le gate économique : mesurer avant de construire ... 560

40.2. Architecture : master, workers et code adressé par contenu 561
40.3. Enregistrer une fonction : la surface distribuable 562
40.4. Soumettre un calcul : shards, parallélisme et réducteurs 563
40.5. Quatorze builtins, quatre capabilities 564
40.6. Déploiement, TLS et rotation de secrets 565
40.7. Exploitation : métriques, audit, drain et sauvegarde 566
40.8. Les pannes attendues 566
40.9. Un cas réaliste : la tarification ontologique distribuée 567
40.10. Synthèse : distribuer sans renoncer au déterminisme 568
Conclusion générale 570
Annexe A — Référence des builtins VVL 571
A.1. Conventions fondamentales du langage 571
Affectation et variables 571
Booléens et égalité 571
Quasiquote et unquote 571
Interpolation de chaînes 572
Accesseurs pointés sur valeurs de base 572
A.2. Opérateurs arithmétiques, logiques et comparaisons 572
Opérateurs flous infix 573
A.3. Contrôle de flux 573
A.4. Fonctions d'ordre supérieur 574
A.5. Listes 574
A.6. Chaînes de caractères 575
A.7. Conversions de types et fonctions mathématiques 576
A.8. JSON et XML 577
A.9. Système de fichiers (jail-protégé) 577
A.10. Utilitaires PDF et images 578
A.11. Client et serveur HTTP 578
A.12. Programmation orientée objet 578
A.13. Fibers (coroutines coopératives) 579
A.14. Asynchronisme, promesses, channels et sémaphores 579
A.15. Administration : utilisateurs, workspaces, sessions 580
A.16. Logs 580
A.17. UI : prompts, dialogues, alertes 580
A.18. Méta : aide, introspection, évaluation 581
A.19. Utilitaires divers 582
A.20. Logique floue et bootstrap de système expert 582
Opérateurs flous et familles 582
Hedges linguistiques et agrégateurs 582

Système expert : ontologie et règles depuis NL ... 582
A.21. Commandes IA : sémantique, agent, architect ... 583
A.22. RAG (Retrieval-Augmented Generation) ... 583
Indexation et insertion ... 583
Requêtage ... 584
Gestion des KB et inspection ... 584
A.23. Base de connaissances Prolog ... 584
A.24. Système ontologique ... 585
Définition de schémas et classes ... 585
Extraction et instances ... 585
Inférence et règles ... 586
Mondes possibles (Zalta) ... 586
Requêtes et exports ... 586
Persistance et stores nommés ... 587
A.25. Réification de concepts (`@`) ... 587
A.26. Résolution NL→VVL (`§§`, `§§?`) ... 588
A.27. Catalogue des génériques (keywords paramétrés) ... 588
A.28. Persistance objet (SQLite + Prolog) ... 588
A.29. Calcul stack (Uiua-style) ... 589
A.30. CSV : core et extensions ... 589
Constructeurs ... 589
Manipulation et requêtage CSV ... 589
Manipulation des batchs ... 590
A.31. Office COM (Word, Excel, Outlook, PowerPoint) ... 590
Connexion et gestion de l'application ... 590
Word, Excel, Outlook, PowerPoint ... 590
A.32. Selenium WebDriver et scraping ... 591
A.33. Image : vision et génération (DALL-E) ... 591
A.34. Email (SMTP/IMAP/POP3) ... 592
A.35. Google APIs (Gmail, Calendar) ... 592
A.36. Hypervisor (Portainer / Docker) ... 593
A.37. MCP (Model Context Protocol) ... 593
A.38. Bases de données (SQLite, PostgreSQL) ... 593
A.39. Écosystème ProSuite — clients métier ... 593
A.40. Voice (OpenAI Realtime) et Whole (FFI VDOM) ... 594
Voice ... 594
Whole — FFI VDOM ... 594
A.41. Façades RAG spécialisées ... 594
A.42. Index alphabétique des principaux builtins ... 595

A.43. Pour aller plus loin 595
Annexe B — Système de plugins 597
B.1. Principes directeurs 597
Confiance ancrée cryptographiquement 597
Vérification à chaque étape — anti-TOCTOU 597
Code non signé : jamais d'exécution 598
Privilèges : opt-in, doublement gardés 598
B.2. Modèle cryptographique : Ed25519 et signature hors-ligne 598
B.3. Le bundle .vvlplugin et son manifeste 599
B.4. Le magasin de clés <PLUGIN_TRUST> 600
B.5. Cycle de vie : keygen → sign → install → load 601
5.1. Génération de la clé (auteur, hors-ligne) 601
5.2. Signature du bundle (auteur, hors-ligne) 601
5.3. Import de la clé publique (admin, sur le serveur) 601
5.4. Installation du plugin (workspace) 602
5.5. Chargement et invocation (per-session) 602
B.6. Garantie anti-TOCTOU : le record .registry.json 602
B.7. Modes d'exécution Python : subprocess et inproc 603
B.8. Plugins en VScript : sandbox natif et garde env 604
B.9. Dispatch et priorité dans le moteur 605
B.10. Référence des commandes `plugin-trust-*` (admin) 606
B.11. Référence des commandes de cycle de vie 607
B.12. Écrire un plugin Python : le SDK vvl_plugin_sdk 608
B.13. Récapitulatif des garanties de sécurité 608
B.14. Limites connues et perspectives 609
B.15. Carte des fichiers du sous-système 610
Annexe C — Le langage VScript 612
C.1. Nature et origine du langage 612
Une transpilation, pas un interpréteur arbre-marchant 612
Sandbox par construction 613
C.2. Les deux portes d'entrée dans VAILS 613
Forme plugin — lang: "vscript" 614
C.3. Contrat d'appel et marshalling VVL ↔ VScript 614
Entrée : la variable args, indexation 0-based 614
Sortie : la variable result 615
Marshalling automatique 615
C.4. Structure lexicale 616
C.5. Littéraux et constantes intrinsèques 616
Littéraux numériques 616

Littéraux chaîne et chaîne interpolée 616
Littéraux dates 617
Littéraux tableau et dictionnaire (Tier 1) 617
Constantes intrinsèques 617
C.6. Interpolation de chaîne et formatage par pipe 617
Forme générale 617
Échappements C-like — uniquement dans $"…" 618
Formatage par pipe — { expr | f | g(args) } 618
C.7. Subtypes et tests de type 618
C.8. Variables, constantes et déclarations 619
Dim, ReDim, Const, Erase 619
Affectation par valeur vs Set : la règle d'or 619
Passage de paramètres : ByVal et ByRef 620
C.9. Opérateurs et précédence 620
C.10. Court-circuit logique And et Or 621
La règle exacte — court-circuit limité aux gauches numériques 621
C.11. Coalescing ?? et optional chaining ?. 622
Coalescing ?? — valeur par défaut paresseuse 622
Optional chaining ?. — accès null-safe par maillon 622
C.12. Contrôle de flux 622
Conditionnelles : If et Select Case 622
Boucles : For, For Each, Do, While 623
Sorties anticipées : Exit et Return 623
C.13. Procédures : Sub, Function, Return moderne 624
Forme classique — nom_de_fonction = valeur 624
Forme moderne — Return [expr] 624
Assignations composées — sucre arithmétique 625
C.14. Programmation orientée objet : Class et Property 625
Déclaration d'une classe 625
Propriétés — Get, Let, Set 626
Héritage — Inherits, Me, MyBase, MyClass 626
Le bloc With — économiser des répétitions 627
D.16. Fonctions de première classe : AddressOf 628
D.17. Gestion d'erreurs : Try / Catch / Finally / Throw 628
Forme générale 628
Catch typé et binding 629
Throw — lever ou relancer 629
D.18. Bibliothèque standard 629
Conventions d'arguments à connaître 630

D.19. Méthodes membres fluides 630
D.20. Extension RegExp 631
Configuration de l'instance 631
Itération sur les occurrences 631
D.21. Extension XML — API pratique XPath 632
D.22. Bibliothèques réutilisables : [use $lib1 $lib2 …] 633
Syntaxe et exemple canonique 633
Règles de fusion 633
D.23. L'objet hôte ws — surface complète 634
ws.name — identifier le workspace courant 634
ws.files — fichiers, jailés à {ws}/files 634
ws.db(name) — SQLite jailée 635
ws.csv(name) — tables CSV de premier niveau 635
ws.mem(name) — magasin clé-valeur léger 636
ws.log msg — journalisation 636
ws.vvl(code) — rappel VVL depuis VScript 636
D.24. Interopérabilité OO — proxys vivants VVL ⇄ VScript 637
Sens 1 — instance VVL exposée à VScript 637
Le pont de casse — bestFriend ↔ bestfriend 638
Sens 2 — instance VScript exposée à VVL 638
D.25. Modernisations VScript — récapitulatif vs VBScript 639
D.26. Pièges classiques et déviations à connaître 640

Introduction — comment lire ce livre

Ce livre part d'un constat que la plupart des praticiens de l'intelligence artificielle ont fait sans toujours le formuler : les grands modèles de langage sont remarquables pour produire du plausible, et structurellement incapables de garantir du vrai. Ils hallucinent, ils n'exposent pas leur raisonnement, et rien en eux ne distingue une réponse fondée d'une réponse inventée. Symétriquement, les systèmes symboliques — Prolog, moteurs de règles, ontologies — garantissent la correction de ce qu'ils déduisent, mais se heurtent au coût de l'acquisition des connaissances : quelqu'un doit écrire les règles.

La programmation neuro-symbolique consiste à faire coopérer les deux. La thèse de cet ouvrage est qu'elle ne peut y parvenir durablement en assemblant des bibliothèques, mais seulement en devenant le paradigme natif d'un langage. C'est ce que VVL tente : un runtime unique où le LLM est un opérateur du langage, où la base Prolog et le système ontologique sont des citoyens de première classe, et où une valeur produite par un modèle circule sans conversion jusqu'au moteur d'inférence qui la vérifiera.

Le fil rouge du livre est une question pratique : d'où viennent les règles formelles qui dirigent un système ? Quatre réponses coexistent, et l'ouvrage les déroule l'une après l'autre — un humain les écrit, un LLM les synthétise, une recherche logique les infère, ou un modèle probabiliste en apprend les poids. Aucune n'est supérieure aux autres ; chacune convient à une situation, et elles se composent dans un même programme.

La carte de l'ouvrage

Le livre s'organise en neuf parties, dont l'ordre n'est pas arbitraire : chacune suppose acquise la précédente. La Partie I pose les fondements — l'histoire des deux paradigmes de l'IA, le vocabulaire théorique du raisonnement, et l'architecture de VAILS. La Partie II présente le langage lui-même : syntaxe, types, programmation fonctionnelle, système objet. La Partie III introduit ce qui fait la spécificité de VVL — le LLM intégré comme co-processeur, avec ses opérateurs sémantiques. La Partie IV expose le versant opposé, le pont symbolique vers Prolog, l'ontologie et l'algèbre de Zalta. La Partie V construit la liaison entre les deux : RAG, extraction structurée, ontologie dynamique, induction de règles.

Les quatre parties suivantes passent de la capacité à la pratique. La Partie VI formalise l'oscillation neuro-symbolique et la met à l'épreuve sur trois cas industriels complets — diagnostic médical, devisage BTP, détection de fraude. La Partie VII explore l'autonomie et l'optimisation : agents, protocole d'atomes, systèmes experts, contraintes, algorithmes génétiques, induction logique. La Partie VIII construit la plateforme qui rend tout cela utilisable en production : méta-programmation, dialectes, modules, valeurs typées, tests, plugins signés. La Partie IX, enfin, expose les sous-systèmes que le runtime a acquis le plus récemment : preuve formelle, logique relationnelle, machine virtuelle native, calcul distribué.

Trois parcours de lecture

Le lecteur qui veut évaluer l'approche avant d'y investir du temps peut se limiter aux chapitres 1 à 3, puis lire le chapitre 18 sur l'oscillation formalisée et l'un des trois cas industriels de la Partie VI. Ce parcours court donne la thèse, le mécanisme et une preuve par l'exemple.

Le praticien qui veut écrire du VVL suivra les parties I à V dans l'ordre, puis piochera dans les parties VII et VIII selon ses besoins. L'Annexe A lui servira de référence permanente : elle recense les builtins par domaine.

L'architecte ou le chercheur intéressé par les choix de conception trouvera dans les chapitres 23, 34, 37 et 39 les passages les plus argumentés — le protocole d'atomes, la synthèse des paradigmes, le critère de vérification, et le récit d'une optimisation annulée pour préserver un principe.

Conventions

Les exemples de code sont exécutables et leurs sorties sont celles réellement obtenues sur le serveur, indiquées en commentaire après une flèche. Les numéros de ligne servent au commentaire dans le texte et ne font pas partie du code. Les figures et tableaux portent un numéro de la forme chapitre.rang. Les renvois entre chapitres sont explicites : quand un mécanisme est présenté ailleurs, le chapitre est nommé.

Un mot sur les versions. Ce livre décrit VVL et VAILS tels qu'ils sont au moment de sa publication. Les sous-systèmes de la Partie IX sont les plus récents, et leurs limites sont documentées dans le corps du texte plutôt que passées sous silence — cette honnêteté sur ce que le système ne sait pas faire est, on le verra, la propriété qui rend le reste utilisable.

Partie I — Fondements

Chapitres 1 à 3 — d'où vient le neuro-symbolique, et pourquoi il exige un langage

1. Histoire des paradigmes de l'intelligence artificielle

Mots-clés : IA symbolique, connexionnisme, LLM, transformers, neuro-symbolique, Lisp, Prolog, Dreyfus, perceptron, deep learning

L'histoire de l'intelligence artificielle se lit comme une oscillation entre deux pôles : la conviction que l'intelligence est manipulation de symboles selon des règles formelles, et la conviction qu'elle émerge de l'apprentissage statistique sur des données. Soixante-dix ans après les premiers programmes de Newell et Simon, ce balancier n'a jamais cessé d'osciller, et c'est précisément cette tension qui rend nécessaire — et possible — la programmation neuro-symbolique. Comprendre VVL exige de comprendre d'où viennent les deux paradigmes qu'il fusionne, pourquoi chacun s'est imposé à son tour, et pourquoi aucun ne suffit seul.

Ce premier chapitre retrace cette histoire dans ses grandes lignes : l'âge d'or symbolique (1956–1980), l'hiver de l'IA, la renaissance connexionniste, l'irruption des grands modèles de langage, et enfin le constat — désormais largement partagé — que l'avenir n'appartient ni au symbolique pur ni au neuronal pur, mais à leur intégration profonde au sein d'un même langage.

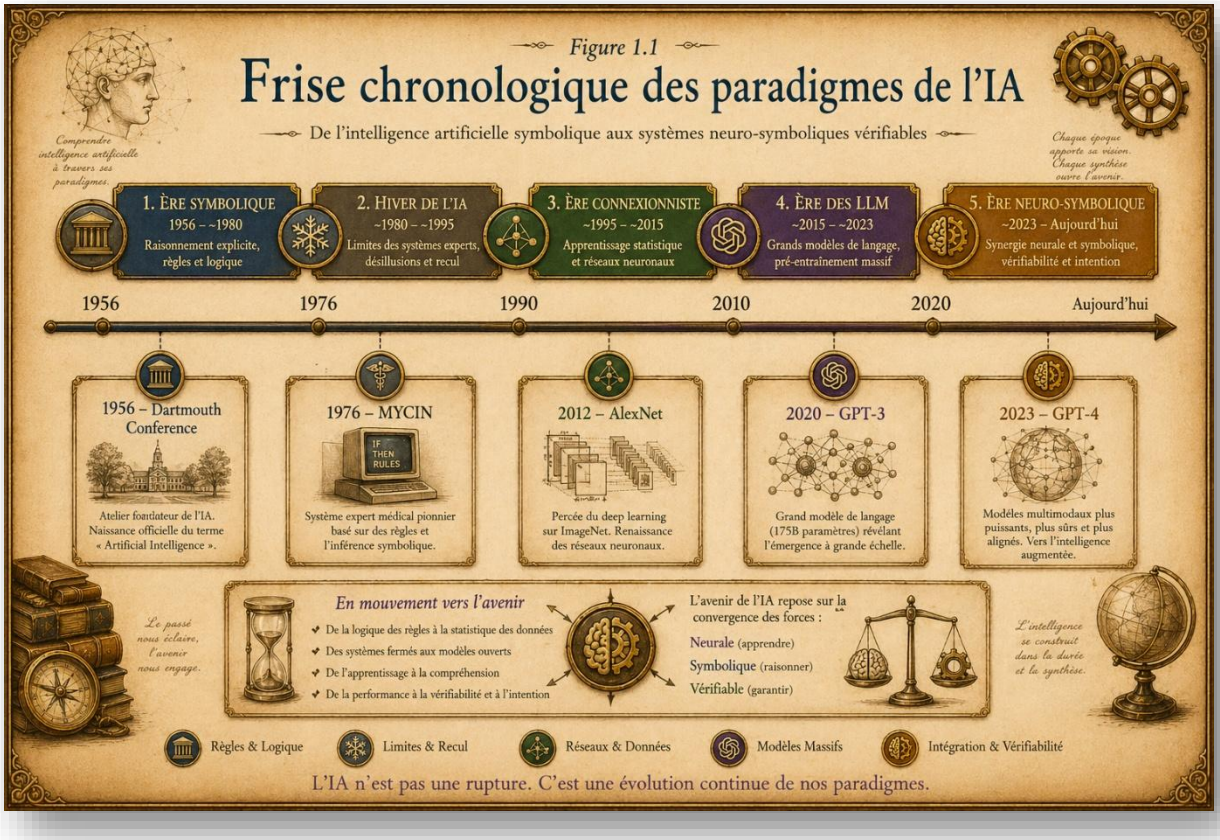


Figure 1.1 — *Frise chronologique des paradigmes de l'intelligence artificielle (1956–2026)*

1.1. L'IA symbolique : Lisp, Prolog et les systèmes experts (1956–1980)

La conférence de Dartmouth, durant l'été 1956, fonde l'intelligence artificielle comme champ de recherche autonome. John McCarthy, Marvin Minsky, Claude Shannon et Nathaniel Rochester y formulent une hypothèse audacieuse : *tout aspect de l'apprentissage ou de l'intelligence peut, en principe, être décrit avec une précision telle qu'une machine puisse le simuler*. Cette hypothèse, dite **hypothèse du système physique de symboles**, sera théorisée plus tard par Newell et Simon : un système qui manipule des symboles selon des règles formelles dispose des moyens nécessaires et suffisants pour produire un comportement intelligent.

De cette hypothèse découlent les deux langages emblématiques de l'IA symbolique. **Lisp**, conçu par McCarthy en 1958, repose sur l'idée que les listes peuvent représenter à la fois les données et les programmes — c'est l'**homoiconicité**, propriété que VVL hérite directement. **Prolog**, créé par Colmerauer et Kowalski au début des années 1970, pousse la logique encore plus loin : un programme est un ensemble de faits et de règles, et l'exécution est un processus d'unification et de chaînage arrière effectué par le moteur du langage. Le programmeur déclare *ce qui est vrai* ; l'ordinateur déduit *ce qui en découle*.

L'aboutissement industriel de ce paradigme est le **système expert**. MYCIN (1976) diagnostique les infections bactériennes à partir d'un corpus de plusieurs centaines de règles écrites avec des médecins ; DENDRAL identifie les structures moléculaires depuis les spectres de masse ; XCON, développé chez Digital Equipment, configure les commandes de mini-ordinateurs VAX et économise des dizaines de millions de dollars à la fin des années 1980. Pour la première fois, la programmation logique génère de la valeur économique mesurable.

Voici un programme Prolog miniature qui illustre la puissance déclarative de cette approche — il définit la relation d'ancêtre par récurrence et permet d'interroger une généalogie :

```
% Faits : relations parent-enfant directes
parent(marie, jean).
parent(jean, paul).
parent(paul, alice).

% Regle recursive : ancetre = parent ou ancetre d'un parent
ancetre(X, Y) :- parent(X, Y).
ancetre(X, Y) :- parent(X, Z), ancetre(Z, Y).

% Requete : qui sont les ancetres d'Alice ?
% ?- ancetre(A, alice).
% A = paul ; A = jean ; A = marie.
```

La beauté de cet exemple tient en deux observations. Premièrement, le programmeur n'a écrit aucune boucle : la récursion est une simple relation logique. Deuxièmement, la même règle sert à plusieurs requêtes — *qui sont les ancêtres d'Alice ?*, *Marie est-elle une ancêtre de Paul ?*, *de qui Marie est-elle ancêtre ?* — sans modifier une seule ligne. C'est la promesse fondamentale du paradigme symbolique : un programme exprime un *savoir*, et le moteur d'inférence en extrait toutes les *conséquences*.

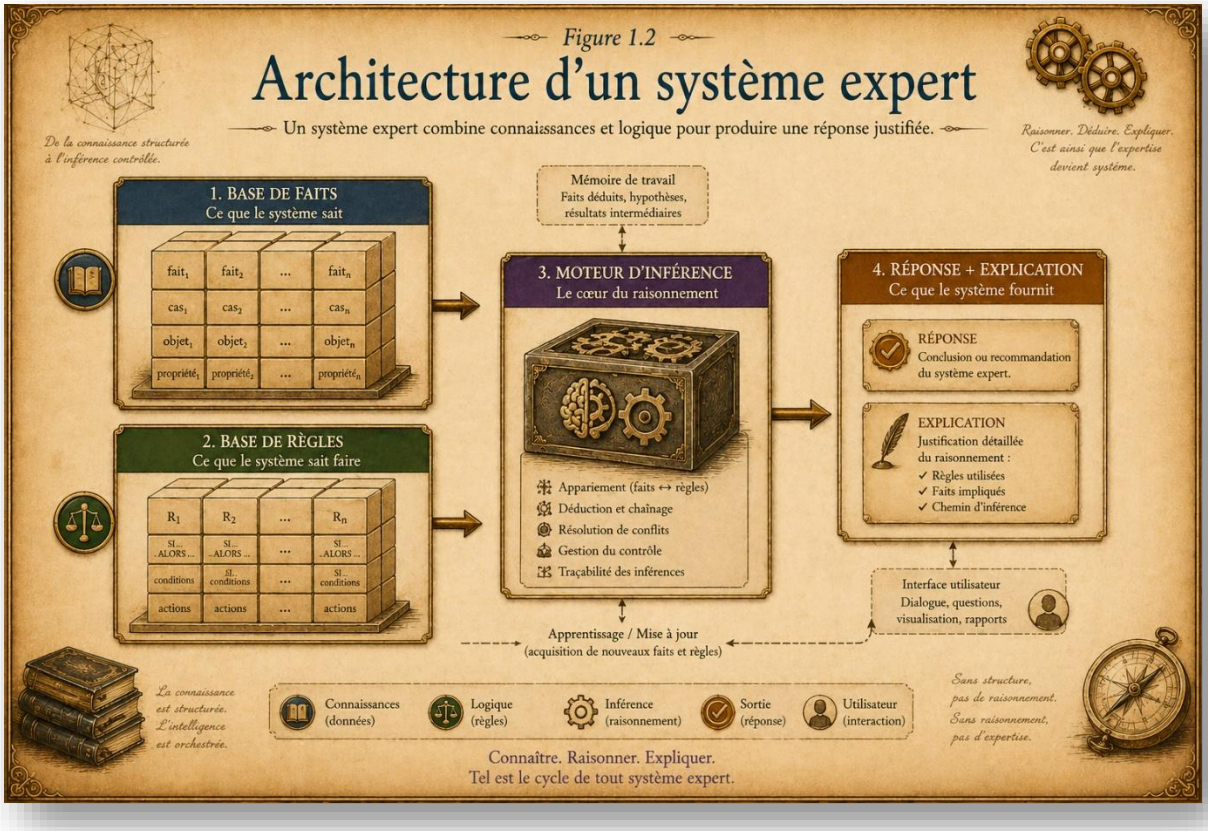


Figure 1.2 — *Architecture en trois blocs : faits, règles, moteur d'inférence*

1.2. L'hiver de l'IA et la critique de Dreyfus

La fin des années 1980 marque la fin de l'enthousiasme. Les systèmes experts atteignent un plafond : leur construction est lente (un expert humain doit verbaliser ses règles, un cogniticien doit les formaliser), leur maintenance est coûteuse (chaque évolution du domaine exige une réécriture), et surtout leur **généralisation est nulle**. Un système qui diagnostique les infections bactériennes ne sait rien faire d'autre. Il ne reconnaît pas le visage d'un patient, ne comprend pas la phrase qu'on lui adresse, ne perçoit pas qu'une radiographie est floue.

Le philosophe **Hubert Dreyfus**, dès 1965 dans *Alchemy and Artificial Intelligence* puis dans *What Computers Can't Do* (1972), avait formulé la critique théorique : la cognition humaine repose massivement sur des **savoirs tacites** — savoir-faire perceptifs, sens commun, intuition incarnée — qu'aucune liste finie de règles explicites ne peut épuiser. Reconnaître un visage, comprendre une plaisanterie, naviguer dans une foule : aucune de ces compétences ne se laisse réduire à un système de production. Le pari de l'IA symbolique, selon Dreyfus, était inspiré d'une mauvaise philosophie de l'esprit héritée de Descartes et Leibniz.

Les financements s'effondrent. La DARPA réduit drastiquement ses crédits, le marché des machines Lisp s'écroule, et le terme même d'« intelligence artificielle » devient suspect au point que beaucoup de chercheurs préfèrent parler de *machine learning*, de *systèmes experts* ou de *traitement des connaissances* pour ne pas être associés à un champ jugé en faillite. C'est l'**hiver de l'IA**, qui durera, par vagues, jusqu'au milieu des années 2000.

1.3. La renaissance connexionniste : du perceptron au deep learning

Pendant que l'IA symbolique perd ses crédits, une autre tradition, plus discrète, prépare son retour. Le **perceptron** de Rosenblatt (1958) avait été le premier modèle algorithmique d'apprentissage à partir d'exemples — un classifieur linéaire entraîné par correction d'erreur. Minsky et Papert l'avaient enterré en 1969 en démontrant qu'il ne pouvait apprendre la fonction XOR. Mais l'idée survit : Hopfield (1982), Rumelhart, Hinton et Williams (1986) avec la **rétropropagation** sur les réseaux multi-couches, LeCun (1989) avec les premiers réseaux convolutifs appliqués à la reconnaissance de chiffres manuscrits.

Trois facteurs convergent au tournant des années 2010 pour faire basculer le rapport de force. D'abord, la disponibilité de jeux de données massifs étiquetés (ImageNet en 2009 : quatorze millions d'images annotées). Ensuite, la puissance de calcul des GPU, conçus à l'origine pour les jeux vidéo mais idéalement adaptés aux multiplications matricielles. Enfin, les progrès algorithmiques (initialisation Xavier, ReLU, dropout, batch normalization) qui rendent enfin praticable l'entraînement de réseaux profonds.

La rupture publique a lieu en 2012 : **AlexNet**, un réseau convolutif profond entraîné sur GPU, remporte le concours ImageNet avec un taux d'erreur de 15,3 % contre 26 % pour le meilleur système classique. L'écart est tel qu'il rend instantanément obsolètes vingt ans de recherche en vision par ordinateur fondée sur des descripteurs faits main. Dans les cinq années qui suivent, le deep learning balaie successivement la reconnaissance d'images, la reconnaissance vocale, la traduction automatique.

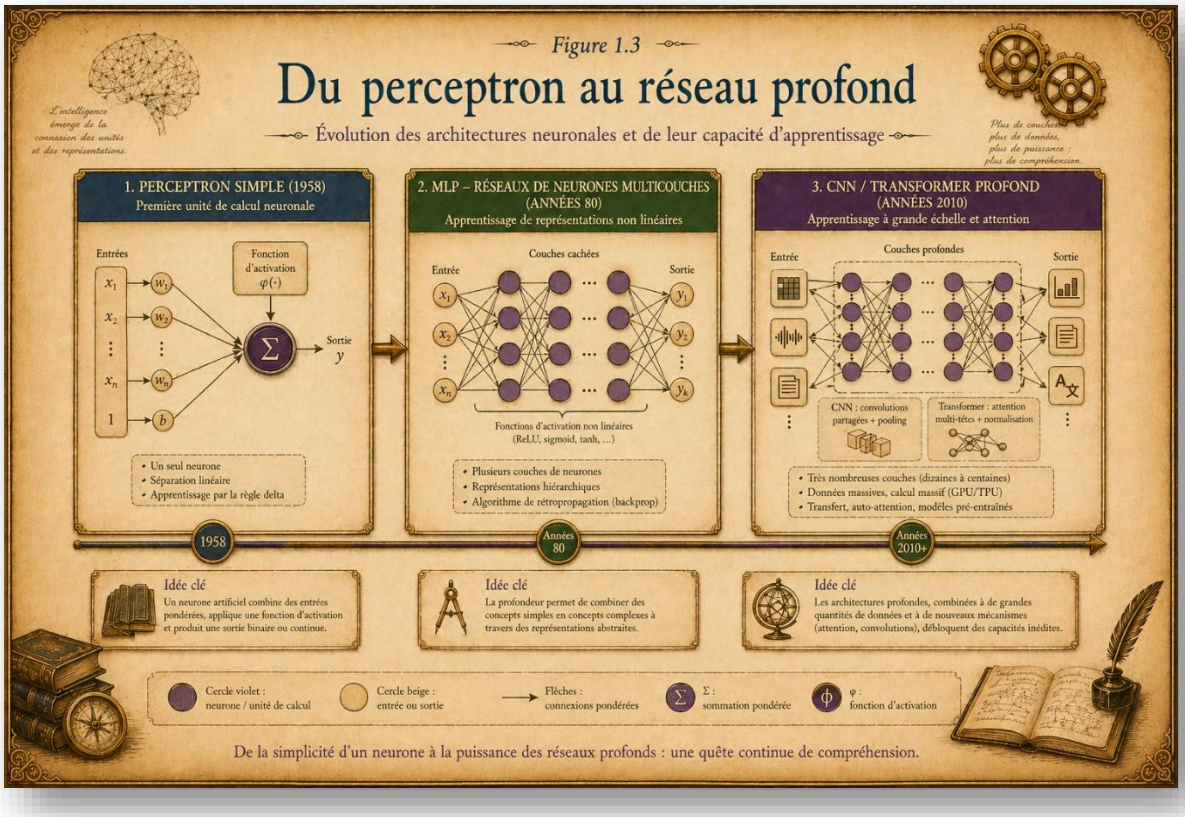


Figure 1.3 — *Évolution architecturale : du perceptron monocouche au réseau convolutif profond*

Le tableau ci-dessous résume la rupture méthodologique :

Aspect	IA symbolique	Connexionnisme
Représentation	Symboles discrets, règles	Vecteurs continus, poids
Acquisition	Codage manuel des règles	Apprentissage sur données
Inférence	Chaînage logique, déterministe	Propagation, probabiliste
Explicabilité	Élevée (trace de règles)	Faible (boîte noire)
Généralisation	Quasi nulle hors domaine	Forte sur distribution apprise
Robustesse au bruit	Faible	Élevée

Figure 1.4 — *Comparaison des deux paradigmes historiques de l'intelligence artificielle*

1.4. L'ère des transformers : GPT et le raisonnement statistique émergent

L'année 2017 marque une seconde rupture majeure avec l'article *Attention Is All You Need* de Vaswani et collègues, qui introduit l'architecture **Transformer**. En remplaçant les réseaux récurrents par un mécanisme d'attention multi-têtes massivement parallélisable, les Transformers rendent praticable l'entraînement sur des corpus de l'ordre du téraoctet, ce qui était impensable jusqu'alors.

Sur cette architecture se construisent les **grands modèles de langage** : BERT (2018) pour la compréhension, GPT-2 (2019) puis GPT-3 (2020) pour la génération, GPT-4 (2023) qui franchit les seuils de la complétion de code, du raisonnement multi-étapes et de la résolution de problèmes mathématiques. La caractéristique frappante de ces modèles est l'**émergence** : à partir d'une certaine échelle, des comportements qualitativement nouveaux apparaissent — capacité à suivre des instructions en langage naturel, à raisonner par analogie, à apprendre en contexte à partir de quelques exemples (*few-shot learning*).

Le LLM réalise, à sa manière, le rêve initial de l'IA : un système unique qui répond à des questions, traduit des langues, écrit du code, raisonne sur des situations nouvelles. Mais il le fait par un chemin radicalement opposé à celui des systèmes experts. Là où MYCIN appliquait des règles formelles à des faits structurés, GPT prédit le prochain token à partir d'un contexte de tokens précédents, sans aucune représentation explicite des concepts manipulés. Le savoir est *encodé dans les poids* du réseau, distribué sur des milliards de paramètres, sans qu'on puisse pointer où réside telle connaissance précise.

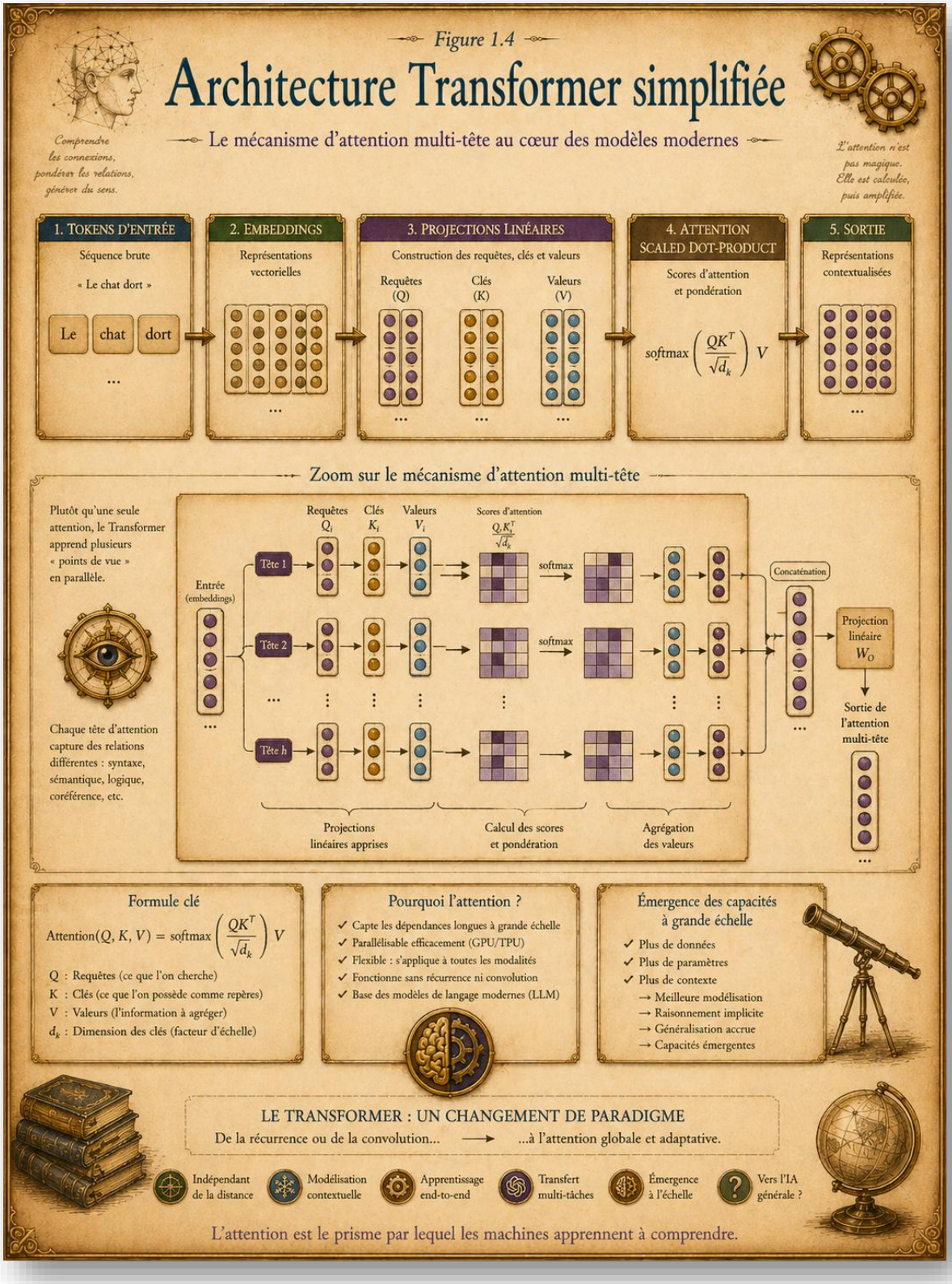


Figure 1.5 — *Architecture Transformer et émergence des capacités à grande échelle*

1.5. Les limites des LLM purs : hallucinations, opacité, non-vérifiabilité

Cette puissance s'accompagne de limites structurelles que la simple augmentation d'échelle ne fait pas disparaître.

Les hallucinations sont le symptôme le plus visible. Un LLM ne distingue pas, en interne, ce qu'il sait de ce qu'il invente : il produit la complétion la plus probable étant donné le contexte, qu'elle soit factuellement correcte ou entièrement fabriquée. Demander à GPT-4 « Quel est le titre du dernier article de tel chercheur ? » peut produire une référence parfaitement formatée, avec auteurs, année, journal et numéro de page — mais entièrement fictive. Le modèle ne ment pas au sens où il aurait l'intention de tromper : il génère, dans son espace de probabilités, la séquence qui ressemble le plus à une réponse plausible.

L'opacité est le second problème. Quand un système expert recommandait un antibiotique, MYCIN pouvait afficher la chaîne de règles ayant conduit à la décision : *règle 47 a tiré R1, règle 184 a tiré R2, donc conclusion C*. Un LLM ne dispose d'aucune trace équivalente. La justification qu'il produit *a posteriori* — appelée *chain-of-thought* — est elle-même une génération probabiliste, qui peut diverger de la « raison » ayant conduit à la réponse. Dans les domaines régulés (médecine, finance, droit), cette absence de traçabilité est rédhibitoire.

La non-vérifiabilité en est le corollaire pratique. Aucun mécanisme interne au LLM ne garantit que la réponse satisfait une contrainte donnée — qu'elle est factuellement vraie, juridiquement valide, ou même cohérente avec les phrases précédentes du même texte. Pour s'en assurer, il faut *rajouter* une couche externe : un validateur, un retrieval-augmented generator, un système de règles. C'est exactement le diagnostic dont part la programmation neuro-symbolique.

Enfin, **la connaissance des LLM est figée à la date d'entraînement** et leur fenêtre de contexte, bien qu'élargie à plusieurs centaines de milliers de tokens, reste finie. Pour rendre un LLM opérationnel sur un corpus métier, il faut un mécanisme externe d'injection de contexte — typiquement le RAG, dont nous verrons au chapitre 14 qu'il est lui-même intégré nativement à VVL.

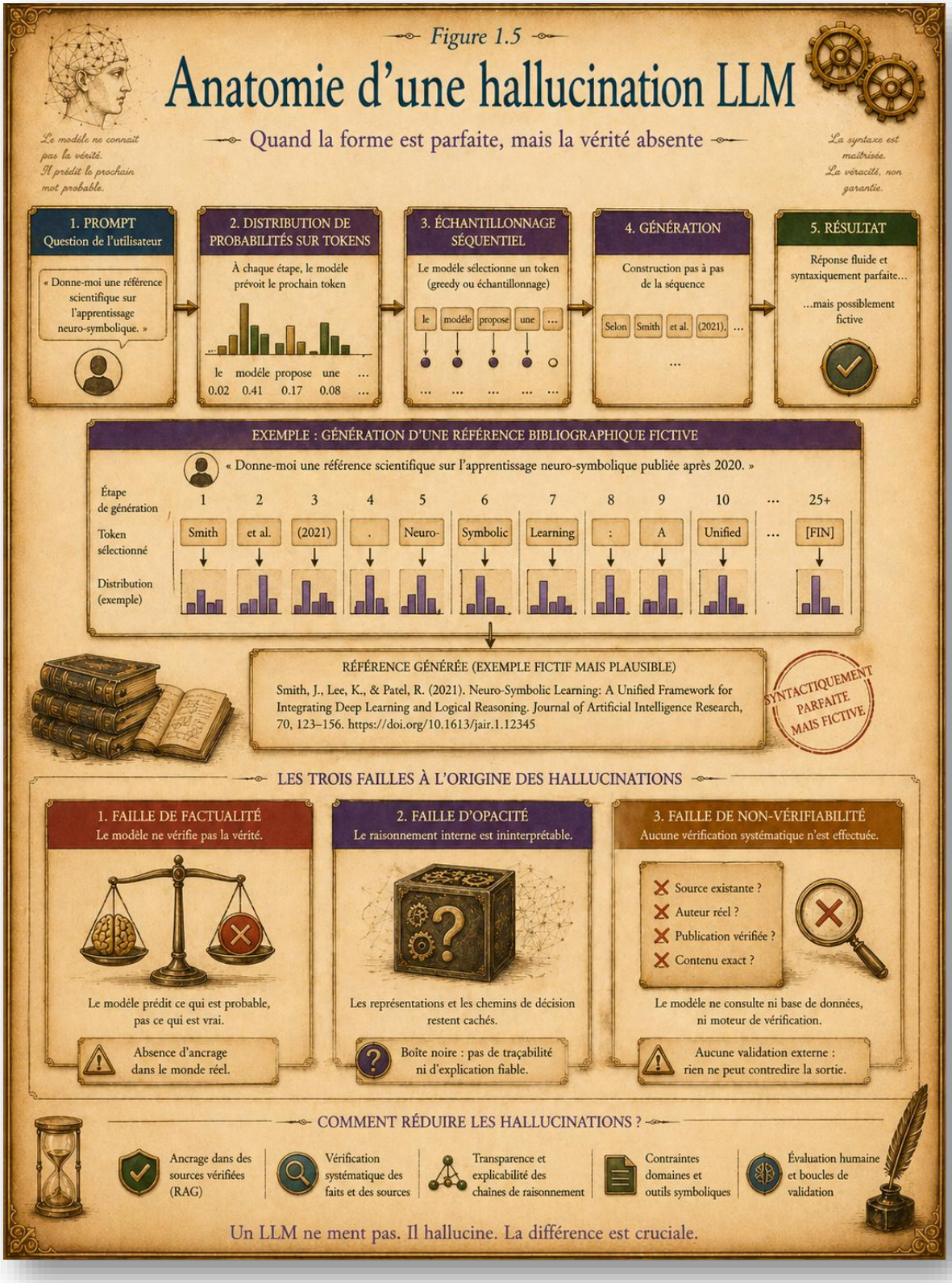


Figure 1.6 — *Mécanisme d'une hallucination : génération plausible sans ancrage factuel*

1.6. Le retour du symbolique et la fusion neuro-symbolique

Face à ces limites, deux stratégies s'affrontent. La première parie qu'augmenter encore la taille des modèles, la qualité des données et le raffinement de l'apprentissage par renforcement résoudra progressivement les hallucinations et fera émerger un raisonnement véritable. C'est la voie *scale is all you need*, défendue par les principaux laboratoires industriels.

La seconde stratégie, défendue par une partie croissante de la recherche académique — Gary Marcus, Henry Kautz, Bart Selman, Luis Lamb, Artur d'Avila Garcez — tient que les limites des LLM ne sont pas des défauts à corriger mais des **propriétés intrinsèques** du paradigme statistique. Aucun modèle de prédiction de tokens, aussi grand soit-il, ne peut intrinsèquement garantir la cohérence logique, la traçabilité ou l'absence d'hallucinations, parce que ces propriétés ne sont pas définissables dans l'espace des probabilités sur les séquences. Elles requièrent une représentation symbolique.

La proposition neuro-symbolique consiste donc à **réintroduire la couche symbolique non pas en remplacement du neuronal, mais en complément**. Le LLM devient le moteur d'extraction et de génération — il transforme du texte non structuré en faits, il génère du code, il propose des hypothèses. La couche symbolique (logique, règles, ontologies) devient le moteur de validation, d'inférence et d'explication — elle vérifie la cohérence, déduit les conséquences, trace les décisions.

Plusieurs architectures académiques explorent cette voie depuis le milieu des années 2010 : DeepProbLog (Manhaeve et al., 2018) intègre des prédicats Prolog différenciables ; Logic Tensor Networks (Serafini & Garcez, 2016) encode la logique floue dans des réseaux de neurones ; NeurASP (Yang et al., 2020) combine answer set programming et perception neuronale. Chacune de ces approches reste cependant un système de recherche : elles exigent l'assemblage de plusieurs frameworks, d'experts dans deux paradigmes, et leur déploiement industriel demeure exceptionnel.

VVL adopte une autre stratégie : faire du neuro-symbolique non pas une technique d'intégration d'outils, mais le paradigme natif d'un langage de programmation. L'opérateur sémantique (chapitre 7) inscrit le LLM dans la grammaire même du langage, au même titre que `+` ou `==`. Le système ontologique (chapitre 12) réintègre Prolog comme moteur de raisonnement directement accessible par l'interpréteur. Les opérateurs de Zalta (chapitre 13), fondés sur la théorie des objets abstraits exposée au chapitre 2, fournissent le pont conceptuel entre les concepts amorphes du LLM et les objets formels manipulables par la logique. Le tout dans un seul runtime, une seule base de connaissances, une seule syntaxe.

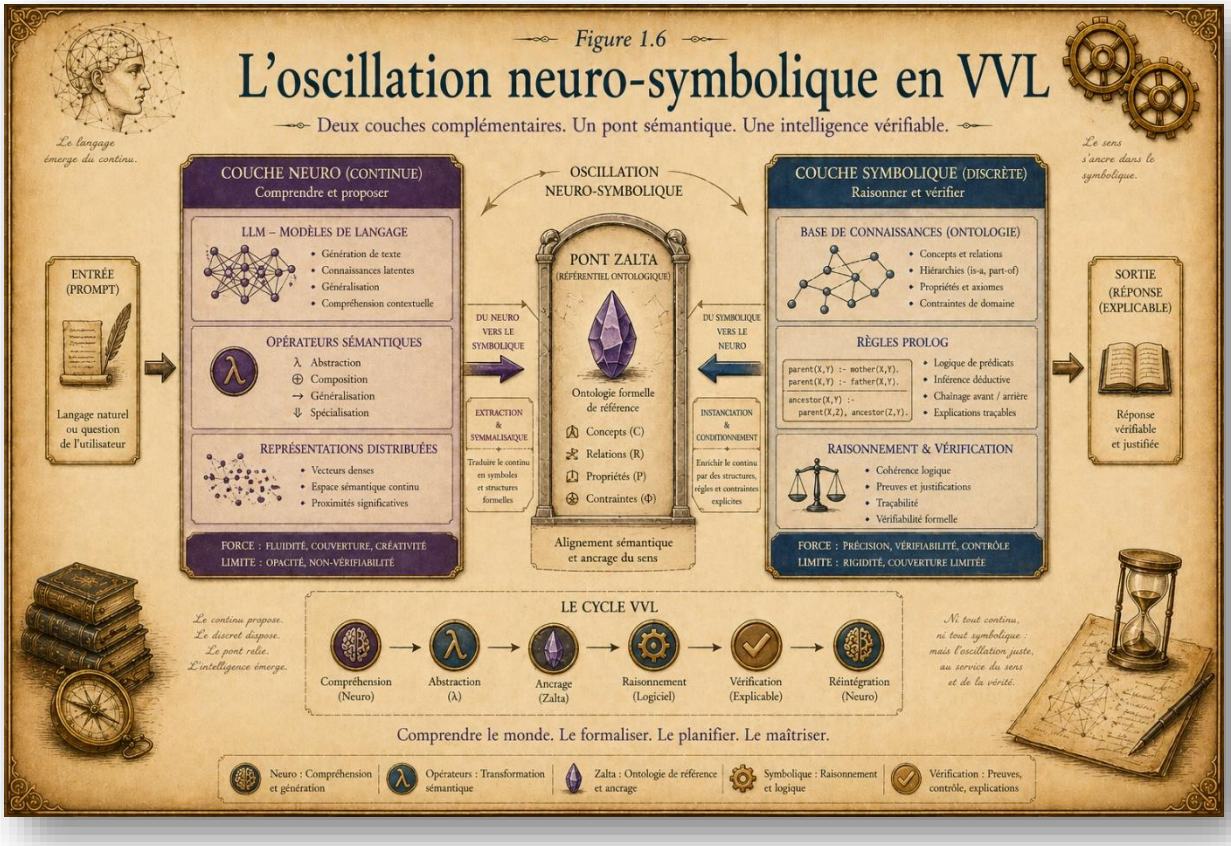


Figure 1.7 — *L'architecture neuro-symbolique de VVL : trois couches, un seul runtime*

Le tableau suivant résume les trois âges de l'IA et la place que VVL occupe dans cette histoire :

Période	Paradigme dominant	Limite atteinte	Réponse VVL
1956–1985	IA symbolique (Lisp, Prolog)	Pas d'apprentissage, fragilité	Conserve la rigueur logique
1986–2016	Apprentissage statistique, deep learning	Opacité, peu de raisonnement	Conserve l'extraction par LLM
2017–aujourd'hui	LLM (transformers)	Hallucinations, non-vérifiabilité	Couple LLM + Prolog + ontologie
Avenir	Neuro-symbolique natif	—	Un seul langage, un seul runtime

Figure 1.8 — *Les trois âges de l'IA et la position de VVL dans cette généalogie*

Les chapitres qui suivent vont reprendre cette généalogie point par point. Le chapitre 2 expose les fondements théoriques — logique du premier ordre, induction, théorie des objets abstraits — qui sous-tendent les opérateurs de VVL. Le chapitre 3 présente la vision architecturale du langage et le situe par rapport aux frameworks d'intégration existants. Les Parties II à V dérouleront ensuite, brique par brique, les trois piliers de la programmation neuro-symbolique en VVL : un langage homoiconique solide, un LLM intégré comme co-processeur du langage, et un pont symbolique vers Prolog et l'ontologie.

2. Fondements théoriques du neuro-symbolique

Mots-clés : logique du premier ordre, déduction, induction, abduction, ILP, Edward Zalta, objets abstraits, double prédication, DeepProbLog, NeurASP, LTN

Le chapitre précédent a retracé l'histoire des paradigmes ; celui-ci en pose les fondements théoriques. Avant de plonger dans la syntaxe et les opérateurs de VVL, il est essentiel d'avoir un vocabulaire précis pour les trois modes de raisonnement (déduction, induction, abduction), pour la logique formelle qui les sous-tend, et pour la théorie métaphysique — celle d'Edward Zalta — qui fournit le pont conceptuel entre concepts neuronaux amorphes et objets symboliques manipulables.

Ce chapitre n'est pas un cours complet de logique mathématique : c'est l'inventaire minimal des outils théoriques qui éclairent les choix de conception de VVL. Chaque section pointera vers le chapitre où l'idée trouve son expression concrète — les opérateurs sémantiques au chapitre 7, l'ontologie au chapitre 12, les opérateurs Zalta au chapitre 13, l'apprentissage inductif au chapitre 17.

2.1. Raisonnement déductif : logique du premier ordre et chaînage

La **déduction** est le mode de raisonnement qui va du général au particulier. Si l'on accepte les prémisses, on doit accepter la conclusion : c'est ce qui distingue la déduction des deux autres modes. Le syllogisme aristotélicien en donne la forme la plus simple :

Tous les hommes sont mortels. Socrate est un homme. Donc Socrate est mortel.

La logique du premier ordre, formalisée à la fin du XIX^{e} siècle par Frege puis Peano, généralise ce schéma. Elle introduit les **prédicats** (relations entre objets : `mortel(x)`, `parent(x, y)`), les **quantificateurs** universel ∀ et existentiel ∃, et les **connecteurs** logiques (∧, ∨, ¬, →). Le syllogisme s'écrit alors :

`∀x (homme(x) → mortel(x)) ∧ homme(socrate) ⊢ mortel(socrate)`

Le symbole ⊢ désigne la dérivation syntaxique : la conclusion peut être obtenue par application mécanique de règles d'inférence, sans appel à la signification des symboles. C'est cette propriété qui rend la déduction *automatisable* : un moteur d'inférence n'a pas besoin de comprendre ce que veut dire « mortel » pour appliquer correctement les règles.

Deux règles d'inférence dominent la pratique : le **modus ponens** (de `A → B` et `A`, conclure `B`) et le **modus tollens** (de `A → B` et `¬B`, conclure `¬A`). Toutes les autres formes de raisonnement déductif peuvent en principe se ramener à ces deux schémas, éventuellement combinés avec la généralisation et l'instanciation.

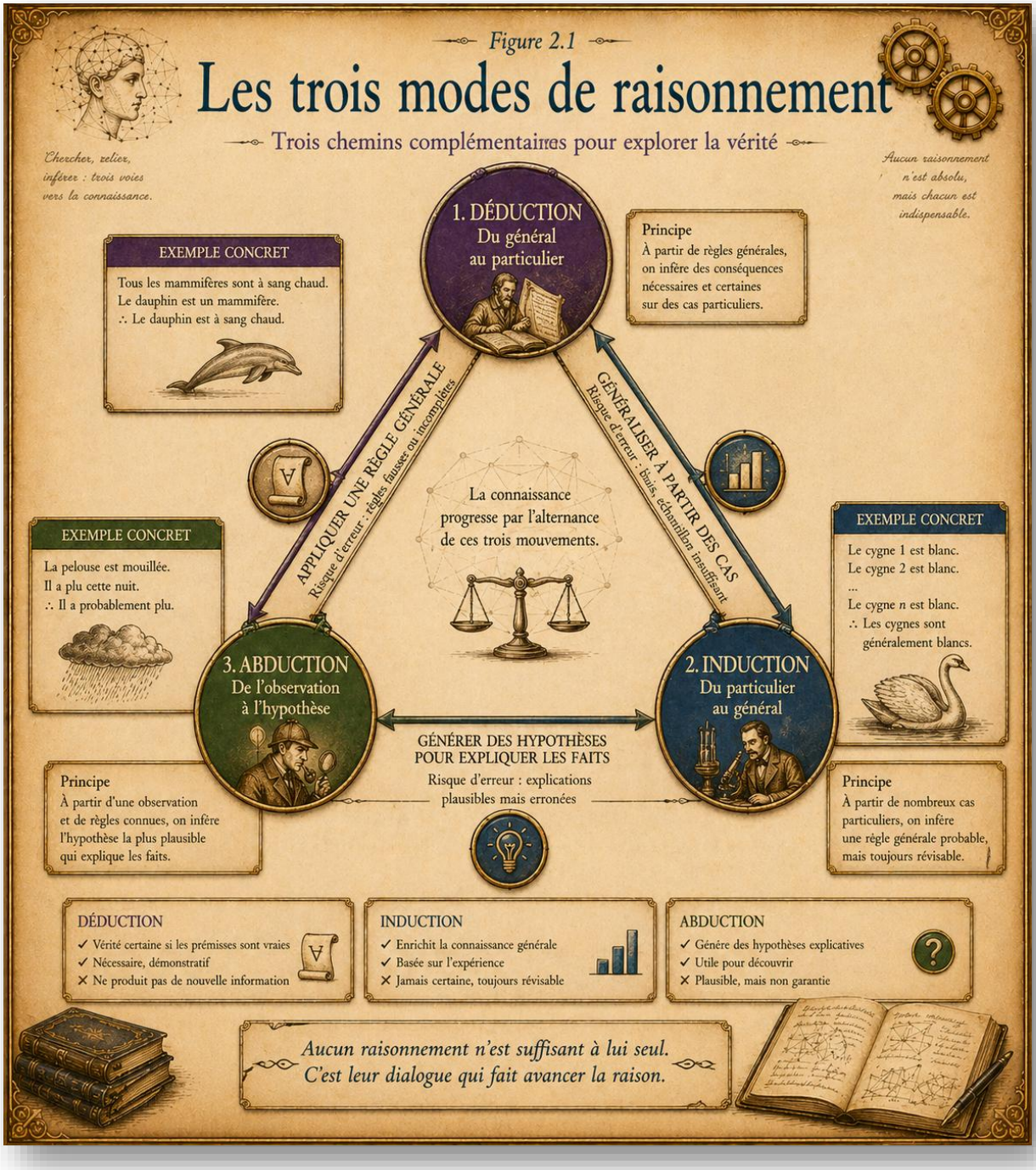


Figure 2.1 — *Les trois modes de raisonnement et leur articulation*

En programmation logique, deux stratégies opérationnelles implémentent la déduction. Le **chaînage avant** (forward chaining) part des faits connus, applique les règles, déduit de nouveaux faits, et recommence jusqu'à saturation. C'est la stratégie qu'utilise le moteur d'inférence ontologique de VVL (`ontology:infer`, voir chapitre 12). Le **chaînage arrière** (backward chaining) part de la requête, identifie les règles dont la conclusion correspond, et réduit récursivement le problème à ses sous-buts jusqu'à atteindre des faits. C'est la stratégie de Prolog par défaut, utilisée par la commande `query` de VVL (voir chapitre 11).

Voici un exemple Prolog illustrant les deux stratégies sur un système de classification médicale élémentaire :

```prolog
% Faits observés
symptome(patient1, fievre).
symptome(patient1, toux).
symptome(patient1, fatigue).

% Regles diagnostiques
syndrome_grippal(P) :- symptome(P, fievre), symptome(P, toux), symptome(P, fatigue).
prescription(P, repos) :- syndrome_grippal(P).
prescription(P, paracetamol) :- syndrome_grippal(P), symptome(P, fievre).

% Chainage arriere (Prolog defaut) :
% ?- prescription(patient1, X).
% X = repos ;
% X = paracetamol.

% Chainage avant (saturation) :
% Iter 1 : decouvre syndrome_grippal(patient1)
% Iter 2 : decouvre prescription(patient1, repos)
% Iter 3 : decouvre prescription(patient1, paracetamol)
% Iter 4 : aucun nouveau fait, point fixe atteint
```

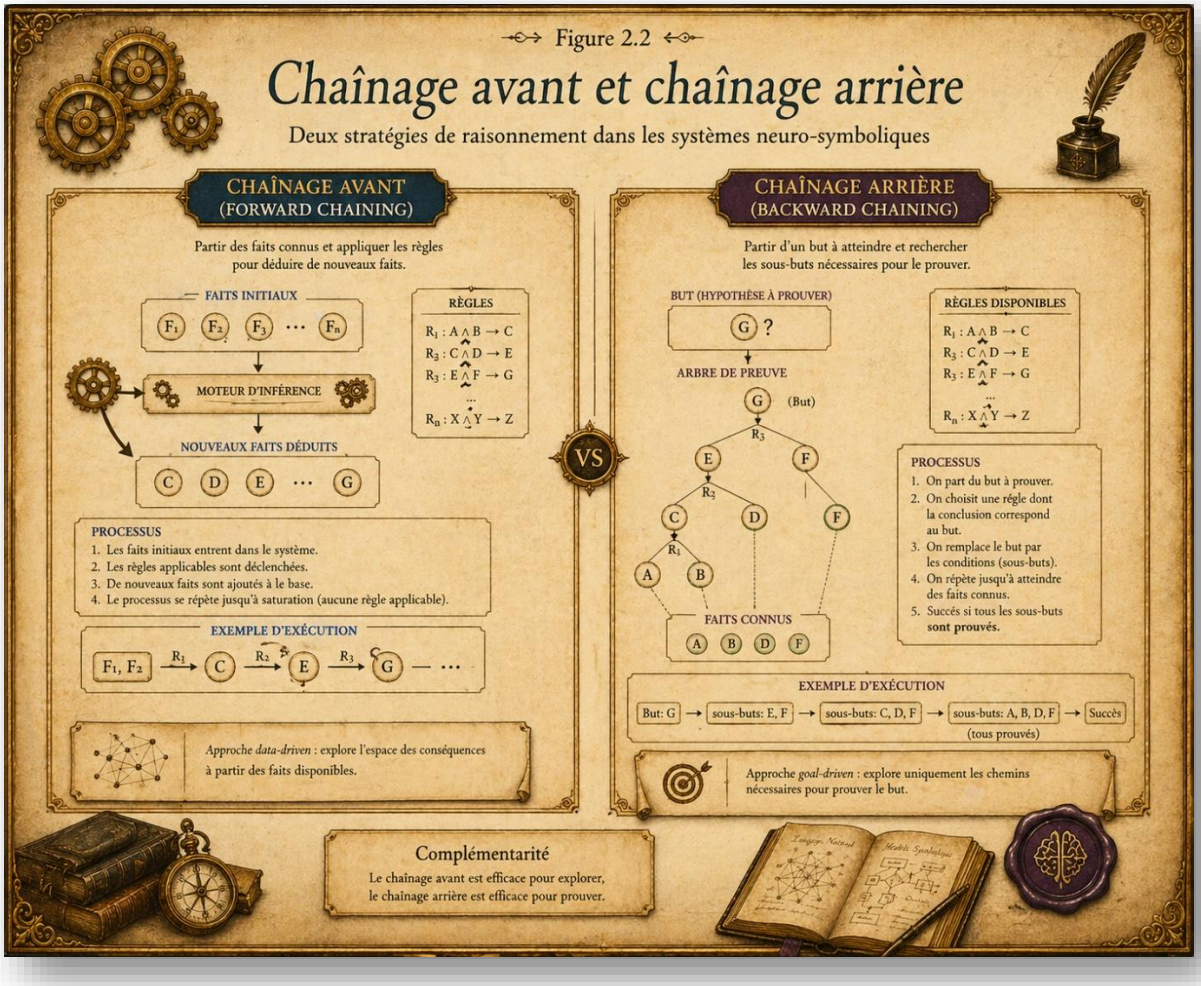


Figure 2.2 — *Les deux stratégies opérationnelles de la déduction logique*

La garantie de la déduction est sa principale force : si les prémisses sont vraies et les règles correctes, la conclusion est nécessairement vraie. Mais c'est aussi sa principale faiblesse pratique : elle suppose que toutes les prémisses pertinentes ont été explicitement formulées, et que toutes les règles applicables ont été codées. Or dans la plupart des domaines réels, ni l'une ni l'autre de ces hypothèses ne tient. C'est ce constat qui ouvre la place à l'induction.

2.2. Raisonnement inductif : de l'exemple à la règle

L'**induction** est le mouvement inverse de la déduction : à partir d'observations particulières, on infère une règle générale. *Le soleil s'est levé tous les jours observés, donc le soleil se lève tous les jours.* Aucune garantie logique n'accompagne cette conclusion — c'est tout l'enjeu du **problème de l'induction** énoncé par David Hume au XVIII^{e} siècle : aucune accumulation finie d'observations ne peut établir une loi universelle. L'induction produit des règles plausibles, jamais des règles certaines.

Cette faiblesse logique apparente est compensée par une force pratique considérable : l'induction est **le seul mode de raisonnement qui crée de nouvelles règles**. Sans induction, le savoir d'un système se réduit à ce qui y a été codé explicitement. C'est précisément ce qui plafonne les systèmes experts classiques et ce que le machine learning, héritier moderne de l'induction, vient débloquer.

En programmation logique, l'**Inductive Logic Programming** (ILP), théorisée par Stephen Muggleton dans les années 1990, formalise ce processus : à partir d'exemples positifs E^+, d'exemples négatifs E^- et d'une connaissance de fond B, l'algorithme cherche une hypothèse H (ensemble de clauses Prolog) telle que $B \wedge H \vdash E^+$ et $B \wedge H \nvdash E^-$. Les systèmes Metagol et Aleph sont les deux outils de référence du domaine.

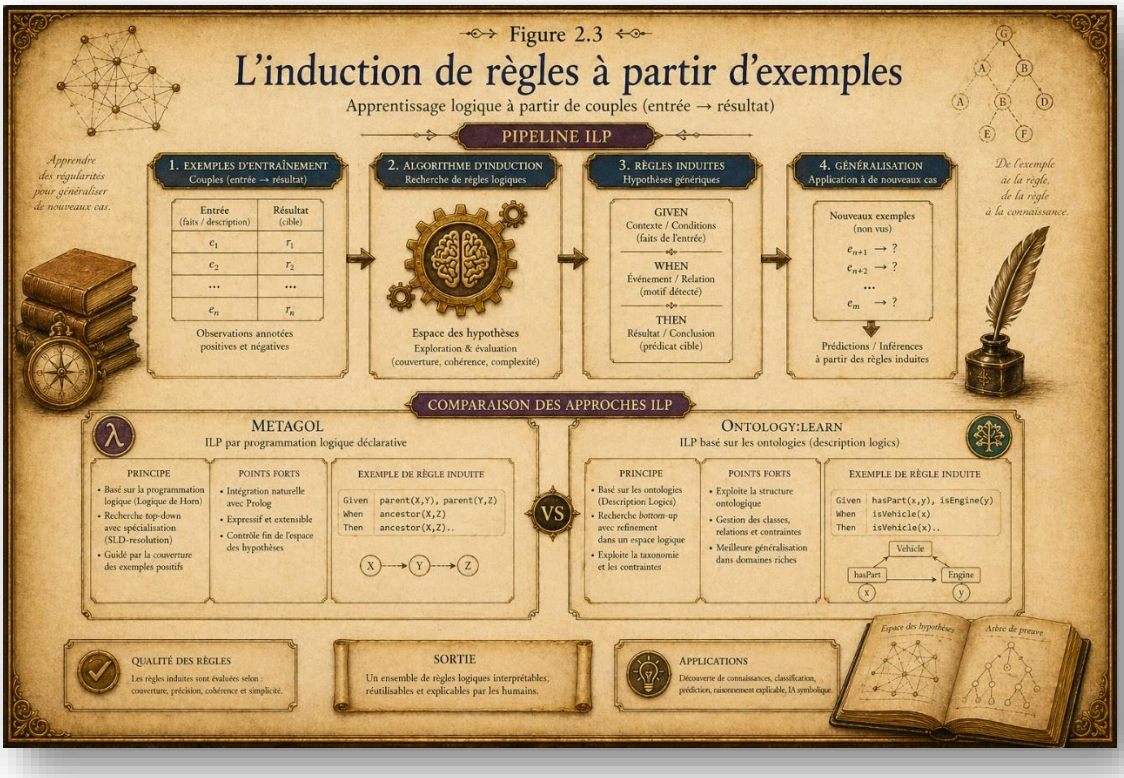


Figure 2.3 — *Pipeline d'induction de règles : de l'exemple au code exécutable*

VVL implémente l'induction via la commande `ontology:learn` (chapitre 17). À la différence des outils ILP académiques, qui explorent combinatoirement l'espace des clauses possibles, VVL délègue cette recherche au LLM : on lui présente les couples (entrée → résultat), on lui demande de produire les règles Given/When/Then qui les expliquent. Le résultat est en Gherkin — donc lisible par les experts métier, pas seulement par les développeurs — et il est ensuite compilé en code VVL exécutable. C'est un détournement créatif du LLM : on n'utilise pas sa capacité à *répondre* à une question, mais sa capacité à *généraliser* à partir d'exemples.

Cette stratégie a une conséquence importante. Là où l'ILP classique est souvent limitée à des domaines simples par l'explosion combinatoire, le LLM peut traiter des règles métier complexes impliquant des concepts flous, des seuils numériques, des conditions composites — au prix d'une vérification a posteriori que les règles produites couvrent effectivement les exemples. C'est l'asymétrie typique du neuro-symbolique : la recherche est rapide et créative côté LLM, la vérification est lente et fiable côté symbolique.

2.3. Raisonnement abductif : la meilleure explication d'une observation

L'**abduction**, théorisée par Charles Sanders Peirce à la fin du XIX^{e} siècle, est le troisième mode de raisonnement, souvent négligé alors qu'il est probablement le plus fréquent dans la pratique scientifique et médicale. L'abduction part d'une observation surprenante et d'un corpus de règles connues, et cherche **la meilleure explication** — l'hypothèse qui, si elle était vraie, rendrait l'observation attendue.

Le schéma formel est : étant donnée une observation `O` et une théorie `T` telle que `T ∪ {H} ⊢ O`, trouver l'hypothèse `H` la plus plausible. Cette plausibilité ne se déduit pas de la logique seule : elle dépend de critères extra-logiques (simplicité, parcimonie, cohérence avec d'autres connaissances) que Peirce a regroupés sous le nom de *retroduction*.

Le diagnostic médical en est l'illustration canonique. Un patient présente fièvre, toux et fatigue. Plusieurs hypothèses pourraient expliquer cet ensemble : grippe, pneumonie, COVID, mononucléose. Le médecin ne *déduit* pas le diagnostic — aucune règle ne dit *si fièvre + toux + fatigue alors grippe* avec certitude. Il *abduit* la meilleure explication, en pondérant chaque hypothèse par sa prévalence, sa cohérence avec les autres signes, et la facilité avec laquelle elle expliquerait l'ensemble.

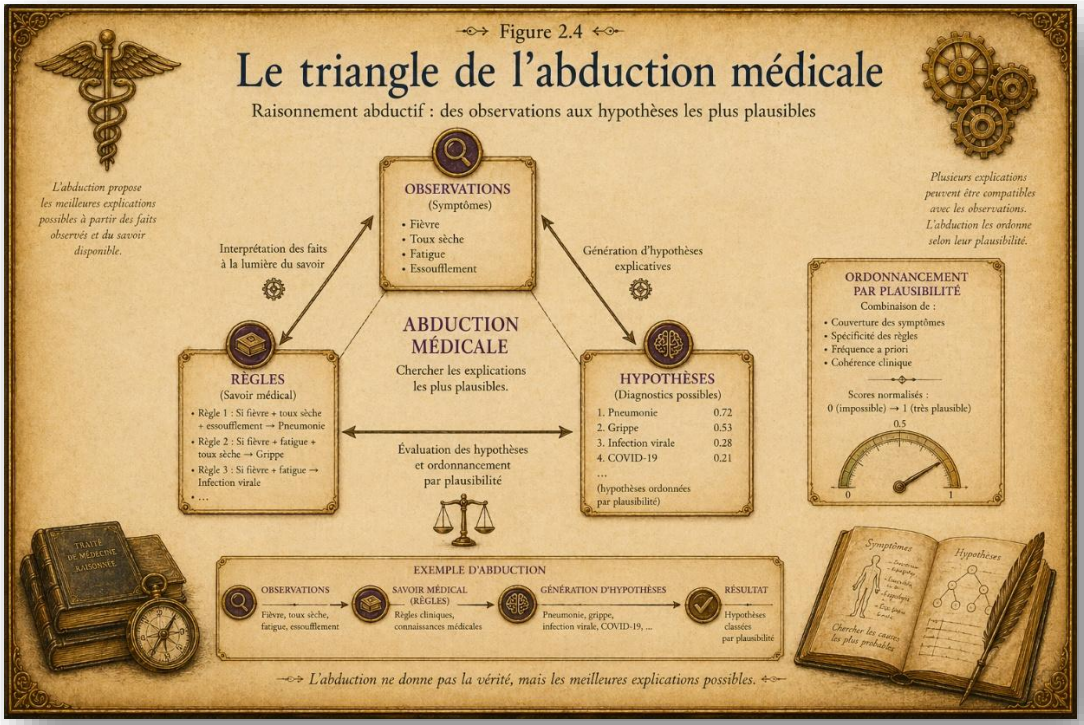


Figure 2.4 — *L'abduction comme classement d'hypothèses par plausibilité*

L'abduction est précisément ce que VVL implémente avec les opérateurs Zalta du chapitre 13. La commande `~~` calcule le score de satisfaction d'une instance contre un archétype : plus le score est élevé, plus l'instance ressemble à l'archétype. Comparer une instance contre plusieurs archétypes produit un classement — c'est-à-dire l'ordre des hypothèses abductives. La fonction `ontology:identify` formalise ce processus pour la reconnaissance de concepts ; le tutoriel sur le diagnostic médical du chapitre 19 le déroule sur un cas complet.

Le tableau ci-dessous résume les trois modes :

Mode	Direction	Garantie	Implémentation VVL
Déduction	Règle + cas → résultat	Vrai si prémisses vraies	ontology:infer, query
Induction	Cas + résultats → règle	Plausible, jamais certain	ontology:learn
Abduction	Cas + règles → hypothèse	Meilleure explication	~~ (satisfies hybride)

Figure 2.5 — *Synthèse des trois modes de raisonnement et leur correspondance VVL*

2.4. La théorie des objets abstraits d'Edward Zalta

Le philosophe **Edward Zalta**, à Stanford, a développé depuis les années 1980 une métaphysique formelle baptisée **théorie des objets abstraits** (*Abstract Object Theory*, AOT). Sans entrer dans la logique modale d'ordre supérieur qui en constitue la base technique, l'idée centrale est d'une simplicité élégante : il existe deux types d'objets, et deux manières dont une propriété peut leur être attachée.

Les **objets ordinaires** sont les objets concrets du monde — Socrate, le Soleil, ce stylo posé sur la table. Ils *exemplifient* leurs propriétés : Socrate exemplifie la propriété d'être mortel, le Soleil exemplifie la

propriété d'avoir une masse de 1,989 × 10^{30} kg. Une propriété exemplifiée est une propriété factuellement possédée, mesurable, vérifiable.

Les **objets abstraits** sont des entités conceptuelles — l'archétype de l'étoile, l'idéal du candidat-développeur senior, le concept de matériau résistant. Ils *encodent* des propriétés sans nécessairement les exemplifier. L'archétype de l'étoile encode la propriété *avoir une masse importante*, *émettre de la lumière par fusion thermonucléaire*, *être sphérique* — mais il ne possède aucune masse réelle, aucune lumière émise. Il est la forme conceptuelle, pas la chose concrète.

Le **principe de compréhension** de Zalta affirme que pour toute condition `ϕ` exprimable dans le langage de la théorie, il existe un unique objet abstrait qui encode exactement les propriétés satisfaisant `ϕ`. Formellement : `∃!x (A!x ∧ ∀F (xF ≡ ϕ))` — il existe un objet abstrait `x` tel que pour toute propriété `F`, `x` encode `F` si et seulement si `F` satisfait `ϕ`.

Cette construction donne un fondement métaphysique précis à la notion d'archétype, qui est exactement ce que VVL réifie via l'opérateur `[@ "..."]` (chapitre 9). Quand on écrit `[@ "Soleil"]`, on demande au LLM de produire un objet qui exemplifie les propriétés du Soleil réel. Quand on écrit `[ontology:abstract MateriauResistant ...]`, on définit un objet abstrait qui encode les propriétés idéales du concept *matériau résistant*. La distinction technique se traduit par deux prédicats Prolog distincts : `prop/3` pour l'exemplification, `enc/3` pour l'encodage.

2.5. Double prédication : exemplification (prop/3) et encodage (enc/3)

La conséquence opérationnelle de la théorie de Zalta dans VVL est la **double prédication** : chaque propriété attachée à un objet est typée selon qu'elle est exemplifiée ou encodée. Cette typage n'est pas une lubie philosophique : il résout un problème concret que rencontrent toutes les ontologies classiques.

Considérons l'archétype *développeur senior*. Si on lui attribue `experience >= 5 ans`, on entend une condition idéale, pas une mesure factuelle (l'archétype n'a pas d'expérience réelle, c'est une forme conceptuelle). Maintenant, considérons l'instance *Alice, 9 ans d'expérience*. Si on attache `experience = 9` à Alice, c'est une mesure factuelle. Les ontologies classiques ne distinguent pas ces deux usages : elles utilisent le même prédicat `has(alice, experience, 9)` et `has(senior, experience, 5)`. Il en résulte des incohérences subtiles : peut-on dire que l'archétype *est* expérimenté, ou seulement qu'il *requiert* de l'expérience ?

Zalta tranche : l'archétype `senior` *encode* `experience` (avec la valeur attendue 5), Alice *exemplifie* `experience` (avec la valeur réelle 9). En Prolog VVL, cela donne :

```
% Encodages de l'archetype 'senior' (proprietes ideales)
enc(senior, experience, 5).
enc(senior, autonomie, eleve).
enc(senior, leadership, oui).

% Exemplifications de l'instance Alice (proprietes factuelles)
prop(alice, experience, 9).
prop(alice, autonomie, eleve).
prop(alice, leadership, oui).
prop(alice, salaire, 65).

% Satisfaction : ratio des proprietes encodees satisfaites par l'instance
satisfies(I, A, Score) :-
    findall(F-V, enc(A, F, V), Encs),
    length(Encs, Total),
    findall(F-V, (member(F-V, Encs), prop(I, F, V)), Sats),
    length(Sats, NSat),
    Score is NSat / Total.

% ?- satisfies(alice, senior, S).
% S = 1.0  (3/3 proprietes encodees satisfaites)
```

La beauté de cette structure est que la satisfaction se calcule sans aucun appel LLM : c'est une opération purement symbolique sur les deux ensembles de prédicats. C'est exactement ce que fait l'opérateur `==` symbolique de VVL (chapitre 13). Lorsque les noms de propriétés ne correspondent pas exactement (par exemple `xp` vs `experience`), l'opérateur hybride `~~` envoie les paires non résolues au LLM en un seul appel batché — c'est l'asymétrie neuro-symbolique appliquée à la satisfaction d'archétype.

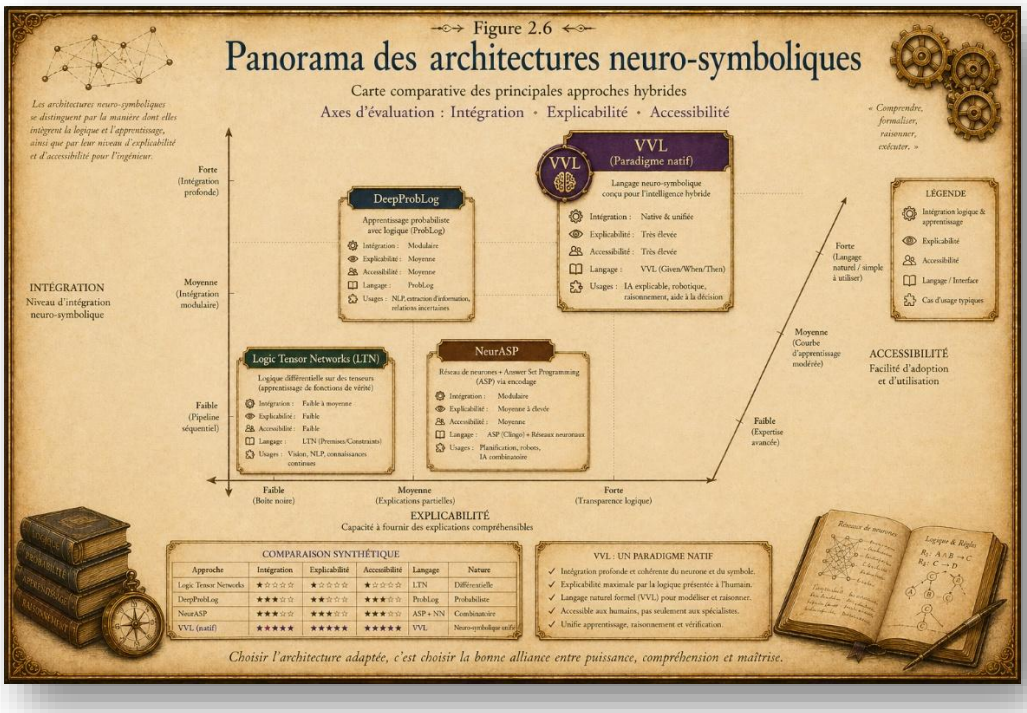


Figure 2.6 — *La double prédication : prop/3 pour l'exemplification, enc/3 pour l'encodage*

2.6. Panorama des architectures hybrides existantes

VVL n'est pas le premier système à proposer une intégration neuro-symbolique. Plusieurs architectures académiques l'ont précédé, chacune avec ses choix de conception propres et ses limites. Ce panorama n'est pas exhaustif — la communauté neuro-symbolique compte aujourd'hui des dizaines d'approches — mais il identifie les trois familles principales et la position que VVL occupe par rapport à elles.

DeepProbLog : Prolog différenciable

Proposé par Manhaeve, Dumančić, Kimmig, Demeester et De Raedt (2018), **DeepProbLog** étend ProbLog (Prolog probabiliste) en autorisant des prédicats dont la valeur de vérité est calculée par un réseau de neurones. Un prédicat `digit(image, n)` peut ainsi être implémenté par un classifieur convolutif, et utilisé dans une règle Prolog : `sum(I1, I2, S) :- digit(I1, A), digit(I2, B), S is A + B`. L'apprentissage des poids du réseau et l'inférence Prolog sont intégrés dans un même processus de propagation de gradient.

La force de DeepProbLog est l'unification mathématique des deux paradigmes via la différentiabilité. Sa limite est pratique : la mise en œuvre exige une expertise simultanée en programmation logique et en deep learning, et le passage à l'échelle reste un défi de recherche.

NeurASP : Answer Set Programming et perception neuronale

NeurASP (Yang, Ishay et Lee, 2020) suit une stratégie comparable mais ancrée dans l'Answer Set Programming (ASP), un paradigme de programmation logique non-monotone bien adapté au raisonnement par défaut et aux exceptions. Les prédicats neuronaux jouent le rôle de fonctions d'évidence dont les sorties alimentent le solveur ASP. NeurASP excelle sur les problèmes de raisonnement combinatoire avec perception (Sudoku visuel, énigmes spatiales).

Logic Tensor Networks : la logique floue dans les réseaux

Logic Tensor Networks (Serafini et d'Avila Garcez, 2016) fait le chemin inverse : au lieu de doter les réseaux d'une couche logique, ils encodent les formules logiques comme contraintes différentiables sur l'apprentissage du réseau. Une règle `∀x (chien(x) → mammifere(x))` devient une perte additionnelle qui pénalise les configurations de poids violant la règle. C'est une approche purement neuronale qui *apprend à respecter* la logique, sans la résoudre symboliquement.

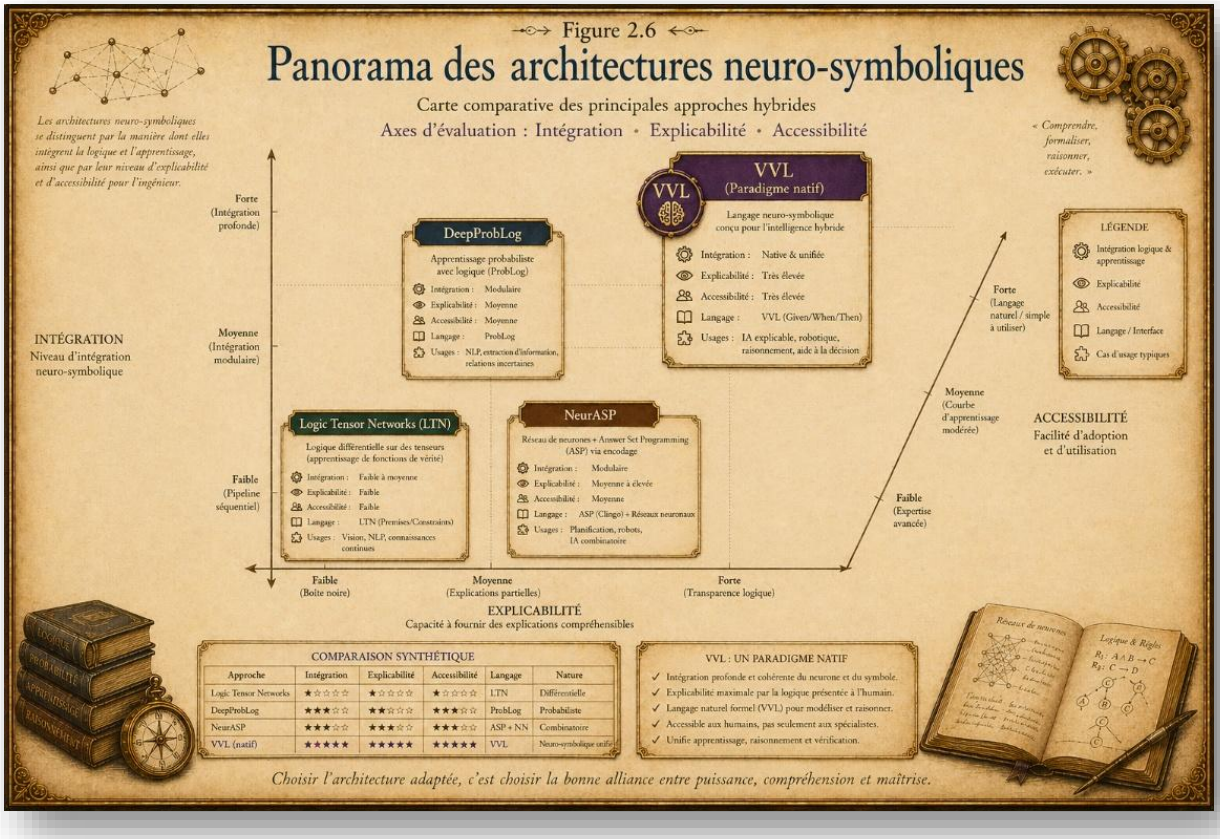


Figure 2.7 — *Position de VVL parmi les architectures neuro-symboliques contemporaines*

La position de VVL

Les trois approches précédentes ont en commun d'être des **frameworks de recherche** : elles requièrent l'assemblage manuel de plusieurs technologies, une double expertise en logique et en apprentissage, et leur déploiement en production reste exceptionnel. VVL fait un autre pari : transformer le neuro-symbolique en **paradigme natif d'un langage de programmation**, accessible à un développeur d'application sans qu'il ait à maîtriser la théorie des graphes computationnels ni les algorithmes ILP.

Concrètement, cela signifie trois choses. D'abord, le LLM n'est pas appelé via une bibliothèque externe : c'est un opérateur du langage, au même titre que + ou ==. Ensuite, la couche symbolique (Prolog, ontologie, règles Gherkin) est intégrée au runtime, accessible par les mêmes commandes que le reste du langage. Enfin, le pont entre les deux — la théorie des objets abstraits de Zalta — est implémenté comme un système d'opérateurs (==, ~~, *, **, +, ++, /, //) qui dispatche automatiquement entre symbolique et hybride selon le contexte.

Le tableau suivant compare les choix de conception :

Critère	DeepProbLog	NeurASP	LTN	VVL
Logique sous-jacente	ProbLog	ASP	Floue	Prolog + Zalta
Couche neuronale	Réseaux dédiés	Réseaux dédiés	Réseau global	LLM intégré
Apprentissage	Backpropagation	Backpropagation	Backpropagation	Inductif via LLM
Statut	Recherche	Recherche	Recherche	Langage applicatif
Public visé	Chercheurs ML+logique	Chercheurs ML+logique	Chercheurs ML+logique	Développeurs métier

Figure 2.8 — *Comparaison des choix de conception entre architectures neuro-symboliques*

Avec ces fondements théoriques en place — trois modes de raisonnement, double prédication zaltaienne, panorama des approches existantes — nous pouvons aborder dans le chapitre 3 la vision architecturale concrète de VVL : comment ces idées se cristallisent dans un langage homoiconique unique, et pourquoi cette cristallisation produit un saut qualitatif par rapport aux frameworks d'assemblage existants.

3. VAILS et VVL : un langage conçu pour le neuro-symbolique

Mots-clés : VAILS, VVL, homoiconicité, runtime unifié, opérateur sémantique, sous-systèmes, LangChain, LlamaIndex, langage de programmation IA

Les deux chapitres précédents ont établi le contexte historique et théorique. Celui-ci présente la réponse architecturale concrète apportée par VAILS et son langage VVL. La thèse défendue ici est simple mais radicale : la programmation neuro-symbolique ne peut atteindre sa maturité industrielle qu'en cessant d'être un schéma d'intégration de bibliothèques, pour devenir le paradigme natif d'un langage de programmation. Toutes les bibliothèques d'IA hybride existantes — DeepProbLog, NeurASP, LangChain, LlamaIndex — partagent un défaut commun : elles assemblent des outils initialement conçus séparément. VVL fait l'inverse : il conçoit le langage *autour* du paradigme.

Ce chapitre déroule cette idée en cinq mouvements. La section 3.1 expose la vision du runtime unifié et ce qu'elle change pour le développeur. La section 3.2 montre comment l'homoiconicité, héritée de Lisp, fournit le substrat technique qui rend cette unification possible. La section 3.3 explique pourquoi le LLM, intégré comme opérateur du langage, devient un co-processeur fiable plutôt qu'un service externe imprévisible. La section 3.4 présente l'architecture modulaire VAILS en sept sous-systèmes. La section 3.5 conclut par une comparaison frontale avec les frameworks d'assemblage les plus diffusés.

3.1. La vision : un seul runtime, un seul espace de représentation

Pour comprendre ce que VVL apporte, il faut d'abord mesurer ce qu'il évite. Considérons un projet typique d'application IA en 2026 : extraction de devis depuis des PDF, classification par règles métier, recherche sémantique dans un catalogue, et génération de réponses contextuelles. La pile technologique habituelle empile au minimum sept composants : un LLM via API (OpenAI, Anthropic), un client de vectorisation (OpenAI Embeddings), une base vectorielle (Pinecone, Weaviate, Chroma), un parseur de documents (PyPDF, Unstructured), un moteur de règles (Drools, expert-py), un ORM pour la persistance (SQLAlchemy, Prisma), et le code glue Python qui connecte le tout.

Chaque composant a son propre format de données, son propre langage de configuration, son propre cycle de versions. Le développeur passe la majorité de son temps à écrire des adaptateurs — convertir le JSON renvoyé par l'API LLM en objets Python, sérialiser ces objets pour la base vectorielle, désérialiser les résultats de recherche pour les passer au moteur de règles, formater les sorties du moteur pour les réinjecter dans un nouveau prompt. Ce code glue représente couramment 60 à 80 % du codebase, sans apporter aucune valeur métier.

VVL propose à la place un runtime unique. Le LLM est un opérateur du langage. La base vectorielle est un builtin (`ragInsert`, `ragAsk`). L'ontologie est un système de classes natif (`ontology:define`). Le moteur de règles est intégré à l'ontologie (`ontology:rule`, `ontology:infer`). La persistance Prolog (`kb`, `obj-assert`) partage le même runtime. Le résultat : le programme exprime directement la logique métier, sans aucune couche d'adaptation.

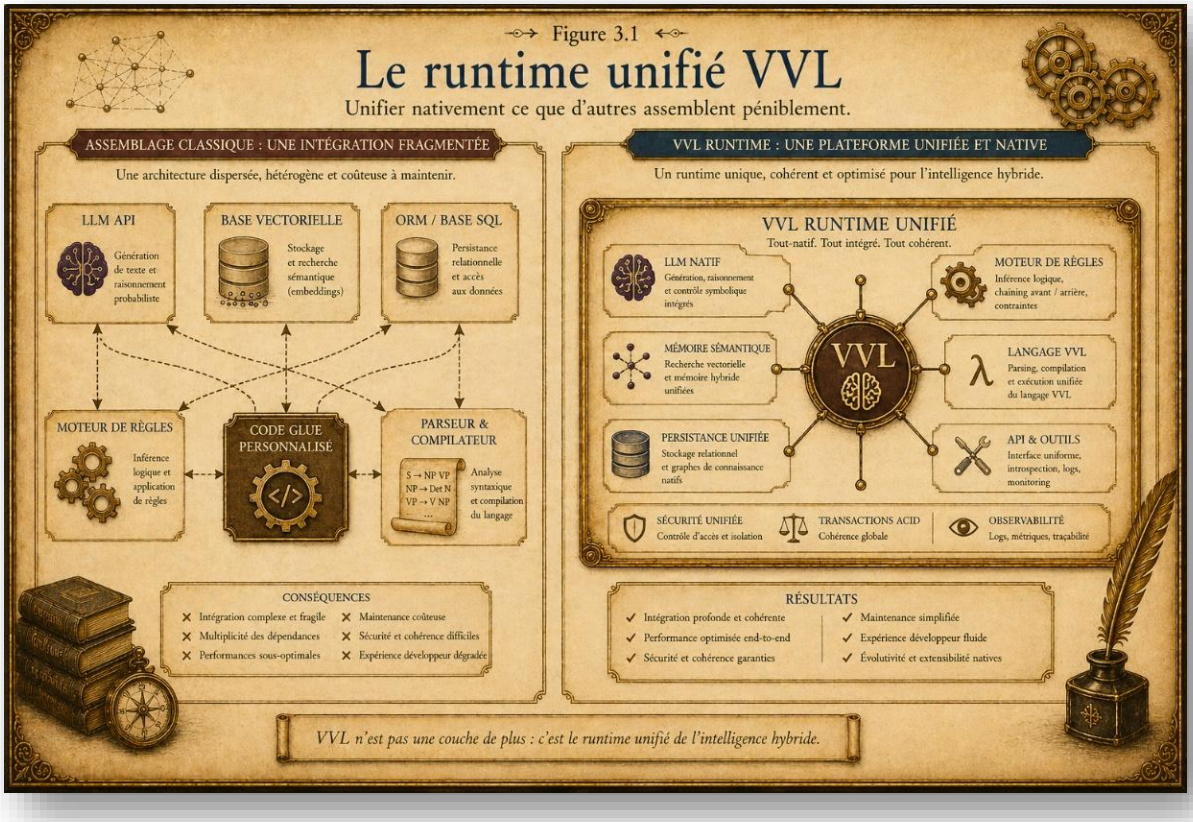


Figure 3.1 — *Le runtime unifié VVL face à l'approche par assemblage*

Cette unification a trois conséquences pratiques que les chapitres suivants exploiteront constamment. D'abord, **les types circulent sans conversion** : un objet ontologique extrait par LLM est directement projetable dans la base Prolog (`obj-assert`), et un fait Prolog inféré est directement convertible en objet VVL (`ontology:toVVL`). Ensuite, **le cache et le retry sont automatiques** au niveau du langage, pas de chaque appel : VVL maintient un cache sémantique persistant qui transforme chaque concept exploré par le LLM en investissement définitif. Enfin, **le débogage est unifié** : on inspecte le runtime VVL au lieu de tracer sept logs hétérogènes répartis dans cinq bibliothèques.

3.2. L'homoiconicité : code, données et langage naturel partagent la même syntaxe

L'unification du runtime n'est rendue possible que par une propriété fondamentale héritée de Lisp (1958) : l'**homoiconicité**. Un langage homoiconique est un langage dont la structure des programmes est identique à la structure de ses données. En Lisp, *tout est liste*. En VVL, *tout est commande entre crochets* :

```vvl
# 1. Du code VVL est une liste de listes
[log [length "hello"]]                    # => 5

# 2. Cette meme structure peut etre construite comme donnee...
$code = "[log [length \"hello\"]]"

# 3. ...puis evaluee comme code
[evaluate $code]                          # => 5

# 4. Du code peut etre genere par le LLM...
$generated = ($data @ [§ "filtrer par age > 30"])

# 5. ...et execute directement, sans compilation, sans sandbox
[filter $generated $personnes]
```

Cette propriété, qui semble n'être qu'une élégance syntaxique, a des conséquences architecturales profondes. Elle abolit la frontière entre trois ordres habituellement distincts :

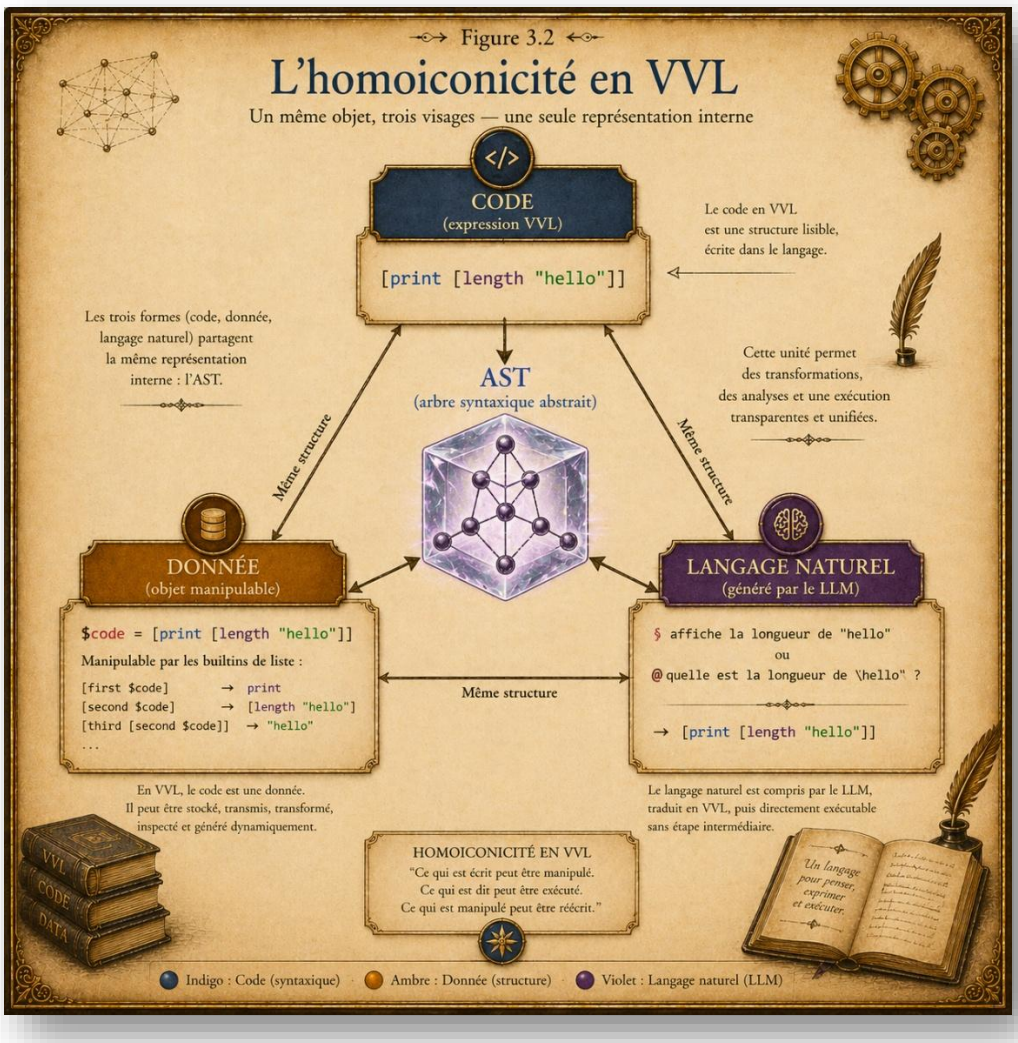


Figure 3.2 — *Code, donnée et langage naturel : trois faces du même AST*

Le code et les données. Là où la plupart des langages distinguent un programme (compilé ou interprété par un parseur dédié) de ses données (manipulées par des structures différentes), VVL utilise la même structure de liste pour les deux. Une fonction peut donc inspecter un autre programme, le réécrire, le composer avec un troisième — c'est le fondement de la **méta-programmation** native. La commande `keyword` qui définit les macros utilisateurs s'appuie directement sur cette identité.

Le programme et le langage naturel. C'est la conséquence la plus radicale dans le contexte du LLM. Une chaîne de caractères produite par GPT — par exemple en réponse à `[§ "écrire une lambda qui filtre par âge"]` — est syntaxiquement une donnée. Mais comme cette donnée a la forme d'une liste VVL valide, elle peut être passée à `evaluate` et exécutée immédiatement. **Aucun pipeline de compilation, aucun sandbox, aucun parseur intermédiaire n'est nécessaire.** Le langage naturel et le code partagent le même espace de représentation.

Le programme et son auto-modification. Un programme VVL peut générer un autre programme, l'évaluer, et incorporer le résultat dans son propre flux d'exécution. C'est ce qu'utilise l'agent autonome `[architect]` (chapitre 22) : il prend un objectif en langage naturel, génère un programme VVL complet, l'exécute, vérifie le résultat, et — si nécessaire — régénère un programme corrigé. Ce cycle créer-évaluer-corriger n'est pas un pattern architectural construit au-dessus du langage : c'est une simple application des primitives `[§ ...]`, `[@ ...]` et `evaluate`.

Sans homoiconicité, la programmation neuro-symbolique resterait nécessairement un pipeline d'intégration : un LLM externe produit du texte, un parseur dédié le valide, un compilateur le transforme en code exécutable, puis un sandbox l'exécute. Avec l'homoiconicité, ces quatre étapes s'effondrent en une seule opération du langage.

3.3. Le LLM comme citoyen de première classe du langage

La deuxième innovation architecturale de VVL est l'intégration du LLM **non comme bibliothèque appelée par le programme, mais comme opérateur du langage lui-même**. Concrètement, lorsqu'un opérande d'une expression a le type `VVL_Semantic` (créé par la syntaxe `[§ "..."]`), les opérateurs arithmétiques, logiques et de comparaison changent de sémantique pour devenir des opérations conceptuelles pilotées par le LLM.

Le contraste avec une bibliothèque comme LangChain est saisissant. Pour comparer deux concepts en LangChain, le développeur écrit explicitement un appel à un chain de comparaison sémantique, configure un prompt, parse la réponse, gère les erreurs, et reçoit un résultat textuel qu'il doit lui-même convertir en booléen. En VVL, la même opération s'écrit `[§ "chat"] == [§ "félin"]` et retourne directement `1`. La différence n'est pas seulement de concision : elle est de **statut épistémique**. Le LLM n'est plus un service externe avec ses idiosyncrasies, c'est un opérateur du langage avec son contrat d'interface.

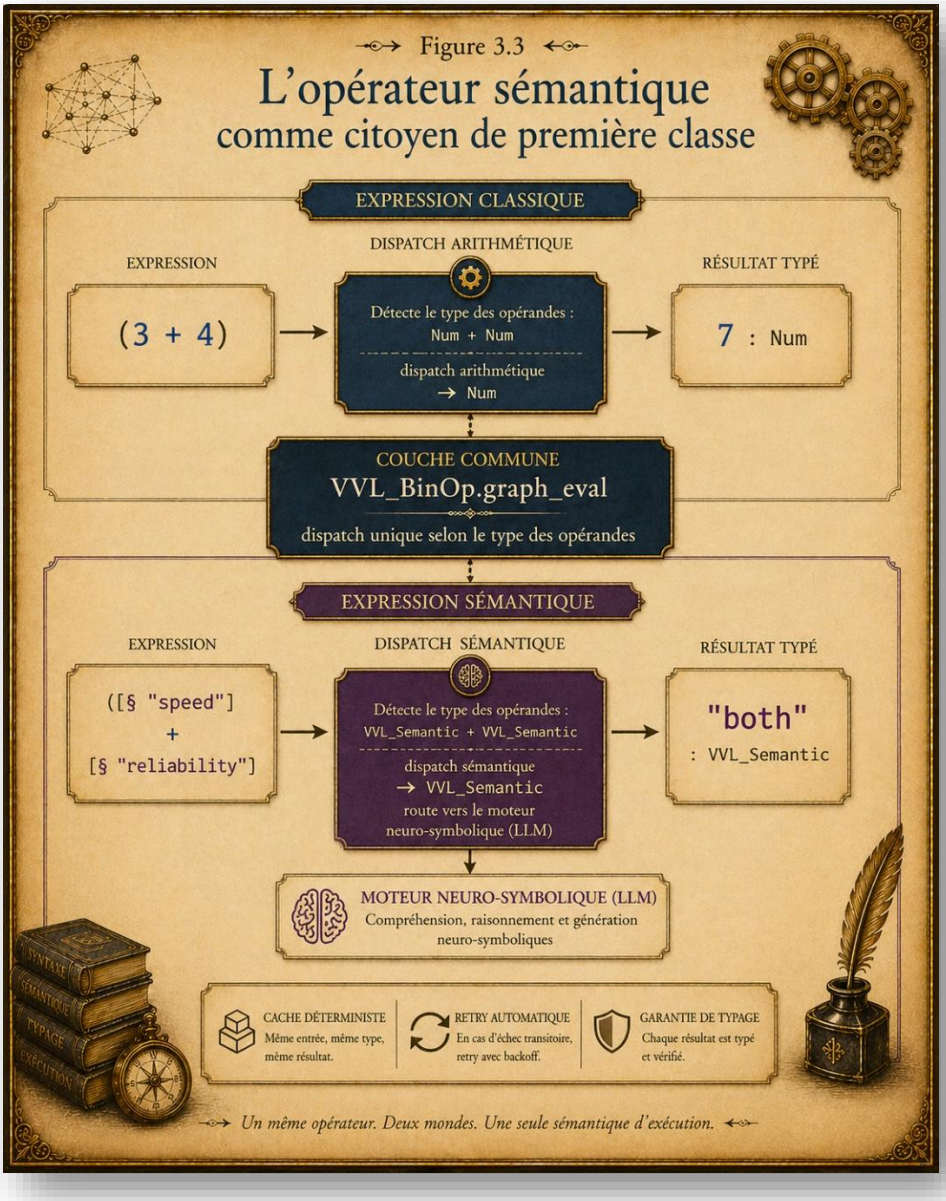


Figure 3.3 — *Le dispatcher VVL_BinOp décide au runtime entre arithmétique et opération sémantique*

Cette intégration repose sur quatre garanties que les bibliothèques externes n'offrent jamais à ce niveau de profondeur. **Premièrement, la garantie de typage** : chaque opérateur sémantique a un type de sortie spécifié (`BOOLEAN`, `STRING`, `SEMANTIC`, `LAMBDA`). Le résultat est immédiatement utilisable dans un `if`, un `filter`, ou une autre expression VVL, sans parsing manuel. **Deuxièmement, le cache déterministe** : chaque appel sémantique est mis en cache par hash de l'expression complète, mais aussi de l'empreinte du prompt effectif de l'opérateur et du modèle qui a répondu ; tant que ces trois éléments sont stables, un même appel produit le même résultat dans toutes les sessions, et toute édition de la spécification ou tout changement de modèle invalide l'entrée au lieu de la laisser survivre. C'est cette clé à trois composantes qui transforme un système probabiliste en composant reproductible et auditable.

Troisièmement, le retry automatique avec validation syntaxique : pour les opérateurs qui produisent du code (notamment `@`), VVL valide que la sortie du LLM est une lambda VVL

syntaxiquement correcte ; si elle ne l'est pas, il relance avec un message d'erreur, jusqu'à un nombre configurable de tentatives. **Quatrièmement, les spécifications XML avec few-shot** : chaque opérateur sémantique est défini par une spécification XML qui inclut sa description, son type de sortie, et trois à cinq exemples canoniques. Ces exemples sont injectés dans le prompt système à chaque appel, garantissant la stabilité du comportement même quand le modèle sous-jacent change.

Le résultat de cette architecture est que **le LLM cesse d'être un système probabiliste imprévisible pour devenir un opérateur aussi fiable que `+` ou `==`**. C'est exactement ce qui rend possible l'écriture de programmes neuro-symboliques en cinq lignes au lieu de cinq cents (chapitre 18).

3.4. L'architecture VAILS en sept sous-systèmes

VAILS — *VAILS Augmented Intelligence Language System* — n'est pas un serveur monolithique mais une architecture modulaire composée de sept sous-systèmes spécialisés. Chacun est fondé sur un paradigme informatique distinct, et leur intégration via le runtime VVL est ce qui produit la richesse fonctionnelle du système. Cette section les présente brièvement ; chacun fera l'objet de chapitres ultérieurs détaillés.

Sous-système	Paradigme théorique	Rôle	Chapitre
VAILS.core	Modélisation orientée graphe, langage homoiconique	Moteur d'exécution, interpréteur VVL	4–6
VAILS.distributed	Calcul d'orchestration (ORC), concurrence par sites	Parallélisme, async, combinateurs	5
VAILS.knowledge	IA symbolique, ontologies, raisonnement Prolog	Base de connaissances, inférence	11–13
VAILS.compute	Programmation stack-array (Uiua), RPN	Calcul vectoriel haute performance	—
VAILS.agents	Délégation de tâches, files de messages	Agents distants, communication	22
VAILS.vapp	VDOM, SOUL AI, interfaces réactives	Applications interactives IA	—
VAILS.voice	WebSocket temps réel, Realtime API	Interaction vocale bidirectionnelle	—

Figure 3.4 — *Les sept sous-systèmes de l'architecture VAILS et leurs paradigmes*

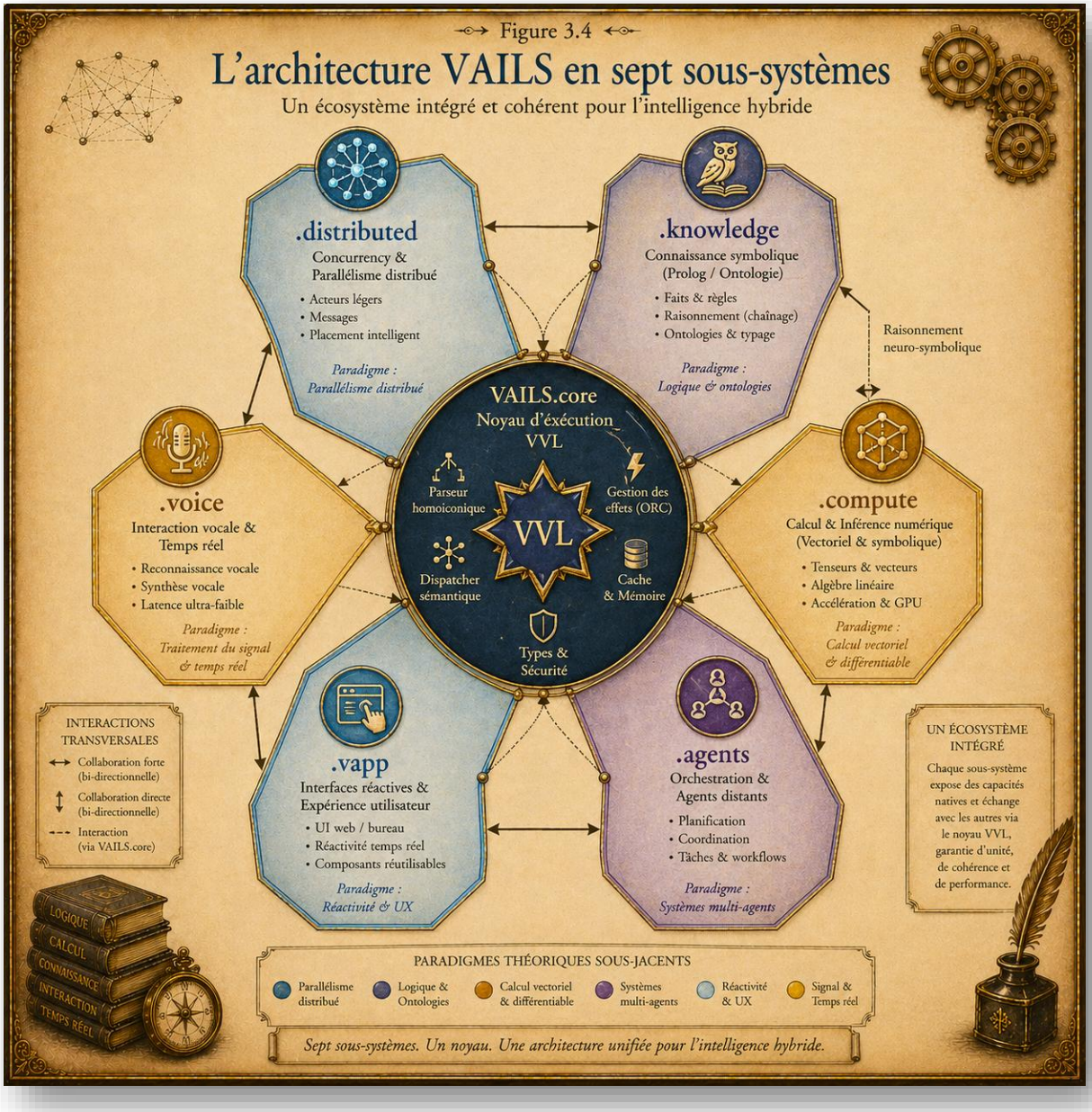


Figure 3.5 — *Cartographie des sous-systèmes VAILS autour du noyau VVL*

VAILS.core est le noyau : interpréteur VVL, gestion des contextes, dispatcher des opérateurs, runtime d'exécution. C'est lui qui implémente l'homoiconicité et qui orchestre l'invocation des autres sous-systèmes. Les chapitres 4 à 6 le couvrent en détail (syntaxe, fonctionnel, OO).

VAILS.knowledge héberge le système symbolique : moteur Prolog intégré, système ontologique avec classes et règles d'inférence, théorie des objets abstraits de Zalta. C'est le pilier symbolique de la programmation neuro-symbolique en VVL ; les chapitres 11 à 13 le détaillent.

VAILS.distributed fournit la couche d'orchestration concurrente : combinateurs ORC (`|`, `>>`, `~>`) inspirés du calcul d'orchestration de Misra et Cook, exécution asynchrone via `async`/`await`, fibers pour les générateurs coroutinés. Cette couche est cruciale pour les pipelines neuro-symboliques en production, où l'on exécute des dizaines d'appels LLM en parallèle (chapitre 5).

VAILS.compute intègre un moteur de calcul vectoriel à pile inspiré du langage Uiua, avec accélération NumPy. Il est utilisé pour les opérations mathématiques intensives (statistiques, transformations matricielles) et pour le post-traitement des résultats RAG.

VAILS.agents implémente la délégation de tâches entre agents distants. Couplé aux opérateurs `[agent]` et `[architect]` du chapitre 22, il permet de construire des systèmes multi-agents collaboratifs.

VAILS.vapp et **VAILS.voice** sont les couches de présentation et d'interaction : VDOM pour les interfaces web réactives, WebSocket et OpenAI Realtime API pour la voix bidirectionnelle. Elles dépassent le périmètre de cet ouvrage centré sur le neuro-symbolique, mais elles montrent que VAILS est un système intégré jusqu'à la couche utilisateur final.

3.5. Comparaison avec les approches assemblées (LangChain, LlamaIndex)

Pour conclure ce chapitre architectural, il est utile de comparer frontalement VVL aux frameworks les plus répandus dans le développement d'applications IA : **LangChain** et **LlamaIndex**. Ces deux outils ont joué un rôle essentiel en démocratisant l'orchestration LLM + retrieval depuis 2023, mais ils restent des bibliothèques d'assemblage — c'est-à-dire qu'ils additionnent des composants sans les unifier conceptuellement.

Reprenons l'exemple introductif (extraction de devis, classification, recherche, génération) et comparons les approches :

Aspect	LangChain / LlamaIndex	VVL
LLM	Wrapper de l'API, prompt en string	Opérateur du langage : [§ "..."], @, ?
RAG	Chain dédiée, base vectorielle externe	Builtin : ragInsert, ragAsk
Ontologie	Non native (à coder)	Système intégré : ontology:define, ontology:rule
Règles métier	Code Python ou DSL externe	Gherkin compilable : ontology:loadGherkin
Inférence symbolique	Aucune (pas de Prolog)	Moteur Prolog : kb, query, ontology:infer
Apprentissage de règles	Aucun	ontology:learn (LLM-driven ILP)
Concurrence	asyncio Python	Combinateurs ORC natifs
Cache LLM	Optionnel, à configurer	Automatique, persistant, par hash
Retry & validation	À implémenter	Automatique pour @, validation syntaxique
Lignes de code (cas type)	~500	~30

Figure 3.6 — *Comparaison fonctionnelle entre frameworks d'assemblage et runtime VVL*

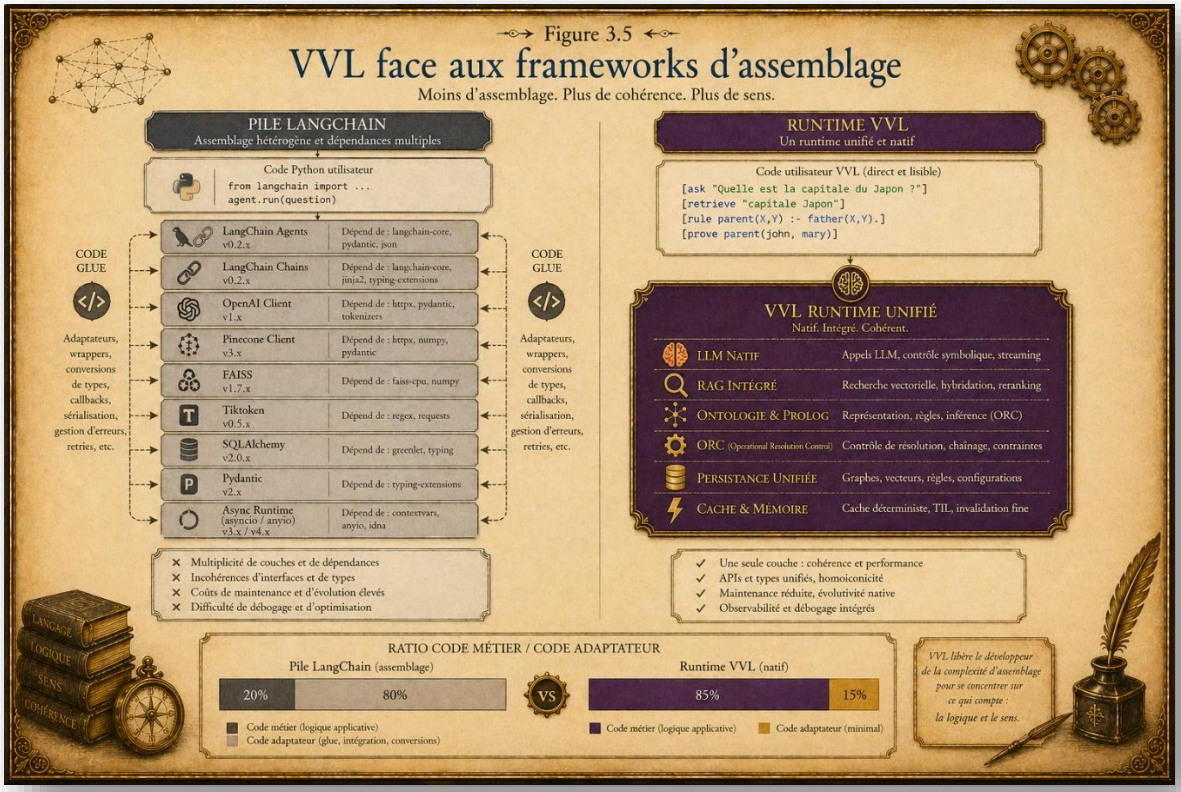


Figure 3.7 — *Le rapport code métier / code d'assemblage : un facteur 15 environ*

Trois différences structurelles méritent d'être soulignées. **Première différence : la couche symbolique.** LangChain et LlamaIndex sont entièrement neuro : ils orchestrent des appels LLM et de la recherche vectorielle, mais ils ne disposent d'aucun moteur de raisonnement symbolique intégré. Pour faire du neuro-symbolique avec eux, il faut greffer un Prolog externe, un moteur de règles tiers, et écrire le code de pont. VVL réintègre Prolog et l'ontologie comme citoyens de première classe.

Deuxième différence : l'apprentissage inductif de règles. Aucun framework d'assemblage ne propose d'équivalent à `ontology:learn`. La conséquence pratique est que dans LangChain on *écrit* les règles métier en Python ; en VVL on *fait apprendre* ces règles par le LLM à partir d'exemples, et on les obtient en Gherkin lisible par les experts. C'est un saut qualitatif sur les projets où le domaine évolue.

Troisième différence : la nature du langage hôte. LangChain est une bibliothèque Python : sa puissance s'arrête où celle de Python s'arrête, et le code utilisateur reste du Python avec ses verbosités habituelles. VVL est un langage à part entière, conçu autour de ses primitives neuro-symboliques. Cette différence est philosophique autant que technique : on ne fait pas du neuro-symbolique en empilant des bibliothèques sur un langage généraliste, on le fait en concevant un langage *autour* du paradigme.

Cela dit, VVL n'est pas un remplacement universel : pour les cas simples où une seule capacité est en jeu (un chatbot avec mémoire courte, un résumeur de documents), LangChain reste plus rapide à mettre en œuvre. La proposition de valeur de VVL apparaît dès qu'on combine plusieurs capacités — extraction structurée, raisonnement sur les faits extraits, apprentissage de règles, recherche sémantique enrichie par les inférences — c'est-à-dire dès qu'on entre véritablement en territoire neuro-symbolique.

Cette première partie s'achève ici. Le lecteur dispose maintenant du contexte historique (chapitre 1), du vocabulaire théorique (chapitre 2) et de la vision architecturale (chapitre 3). La Partie II, qui s'ouvre au chapitre suivant, plonge dans la syntaxe et les types fondamentaux du langage VVL — premier pas vers la pratique concrète de la programmation neuro-symbolique.

Partie II — Le langage

Chapitres 4 à 6 — syntaxe et types, programmation fonctionnelle, système objet

4. Syntaxe et types fondamentaux

Mots-clés : crochets, évaluation eager, variables, types primitifs, listes, JSON, XML, pattern matching, ADT, Option, Result

Ce chapitre ouvre la Partie II du livre : la pratique du langage VVL. Avant d'aborder les opérateurs neuro-symboliques, qui constituent la spécificité de VVL, il est nécessaire d'en maîtriser la syntaxe et les types fondamentaux. La bonne nouvelle est que cette syntaxe tient en une seule règle — *tout est commande entre crochets* — qui se décline ensuite uniformément dans toutes les capacités du langage.

Le lecteur familier de Lisp ou de Tcl reconnaîtra de nombreuses parentés. Le lecteur venant de Python, JavaScript ou Java trouvera un dépaysement initial mais découvrira rapidement que cette uniformité radicale produit, paradoxalement, des programmes plus courts et plus lisibles que leurs équivalents impératifs. Ce chapitre couvre la philosophie syntaxique (4.1), les variables et types primitifs (4.2), les trois représentations natives de données (4.3), les structures de contrôle (4.4) et enfin le pattern matching algébrique avec les types Option et Result (4.5).

4.1. Tout est commande entre crochets

La règle d'or de VVL tient en une formule : toute opération est un appel de commande encadré par des crochets, suivant la forme générale `[commande arg1 arg2 ...]`. Cette uniformité, héritée de Tcl et de Lisp, signifie qu'un affichage, un calcul, une définition de classe, un appel réseau ou une requête LLM partagent rigoureusement la même structure syntaxique.

```
# Forme generale : [commande arg1 arg2 ...]
[log "hello" "world"]
[log [length "Bonjour"]]                # => 7
[log (2 + 3)]                           # => 5

# Evaluation eager : interieur evalue avant exterieur
[log [length [concat "abc" "def"]]]     # => 6
#         (3)   (2)        (1)
```

L'évaluation est **eager** : les arguments sont évalués avant d'être passés à la commande. Les crochets internes sont donc résolus en premier, de l'intérieur vers l'extérieur, exactement comme en Lisp. Cette règle a une exception importante — les **builtins RAW** comme `if`, `for`, `while`, `class`, `match`, `lambda`, `keyword` reçoivent leurs arguments sans pré-évaluation, parce qu'ils doivent contrôler eux-mêmes l'ordre et le moment de l'évaluation. Sans cela, `if` évaluerait les deux branches, ce qui annulerait toute son utilité.

Trois formes syntaxiques voisines doivent être distinguées soigneusement, car elles signifient des choses très différentes :

Forme	Sens	Exemple	Évaluation
[a b c]	Appel de commande ou liste de données (selon le 1er élément)	[log "x"]	Args évalués
(expr)	Expression arithmétique infixée	(2 + 3 * 4)	Calcul direct
`[a b c]	Quasiquote : empêche l'évaluation	`[1 2 3]	Donnée littérale

Figure 4.1 — *Trois formes syntaxiques voisines, trois sémantiques distinctes*

Le **backtick** (quasiquote) est emprunté directement à Lisp : il transforme une expression en donnée littérale, sans évaluation. C'est ce qui permet de stocker une liste comme valeur (`$nums = \[1 2 3]`) sans que `1` soit interprété comme nom de commande. Cette distinction est l'un des rares pièges syntaxiques de VVL : oublier le backtick devant une liste de valeurs déclenche immédiatement une erreur d'évaluation.

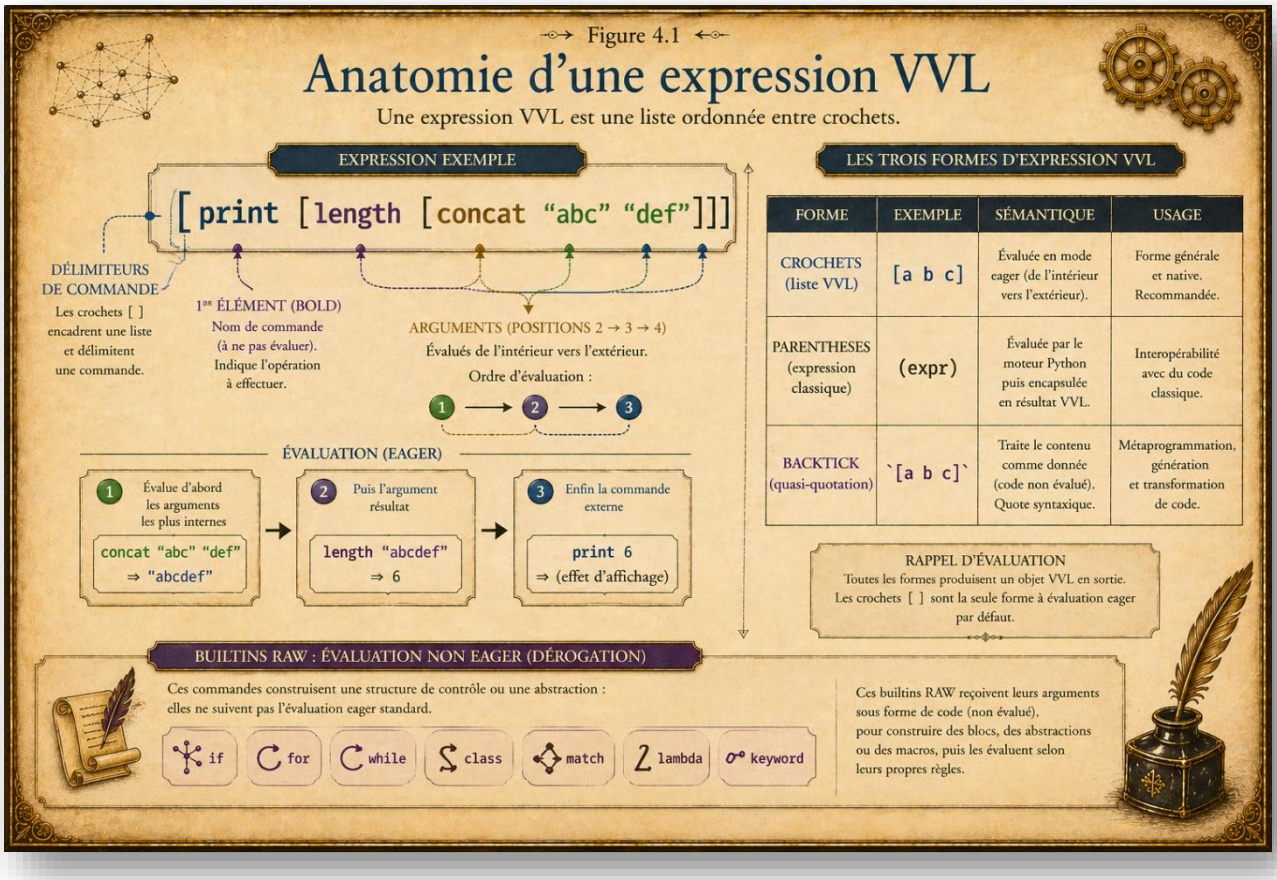


Figure 4.2 — *Anatomie d'une expression VVL et ses trois variantes syntaxiques*

4.2. Variables, types primitifs et opérateurs

Les variables en VVL sont préfixées par le symbole `$` et s'assignent avec l'opérateur `=`. Il n'existe pas de commande `set` ni de mot-clé de déclaration : l'assignation crée implicitement la variable si elle n'existe pas encore. Les noms peuvent contenir des lettres, des chiffres, des underscores et des tirets.

```
# Assignation simple : creation implicite
$name = "Alice"
$age  = 30
$pi   = 3.14159

# Lecture et expressions
[log "Nom : " $name]                    # => Nom : Alice
[log ($age + 1)]                        # => 31

# Destructuration via mset
[mset [$x $y $z] `[10 20 30]]
[log $x " " $y " " $z]                  # => 10 20 30

# Variables dynamiques : ${$var} accede au nom contenu dans une variable
$varName = "foo"
$foo     = "bar"
[log ${$varName}]                        # => bar

# Inference de type a l'execution
$x = 7
[log (3 + $x)]                          # => 10  (addition numerique)
$x = "monde"
[log ("hello " + $x)]                   # => hello monde  (concatenation)
```

Le typage est **dynamique** : les variables n'ont pas de type déclaré, et le moteur détermine au runtime l'opération à effectuer en fonction du contenu. L'expression `(3 + $x)` produira une addition numérique si `$x` contient un nombre, ou une concaténation si `$x` contient une chaîne. Cette flexibilité est cohérente avec la philosophie générale du langage et avec son intégration LLM, où le typage strict serait un obstacle.

VVL reconnaît quatre types de données fondamentaux. Le booléen est représenté en interne par `1` et `0`, ce qui simplifie les compositions arithmétiques et logiques. Les listes sont la structure de données universelle, héritée de Lisp ; elles servent à la fois pour les données et pour le code (homoiconicité, cf. chapitre 3).

Type	Syntaxe littérale	Exemples	getType retourne
String	Guillemets doubles	"hello", "ligne1\nligne2"	STRING
Num	Littéral numérique	42, 3.14, -7, 1e6	NUM
Boolean	Mots-clés ou Num	true, false, 1, 0	NUM (1 ou 0)
List	Crochets ou backtick	[1 2 3], `["a" "b"]	LIST

Figure 4.3 — *Les quatre types fondamentaux de VVL*

Les conversions explicites s'effectuent avec les commandes `INT`, `FLOAT`, `BOOL`, `STRING`, ou la commande générique `convert`. La commande `getType` retourne le nom du type d'une valeur sous forme de chaîne — utile pour le débogage ou pour écrire des fonctions polymorphes.

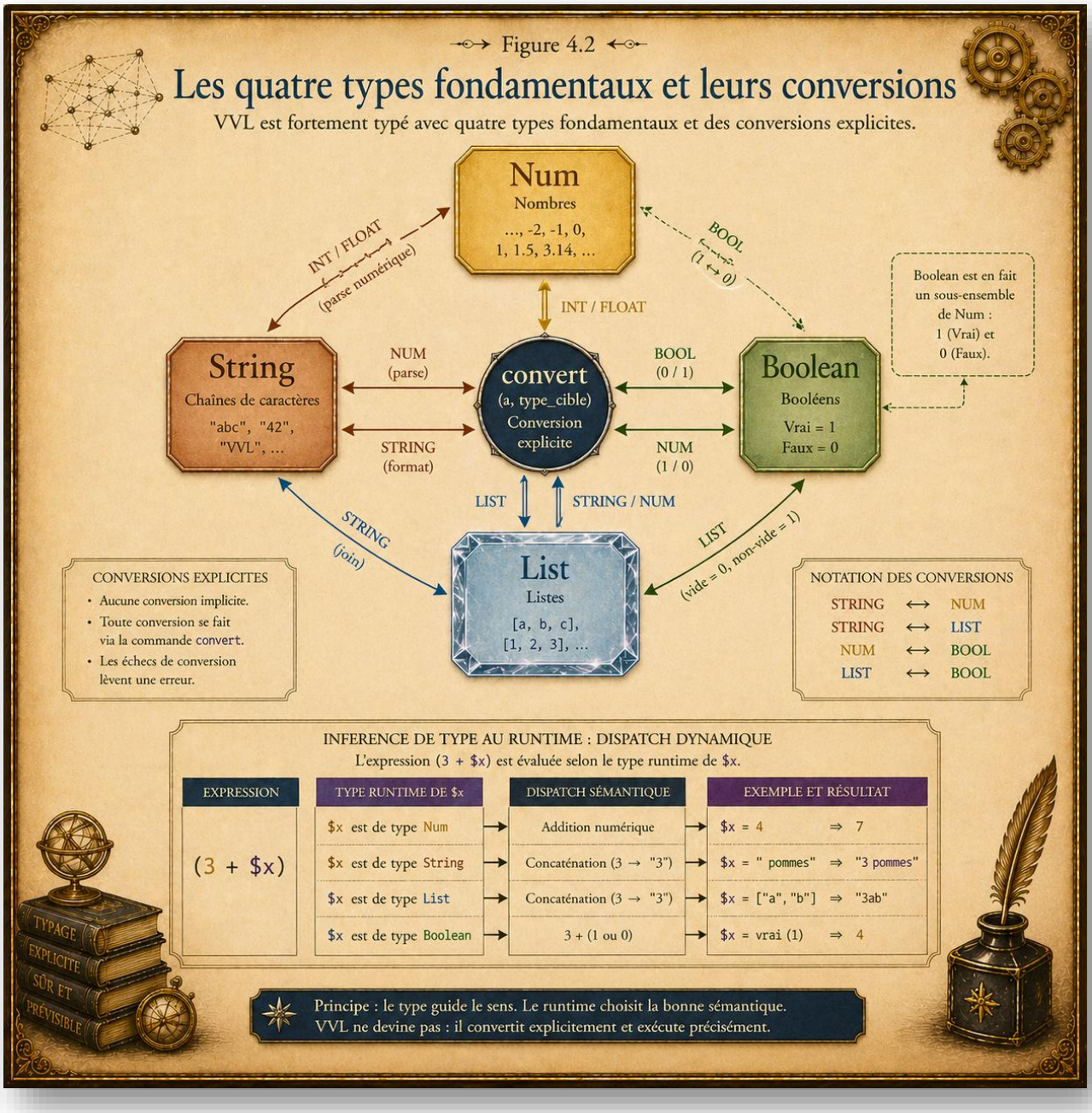


Figure 4.4 — *Le système de types dynamique de VVL et ses conversions*

4.3. Listes, JSON et XML : trois syntaxes natives

Une caractéristique distinctive de VVL est de proposer **trois syntaxes littérales pour les données structurées** : la liste Lisp, le JSON et le XML. Cette redondance apparente est en réalité un choix délibéré : chacune de ces représentations est dominante dans un domaine d'usage différent (interne au langage pour les listes, communication API pour le JSON, configuration et documents pour le XML), et VVL les supporte nativement pour éviter les conversions et les bibliothèques de parsing.

Listes natives

Les listes sont la structure universelle. Elles s'écrivent avec des crochets, et doivent être préfixées d'un backtick lorsqu'elles contiennent des valeurs littérales (sinon le premier élément serait interprété comme nom de commande). Les opérations classiques — append, pop, slice, reverse, range, flat — sont disponibles comme builtins.

```
# Creation
$nums  = `[1 2 3 4 5]
$mots  = `["alpha" "beta" "gamma"]
$mixte = `[1 "deux" 3.0 true]

# Acces et longueur
[log [getValue 1 $nums]]                  # => 1 (1-based)
[log [length $nums]]                      # => 5

# Manipulation
$plus = [append $nums 6]                    # => [1 2 3 4 5 6]
$tail = [slice $nums 2 4]                   # => [2 3 4]
$rev  = [reverse $nums]                     # => [5 4 3 2 1]

# Generation : range
$seq  = [range 1 11 1]                      # => [1 2 3 4 5 6 7 8 9 10]

# Destructuration head/tail (style fonctionnel)
[unpack $nums $head $tail]
[log "tete=" $head " queue=" $tail]       # => tete=1 queue=[2 3 4 5]

# Iteration
[for $n $nums [log $n]]                   # 1, 2, 3, 4, 5
```

JSON natif

Le JSON est représenté par les littéraux `<{...}>` (objet) et `<[...]>` (tableau). Cette syntaxe permet d'écrire des données JSON exactement comme dans une API REST, sans échappement ni parsing manuel. Les builtins `jsonGetValue`, `jsonToList`, `jsonKeys` permettent d'inspecter et de naviguer dans la structure.

```
# Litteral JSON natif (objet)
$user = <{
    "name": "Alice",
    "age": 30,
    "skills": ["python", "vvl"],
    "active": true
}>

# Acces aux champs
[log [jsonGetValue "name" $user]]         # => Alice
[log [jsonGetValue "age" $user]]          # => 30
[log [jsonGetValue "skills" $user]]       # => ["python","vvl"]

# Conversion JSON list -> liste VVL
$skills_list = [jsonToList [jsonGetValue "skills" $user]]
[for $s $skills_list [log $s]]            # python, vvl

# Litteral tableau JSON
$products = <[
    {"id": 1, "name": "Widget"},
    {"id": 2, "name": "Gadget"}
]>
```

XML natif

Le XML est représenté par les littéraux `<<...>>`. Cette syntaxe est précieuse pour la manipulation de configurations, de documents structurés ou de formats hérités. Les builtins `getXMLValue` et `setXMLValue` opèrent par chemin XPath simplifié.

```
# Litteral XML natif
$config = <<
    <server>
        <host>api.example.com</host>
        <port>443</port>
        <auth type="bearer">
            <token>secret123</token>
        </auth>
    </server>
>>

# Acces par chemin
[log [getXMLValue "server/host" $config]]      # => api.example.com
[log [getXMLValue "server/auth/@type" $config]] # => bearer  (attribut)

# Modification
$config2 = [setXMLValue "server/port" "8443" $config]
```

La cohérence entre ces trois syntaxes est garantie par le runtime : un objet JSON peut être converti en liste VVL, une liste VVL peut être sérialisée en JSON, et le XML supporte les mêmes opérations de transformation. Cette interopérabilité native évite l'écueil des langages classiques où l'on jongle entre `dict`, `list`, `json.loads`, `ElementTree.parse`, chacun avec ses propres méthodes et son propre comportement face aux types.

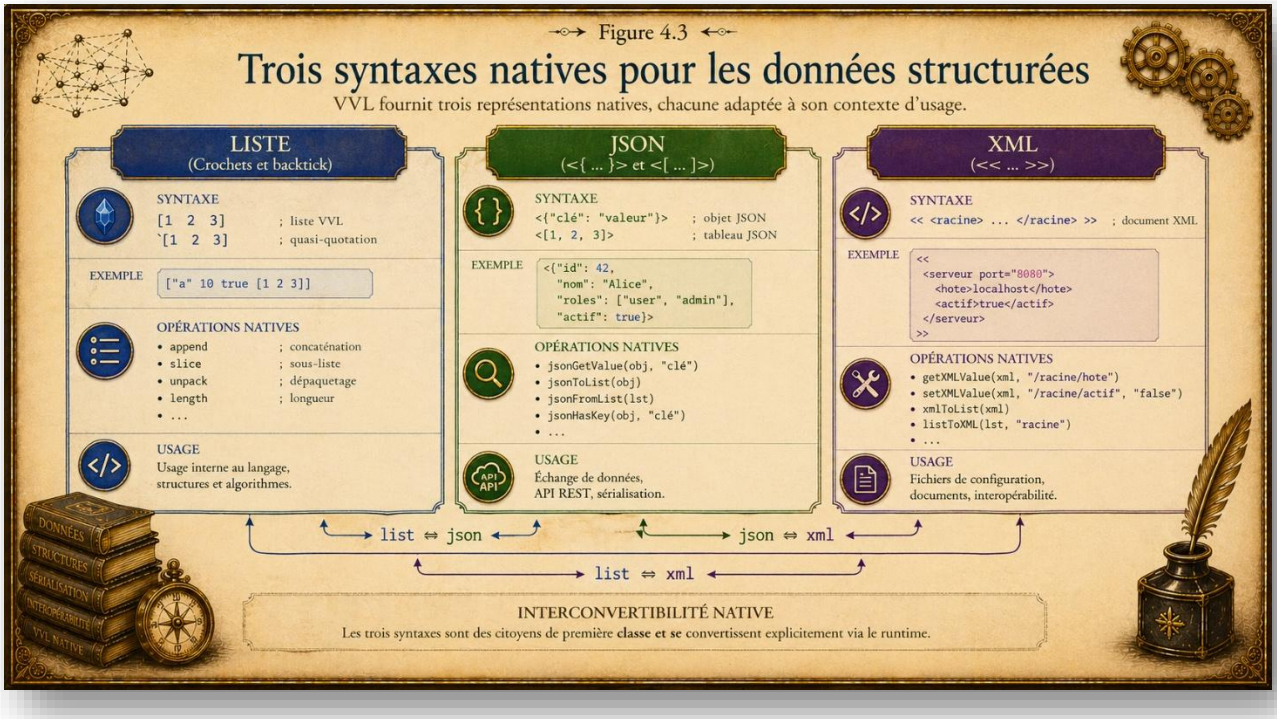


Figure 4.5 — *Listes, JSON et XML : trois représentations interopérables nativement*

4.4. Structures de contrôle : if, when, for, while, match

Les structures de contrôle de VVL couvrent les cas habituels (conditionnelles, boucles, gestion d'erreurs) avec quelques particularités héritées de la philosophie homoiconique. La distinction la plus importante à intégrer est celle entre **expression** (qui retourne une valeur, comme `match`) et **instruction** (qui exécute pour ses effets, comme `switch`).

Conditionnelles : if, when, not

La commande `if` prend trois arguments : la condition, la branche vraie, la branche fausse. Si l'on veut une condition sans branche fausse, on utilise `when`. La commande `not` inverse une condition. Toutes ces commandes sont des **builtins RAW** : elles n'évaluent pas leurs arguments avant de les recevoir, ce qui est essentiel pour éviter d'exécuter la mauvaise branche.

```
# if : condition / branche vraie / branche fausse
$age = 18
[if ($age >= 18)
    [log "Majeur"]
    [log "Mineur"]
]

# when : condition / branche unique
[when ($age >= 18) [log "Acces autorise"]]

# Boucle for : iteration sur liste
[for $i `[1 2 3] [log "Iter " $i]]

# Boucle while : tant que la condition est vraie
$counter = 0
[while ($counter < 3)
    [do
        [log "counter=" $counter]
        $counter = ($counter + 1)
    ]
]

# Bloc do : execute en sequence, retourne la derniere valeur
$result = [do
    $a = 10
    $b = 20
    ($a + $b)                              # valeur de retour du bloc
]
[log $result]                             # => 30

# silent : execute sans produire de sortie
[silent [log "Invisible"]]
```

Match : expression de pattern matching

La commande `match` est l'une des plus puissantes du langage. Elle compare une valeur à une série de paires (pattern, body), retourne la valeur du body matché, et supporte plusieurs niveaux de sophistication : littéraux, wildcards, captures de variables, matching structurel de listes.

```
# Match litteral avec retour de valeur
$code = 418
$msg = [match $code
    200 "OK"
    404 "Not Found"
    500 "Server Error"
    "_" "Code inconnu"                      # wildcard
]
[log $msg]                                  # => Code inconnu

# Capture avec ?name (accessible via $__name dans le body)
$val = 42
[match $val
    ?v [log "capture : " $__v]              # => capture : 42
]

# Matching structurel sur liste
$point = `[1 99]
[match $point
    `[0 0]    [log "origine"]
    `[1 ?y]   [log "x=1, y=" $__y]          # => x=1, y=99
    `[?x ?y]  [log "x=" $__x " y=" $__y]
]

# Match imbrique (dispatch bidimensionnel)
$outer = "b"
$inner = 2
[log [match $outer
    "a" [match $inner 1 "a1" 2 "a2" "a?"]
    "b" [match $inner 1 "b1" 2 "b2" "b?"]
    "autre"
]]
                                        # => b2
```

Switch : instruction de branchement

Là où `match` retourne une valeur, `switch` exécute des blocs pour leurs effets. Il est préférable lorsque les branches contiennent des effets de bord (affichage, écriture fichier, modification d'état) plutôt qu'un calcul de valeur.

Gestion d'erreurs : try / catch

La commande `try` prend exactement deux arguments : le bloc à essayer et le bloc de récupération. Si une exception survient dans le premier, le contrôle passe immédiatement au second. VVL n'a pas de `finally` — le nettoyage doit être effectué explicitement. Pour les flux sans exception, la programmation fonctionnelle préfère cependant les types Result, présentés à la section suivante.

```
# try : essai et capture
[try
    [do
        [log "Tentative..."]
        $r = (1 / 0)                        # division par zero
        [log "Inatteignable"]
    ]
    [log "Erreur interceptee"]
]
```

4.5. Pattern matching algébrique : types Option et Result

L'application la plus puissante du pattern matching structurel de VVL est la construction de **types de données algébriques** (ADT). Ces types, omniprésents en Haskell, Rust, Scala et OCaml, modélisent des valeurs qui peuvent prendre plusieurs formes mutuellement exclusives. Les deux ADT fondamentaux sont **Option** (une valeur peut être présente ou absente) et **Result** (une opération peut réussir ou échouer avec une raison qualifiée).

Dans les langages classiques, l'absence de ces types conduit aux deux pathologies les plus fréquentes du code de production : les *null pointer exceptions* (oubli de tester si une valeur est nulle) et les codes d'erreur numériques opaques (-1, 0, 999, dont la signification dépend du contexte). VVL implémente les ADT via des **listes tagguées** et le matching structurel — donc sans extension du langage, simplement par convention.

Le type Option : Some / None

Une `Option` est une liste tagguée : `\[Some valeur]` `représente une valeur présente,` `\[None]` représente l'absence. Le pattern matching extrait la valeur en toute sécurité — il est impossible d'oublier le cas `None`, car le compilateur passera au branche par défaut si on l'omet, ce qui rend l'erreur visible immédiatement.

```
# Construction d'Options
$present = `[Some 42]
$absent  = `[None]

# Pattern matching pour extraction
[match $present
    `[Some ?v] [log "valeur : " $__v]      # => valeur : 42
    `[None]    [log "aucune valeur"]
]

# Bibliotheque Option en quelques keywords
[keyword isSome `[if ([getValue 1 @1] == "Some") 1 0]]
[keyword unwrapOr
    `[if [isSome @1] [getValue 2 @1] @2]
]
[keyword mapOption
    `[if [isSome @2]
        `[Some [@1 [getValue 2 @2]]]
        `[None]]
]

# Usage
[log [unwrapOr `[Some 42] 0]]              # => 42
[log [unwrapOr `[None]    0]]              # => 0  (defaut)

$inc = [lambda [x] ($__x + 1)]
[log [mapOption $inc `[Some 42]]]          # => [Some 43]
[log [mapOption $inc `[None]]]             # => [None]
```

Le type Result : Ok / Error

Le type `Result` étend l'idée d'`Option` en ajoutant une raison qualifiée à l'échec. `\[Ok valeur]` `représente un succès,` `\[Err code message]` représente un échec. C'est l'alternative fonctionnelle aux exceptions : l'erreur est une **valeur** comme une autre, qui se propage par composition de fonctions plutôt que par interruption du flux d'exécution.

```
# Division sure : retourne un Result
[keyword safeDiv
    `[if (@2 == 0)
        `[Err "DIV_ZERO" "Division par zero"]
        `[Ok (@1 / @2)]]
]

# Pattern matching pour traiter le resultat
[match [safeDiv 10 2]
    `[Ok ?v]      [log "resultat=" $__v]  # => resultat=5
    `[Err ?c ?m]  [log "erreur " $__c " : " $__m]
]

[match [safeDiv 10 0]
    `[Ok ?v]      [log "resultat=" $__v]
    `[Err ?c ?m]  [log "erreur " $__c " : " $__m]
    # => erreur DIV_ZERO : Division par zero
]

# Composition : la chaine s'arrete au premier Err
[keyword andThenResult
    `[match @1
        `[Ok ?v]      [@2 $__v]
        `[Err ?c ?m]  @1                    # propagation de l'erreur
    ]
]

# Pipeline securise
$result = [andThenResult [safeDiv 100 5]
              [lambda [x] [safeDiv $__x 4]]]
[log $result]                               # => [Ok 5]
```

Le tableau ci-dessous compare `Option` et `Result` :

	Option (Some/None)	Result (Ok/Err)
Cas de succès	`[Some valeur]	`[Ok valeur]
Cas d'échec	`[None]	`[Err code message]
Information sur l'échec	Aucune (juste « absent »)	Qualifiée (code + message)
Composition	mapOption	mapResult, andThenResult
Analogie	Optional (Java), Option (Rust)	Result (Rust), Either (Haskell)
Cas d'usage	Valeur potentiellement absente	Opération potentiellement échouante

Figure 4.6 — *Comparaison fonctionnelle d'Option et Result*

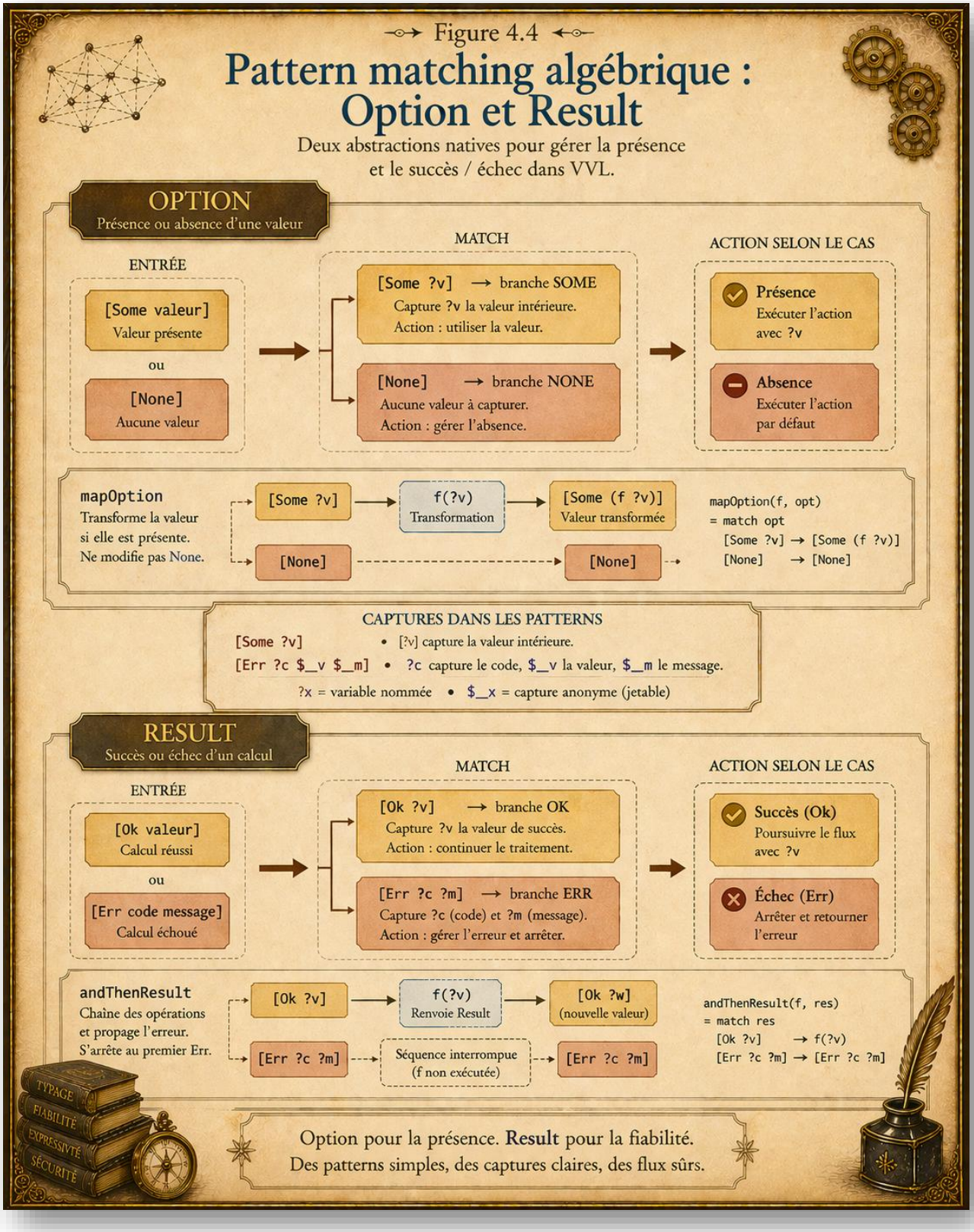


Figure 4.7 — *Les deux ADT fondamentaux et leurs patterns d'utilisation*

Trois propriétés font la valeur de cette approche. **L'explicite l'emporte sur l'implicite** : le type de retour annonce clairement qu'une opération peut échouer, et le compilateur — ou plus précisément,

le développeur attentif au pattern matching — ne peut pas l'ignorer. **La composition est sans effet de bord** : les opérations s'enchaînent par fonctions, pas par interruption ; le programme reste prévisible et testable. **La propagation est uniforme** : qu'une chaîne ait deux étapes ou cinquante, le mécanisme est identique.

Dans les chapitres ultérieurs sur le neuro-symbolique, ces types prendront une importance particulière. L'extraction LLM peut échouer (texte ambigu, document corrompu) — `Result` est le bon véhicule pour le signaler. Une recherche RAG peut ne rien retourner — `Option` exprime cette absence sans confusion. Le pipeline neuro-symbolique du chapitre 18 compose plusieurs étapes faillibles en utilisant systématiquement `andThenResult` pour la propagation.

Avec ce chapitre, le lecteur maîtrise la grammaire de base de VVL. Le chapitre 5 enchaîne sur la programmation fonctionnelle — lambdas, keywords, fonctions d'ordre supérieur — qui constituent l'étape suivante avant d'aborder les opérateurs sémantiques de la Partie III.

5. Programmation fonctionnelle

Mots-clés : lambda, keyword, macro, fonction d'ordre supérieur, map, filter, pmap, composition, fiber, générateur, évaluation paresseuse, coroutine

La programmation fonctionnelle n'est pas un style décoratif que l'on adopterait pour le plaisir de l'élégance : c'est le **fondement opérationnel** de la programmation neuro-symbolique en VVL. Les opérateurs sémantiques de la Partie III — notamment l'opérateur `@` qui fait générer du code par le LLM — produisent des **lambdas** que le langage exécute immédiatement. Sans lambdas de première classe, ces opérateurs n'existeraient tout simplement pas. Sans fonctions d'ordre supérieur (`map`, `filter`), les pipelines neuro-symboliques deviendraient des boucles impératives ingérables. Sans `pmap`, les appels LLM concurrents seraient impossibles à orchestrer.

Ce chapitre couvre quatre constructions fondamentales : les **lambdas** (5.1), qui sont les fonctions anonymes de première classe, les **keywords** (5.2), qui sont des fonctions nommées doublées d'un système de macros à la Lisp, les **fonctions d'ordre supérieur** (5.3) qui transforment les collections par application de fonctions, et enfin les **fibers** (5.4) qui implémentent l'évaluation paresseuse via des coroutines à la `yield` / `resume`. Chacune de ces constructions trouvera une application directe dans les chapitres ultérieurs sur le neuro-symbolique.

5.1. Lambdas et fermetures

Le **lambda calcul**, inventé par Alonzo Church en 1936, est un système formel où toute computation est exprimée comme application de fonctions anonymes. C'est le fondement théorique de tous les langages fonctionnels (Lisp, Haskell, ML, Scheme), et il est prouvé Turing-complet : toute computation réalisable par un programme peut être exprimée en lambda calcul. VVL hérite directement de cette tradition via Lisp.

Une lambda est créée par la commande `lambda`, qui prend deux arguments : une liste de paramètres et un corps. Les paramètres sont déclarés entre crochets et sont accessibles dans le corps avec le préfixe `$__` (double underscore) — la même convention que les captures de match. Ce préfixe garantit l'**isolation de portée** : la lambda ne modifie jamais les variables de la portée englobante. Le symbole Unicode `λ` est un alias de `lambda` pour une syntaxe plus concise.

```
# Definition de lambdas
$add    = [lambda [a b] ($__a + $__b)]
$square = [lambda [x]   ($__x * $__x)]

[log [$add 3 4]]                          # => 7
[log [$square 5]]                         # => 25

# Syntaxe Unicode : λambda
$inc = [lambda [x] ($__x + 1)]
[log [$inc 10]]                           # => 11

# Lambda multi-instructions avec do
$greet = [lambda [name] [do
    $msg = [concat "Bonjour, " $__name " !"]
    [log "Creation du message pour " $__name]
    $msg                                    # derniere expression = retour
]]
[log [$greet "Alice"]]                    # => Bonjour, Alice !

```

```
# Lambda sans parametre
$hello = [lambda [] [log "Hello World !"]]
[$hello]                                  # => Hello World !
```

Les lambdas sont des **valeurs de première classe** : elles peuvent être stockées dans des variables, passées en argument à d'autres fonctions, retournées comme résultats. C'est cette propriété qui rend possible la programmation fonctionnelle authentique, par opposition aux langages impératifs qui n'offrent que des callbacks limités.

```
# Lambda stockee dans une variable
$op = [lambda [a b] ($__a * $__b)]

# Lambda passee en argument
$apply = [lambda [fn x y] [$__fn $__x $__y]]
[log [$apply $op 6 7]]                    # => 42

# Lambda retournee par une autre lambda : currying
$multiplier = [lambda [factor]
    [lambda [x] ($__x * $factor)]
]
$triple = [$multiplier 3]
[log [$triple 5]]                         # => 15
[log [$triple 8]]                         # => 24

# Fermeture sur une variable de la portee englobante
$counter = 0
$tick    = [lambda [] [do
    $counter = ($counter + 1)
    $counter
]]
# Note : les fermetures VVL capturent par reference dans le contexte parent
```

Dans le contexte spécifique de VVL, les lambdas ont une **importance neuro-symbolique** que les langages classiques n'offrent pas : l'opérateur `@` du chapitre 8 demande au LLM de produire une lambda VVL à partir d'une description en langage naturel. Grâce à l'homoiconicité (chapitre 3), le code généré est directement exécutable sans compilation ni sandboxing. Le LLM devient un **méta-programmeur** qui produit des fonctions, et les lambdas sont l'unité naturelle de cette production. C'est par exemple ainsi que l'on écrit `$filtre = ($data @ [§ "filtrer par age > 30"])` puis `[filter $filtre $personnes]` — la lambda générée par le LLM est immédiatement applicable.

5.2. Keywords : fonctions nommées et macros

La commande `keyword` définit un bloc de code réutilisable associé à un nom. Après définition, le keyword s'invoque exactement comme une commande built-in. C'est l'équivalent de `def` en Python, `function` en JavaScript, ou `defun` en Lisp — mais avec une différence majeure : le **corps est un template** (quasiquote, marqué par le backtick) et les paramètres sont accédés via `@1`, `@2`, etc., pas via des noms explicites. Le corps n'est évalué qu'à l'appel, avec substitution des paramètres.

```
# Definition et appel d'un keyword
[keyword saluer `[log "Bonjour, " @1 " !"]]
[saluer "Alice"]                          # => Bonjour, Alice !
[saluer "Bob"]                            # => Bonjour, Bob !

# Keyword avec arithmetique
[keyword add2 `(@1 + @2)]
[log [add2 3 5]]                          # => 8

# Keyword avec structure de controle
[keyword max2 `[if (@1 > @2) @1 @2]]
[log [max2 4 9]]                          # => 9
[log [max2 12 3]]                         # => 12

# Keywords recursifs : factorielle
[keyword fact `[if (@1 <= 1) 1 (@1 * [fact (@1 - 1)])]]
[log [fact 5]]                            # => 120
[log [fact 10]]                           # => 3628800

# Recursion sur liste : somme
[keyword sum-list `[if ([length @1] == 0)
    0
    [do
        [unpack @1 $head $tail]
        ($head + [sum-list $tail])
    ]
]]
[log [sum-list `[1 2 3 4 5]]]             # => 15
```

La distinction fondamentale entre `lambda` et `keyword` est leur nature : une lambda est une **valeur anonyme stockable**, un keyword est un **nom global invocable** comme une commande. Les lambdas sont utilisées pour les fonctions de transformation éphémères (arguments de `map`, `filter`, callbacks d'événements) ; les keywords pour les opérations nommées réutilisables dans tout le programme.

Aspect	lambda	keyword
Syntaxe	[lambda [params] body]	[keyword name `[body @1]]
Nommage	Anonyme (assignable à une variable)	Nommé (global)
Paramètres	[a b] → $__a, $__b	@1, @2, @3...
Corps	Évalué à l'appel, capture le contexte	Template (backtick), évalué à l'appel
Portée	Locale (closure)	Globale
Récursion	Possible si la lambda est nommée	Native (auto-référence par nom)
Usage typique	Argument de map/filter, callback	Fonction nommée, macro, générateur

Figure 5.1 — *Comparaison fonctionnelle entre lambda et keyword*

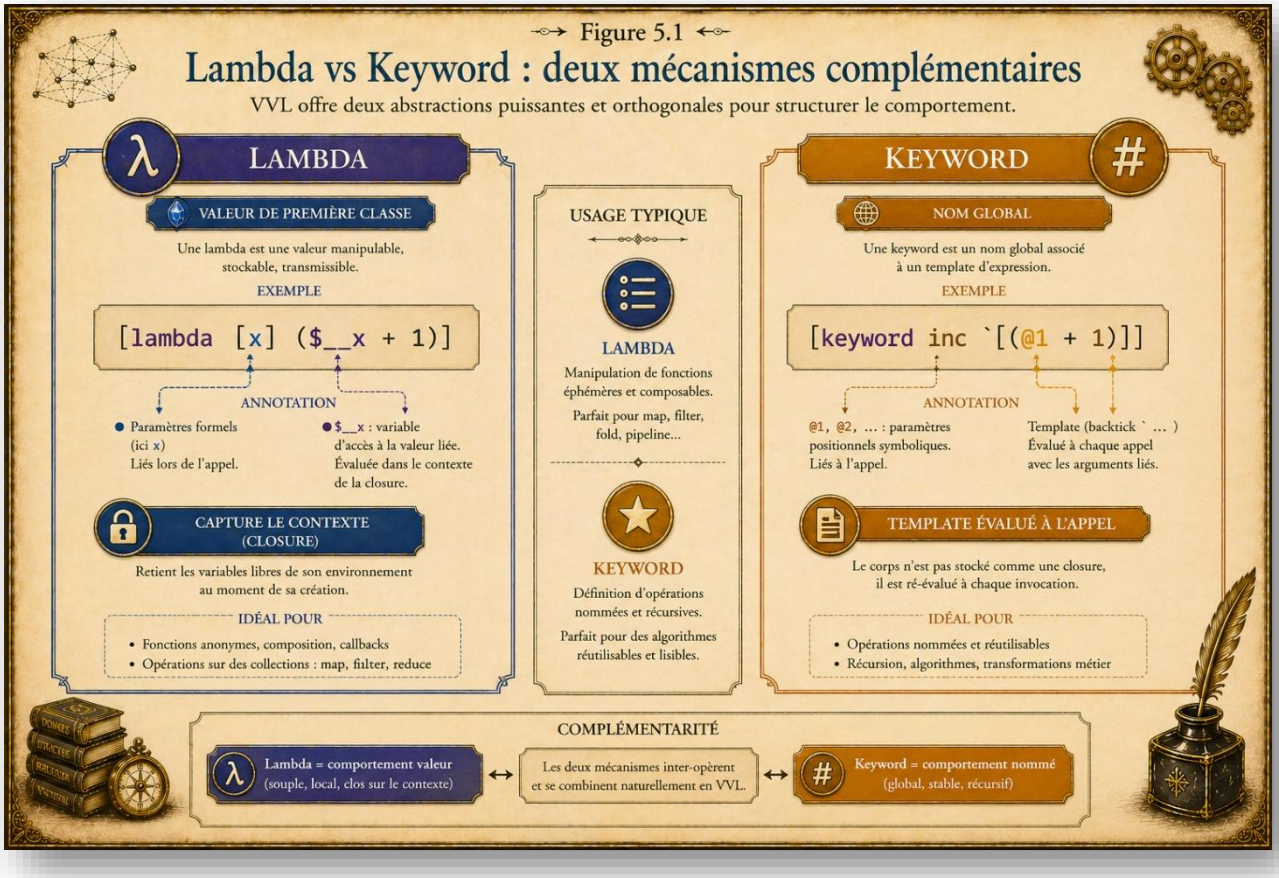


Figure 5.2 — *Lambda et keyword : deux mécanismes pour deux usages distincts*

5.3. Fonctions d'ordre supérieur : map, filter, pmap

Une **fonction d'ordre supérieur** est une fonction qui prend une autre fonction en argument ou qui en retourne une. C'est le mécanisme central de la programmation fonctionnelle : au lieu de boucler sur des données avec des effets de bord, on transforme des collections entières par application de fonctions. VVL fournit trois fonctions d'ordre supérieur de base — `map`, `filter` et `pmap` — qui couvrent l'essentiel des besoins de transformation de listes.

map : transformation un-à-un

`map` applique une fonction à chaque élément d'une liste et retourne la liste des résultats. La sémantique est purement fonctionnelle : la liste source n'est jamais modifiée.

```
# map : transformer chaque element
$nums = `[1 2 3 4 5]

# Avec une lambda
$doubles = [map [lambda [x] ($__x * 2)] $nums]
[log $doubles]                          # => [2, 4, 6, 8, 10]

# Avec un keyword utilise comme fonction
[keyword sq `[(@1 * @1)]]
$squares = [map sq $nums]
[log $squares]                          # => [1, 4, 9, 16, 25]

```

```
# filter : selectionner les elements satisfaisant un predicat
$evens = [filter [lambda [x] (($__x % 2) == 0)] $nums]
[log $evens]                                # => [2, 4]

# Composition naturelle : range -> map -> filter
$bigSquares = [filter [lambda [x] ($__x > 20)]
                  [map [lambda [x] ($__x * $__x)]
                      [range 1 10 1]
                  ]
              ]
[log $bigSquares]                           # => [25, 36, 49, 64, 81]
```

pmap : map parallèle

`pmap` a la même sémantique que `map` mais exécute les transformations **en parallèle**. C'est l'outil canonique pour orchestrer des appels coûteux en I/O — typiquement des appels LLM ou des requêtes HTTP — sans construire manuellement une infrastructure asyncio. L'API est strictement identique à `map`, ce qui permet souvent de paralléliser un pipeline existant en remplaçant un seul mot.

```
# pmap : exactement la meme syntaxe que map
$urls = `[
    "https://api.exemple.com/produit/1"
    "https://api.exemple.com/produit/2"
    "https://api.exemple.com/produit/3"
]

# Sequentiel (3 requetes, environ 600 ms)
$data_seq = [map [lambda [u] [httpGET $__u]] $urls]

# Parallele (3 requetes, environ 200 ms)
$data_par = [pmap [lambda [u] [httpGET $__u]] $urls]

# Cas typique neuro-symbolique : extraction parallele de N documents
[keyword extract `[ontology:fromText "Devis" @1]]
$devis = [pmap extract $textes]             # N appels LLM en parallèle
```

Composition et pipelines

La composition de fonctions est l'art d'enchaîner les transformations sans variables intermédiaires. En VVL, elle s'exprime naturellement par imbrication de `map` et `filter` — le résultat de l'expression intérieure devient l'argument de l'expression extérieure. Cette écriture déclarative remplace les boucles impératives à compteurs et accumulateurs des langages classiques.

Le **pattern reduce** (réduction d'une collection à une seule valeur) n'a pas de builtin natif en VVL mais s'implémente idiomatiquement via une boucle `for` avec accumulateur :

```
# Pattern reduce : somme par accumulateur
$sum = 0
[for $n `[1 2 3 4 5]
    $sum = ($sum + $n)
]
[log "Somme : " $sum]                       # => 15

# Pattern reduce : trouver le maximum
$max = -999999
[for $x `[3 7 12 5 18 2]
    [when ($x > $max) $max = $x]
]
[log "Max : " $max]                         # => 18
```

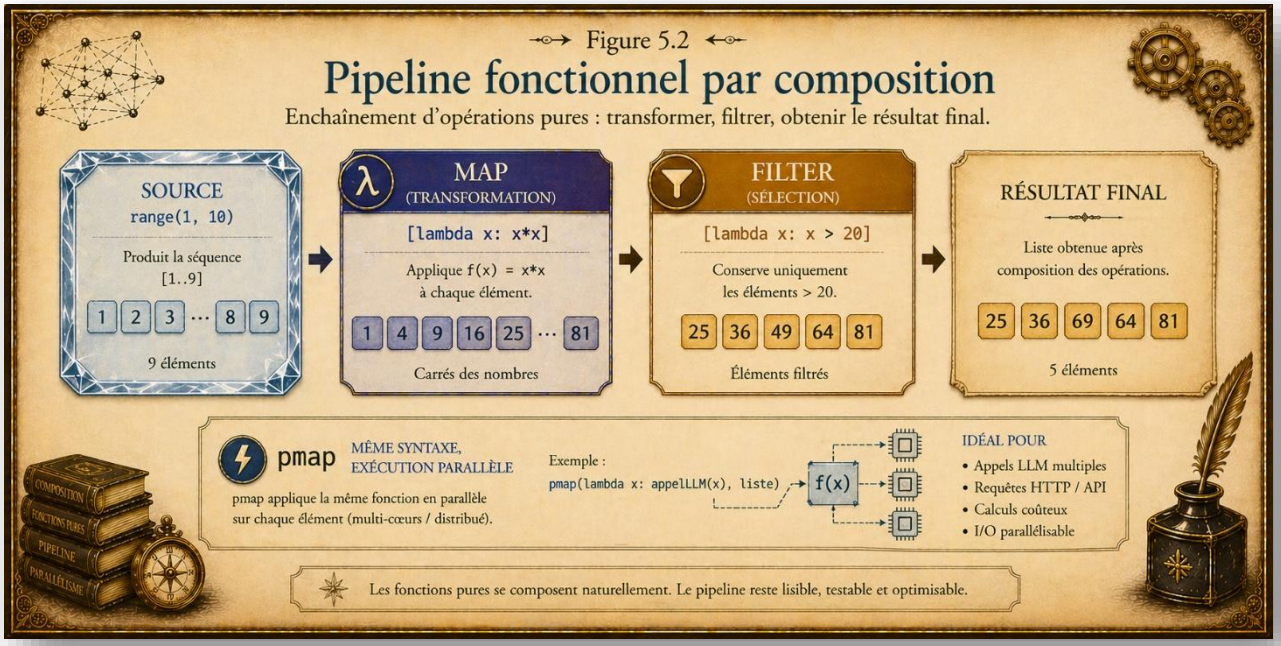


Figure 5.3 — *Composition fonctionnelle : range, map, filter chaînés sans variables intermédiaires*

5.4. Générateurs et fibers : évaluation paresseuse

Un **fiber** (ou générateur) est une coroutine qui produit des valeurs **à la demande**. Contrairement à une fonction qui calcule toutes ses valeurs immédiatement et retourne une liste, un fiber calcule chaque valeur seulement quand on la demande (évaluation paresseuse). L'exécution est suspendue à chaque `yield` et reprend exactement au même point au prochain `fiber-resume`. Ce mécanisme est fondamental pour trois raisons :

Mémoire constante. Un fiber produisant un million de valeurs n'occupe que la mémoire d'une seule valeur à la fois, là où une liste d'un million d'éléments consommerait toute la mémoire disponible. C'est crucial dans les pipelines de traitement de gros volumes.

Flux infinis. Un fiber peut produire une séquence infinie — un compteur, un flux de données réseau, une suite mathématique. Une liste, par définition, ne peut pas être infinie. Les fibers permettent ainsi de programmer naturellement avec des séquences potentiellement non bornées, le consommateur s'arrêtant quand il a obtenu ce qu'il cherchait.

Découplage producteur–consommateur. Le producteur (le fiber) et le consommateur (le code appelant) sont découplés dans le temps : le producteur ne calcule que ce que le consommateur demande, et il peut maintenir un état interne complexe entre les appels sans le révéler à l'extérieur.

Les fibers sont créés en deux étapes : définition d'un keyword contenant des `yield`, puis instanciation via `spawn`.

```
# Etape 1 : definir le generateur avec keyword
[keyword produce3 `[do
    [yield 10]
    [yield 20]
    [yield 30]
]]

# Etape 2 : instancier le fiber avec spawn
$fib = [spawn produce3]

# Consommer les valeurs une par une
[log [fiber-resume $fib]]                # => 10
[log [fiber-resume $fib]]                # => 20
[log [fiber-alive $fib]]                 # => 1  (encore une valeur)
[log [fiber-resume $fib]]                # => 30
[log [fiber-alive $fib]]                 # => 0  (epuise)
```

Les fibers acceptent des paramètres via `@1`, `@2`, etc. (comme tous les keywords) et peuvent contenir des boucles internes. La commande `fiber-to-list` collecte toutes les valeurs restantes en une seule liste — utile lorsqu'on veut basculer d'une consommation paresseuse à un traitement par lot.

```
# Fiber parametre avec boucle interne
[keyword counter `[do
    $__max = @1
    [for $__i [range 0 $__max 1]
        [yield $__i]
    ]
]]

# Instancier avec un argument
$gen = [spawn counter 5]

# Consommer 3 valeurs manuellement
[log [fiber-resume $gen]]                # => 0
[log [fiber-resume $gen]]                # => 1
[log [fiber-resume $gen]]                # => 2

# Drainer le reste en une seule liste
$rest = [fiber-to-list $gen]
[log "Reste : " $rest]                   # => [3, 4]
[log [fiber-alive $gen]]                 # => 0
```

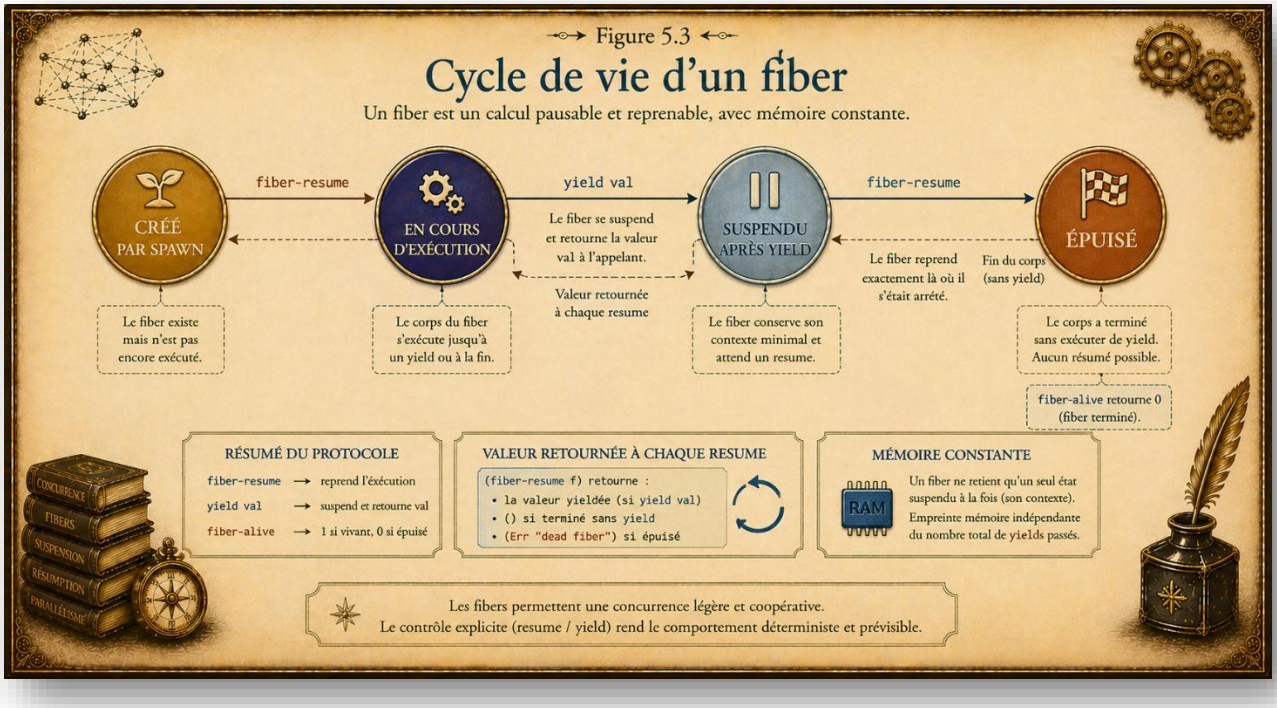


Figure 5.4 — *Le fiber alterne entre exécution et suspension via yield/resume*

Commande	Syntaxe	Description
spawn	[spawn keyword args...]	Instancier un fiber depuis un keyword
yield	[yield val]	Suspendre le fiber et publier une valeur
fiber-resume	[fiber-resume $fib]	Reprendre le fiber, obtenir la prochaine valeur
fiber-alive	[fiber-alive $fib]	Tester si le fiber peut encore produire (1 ou 0)
fiber-to-list	[fiber-to-list $fib]	Collecter toutes les valeurs restantes en liste
stream-pipe	[stream-pipe $f1 $f2]	Connecter la sortie d'un fiber à l'entrée d'un autre

Figure 5.5 — *Les builtins de gestion des fibers*

Producteur–consommateur découplés

La métaphore essentielle pour comprendre les fibers est celle du **canal** entre producteur et consommateur. Le producteur ne calcule rien tant que personne ne lui demande ; le consommateur n'attend qu'à chaque demande, le temps qu'une seule valeur soit produite. Les deux peuvent vivre à des cadences très différentes — un producteur peut être lent (lecture réseau, calcul coûteux) et un consommateur rapide, ou inversement.

Cette propriété est précieuse dans les pipelines neuro-symboliques de production. Un fiber peut par exemple lire un flux Kafka entrant, un autre fiber consommer ces messages au rythme où le LLM peut les traiter, sans que l'on ait à coder explicitement de buffer ni de mécanisme de back-pressure. La

commande `stream-pipe` connecte directement deux fibers, le second consommant les valeurs yieldées par le premier.

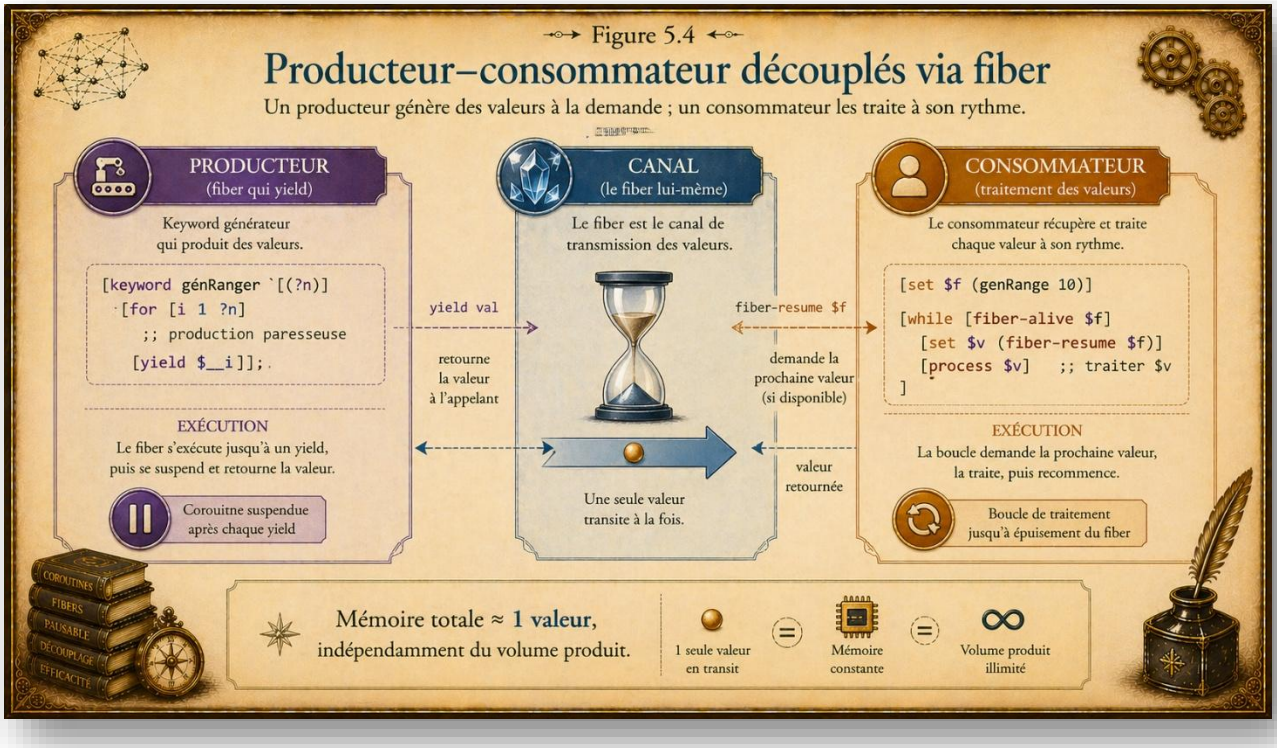


Figure 5.6 — *Découplage temporel et spatial entre producteur et consommateur*

La programmation fonctionnelle de VVL — lambdas, keywords, ordre supérieur, fibers — fournit la palette complète d'outils nécessaires pour construire des pipelines neuro-symboliques propres. Le chapitre 6 ferme cette Partie II par la programmation orientée objet, qui apporte la dernière brique structurelle dont nous aurons besoin avant d'aborder le LLM comme co-processeur du langage à la Partie III.

6. Programmation orientée objet

Mots-clés : class, field, method, $self, $super, new, init, héritage, mixin, MRO, protocol, surcharge d'opérateurs, validation

Ce chapitre clôt la Partie II du livre par la dernière brique syntaxique de VVL : son système orienté objet. Cette présentation tardive — après la programmation fonctionnelle — n'est pas accidentelle. Contrairement aux langages où l'OO est le paradigme dominant qui happe les autres styles, VVL traite l'OO comme un mécanisme parmi d'autres, parfaitement intégré au reste du langage par la règle d'or **« tout reste une liste »**. Une classe est un keyword étendu, une instance est un objet dont le contexte interne est isolé, une méthode est une lambda enveloppée, et l'appel de méthode reste syntaxiquement identique à l'appel de fonction.

La présentation suit l'ordre habituel : philosophie OO (6.1), classes et instances (6.2), héritage avec `$super` (6.3), mixins et MRO (6.4), protocols (6.5), et enfin annotations de validation et surcharge d'opérateurs (6.6). Cette dernière section est particulièrement importante pour la suite du livre : c'est elle qui prépare le mécanisme de dispatch utilisé par les opérateurs Zalta du chapitre 13, où la surcharge d'opérateur devient la passerelle entre la programmation OO classique et la programmation neuro-symbolique.

6.1. Philosophie : tout reste une liste

Le système OO de VVL s'inspire directement de **TclOO** (Tcl 8.6, 2012), qui a réussi à intégrer l'orienté objet dans le noyau de Tcl sans introduire ni nouveau token, ni nouvelle syntaxe de parsing. Le principe fondateur est qu'**une classe est elle-même une liste**, qu'**une instance est un objet doté d'un contexte interne isolé**, et que le **dispatch de méthode** s'appuie sur le mécanisme générique d'évaluation des listes. Ce choix d'architecture a une conséquence importante : on n'a rien à apprendre de nouveau sur la syntaxe pour utiliser l'OO.

Concrètement, `[class Animal ...]` est une commande comme une autre, qui prend un nom et une liste de définitions de champs et méthodes, et qui enregistre la classe dans le contexte d'exécution. `[new Animal "Rex"]` est aussi une commande, qui crée une instance et appelle son constructeur. `[$rex.speak]` n'est qu'une notation pointée vers la méthode, qui se résout dynamiquement par le mécanisme `get_property`. Tout ce mécanisme est purement implémenté en termes de keywords et de listes — il n'y a pas de couche méta-objet séparée.

Cette architecture a quatre conséquences notables. **Premièrement, la cohabitation est trivialisée** : un objet peut être membre d'une liste, passé à `map`, sérialisé en JSON, exporté en Prolog. **Deuxièmement, la méta-programmation est native** : on peut inspecter la définition d'une classe, ses méthodes, ses champs avec les mêmes builtins que pour n'importe quelle structure. **Troisièmement, le dispatch est extensible** : c'est sur ce mécanisme que les opérateurs Zalta s'appuient au chapitre 13, en injectant un mixin dans le MRO. **Quatrièmement, le coût conceptuel est minimal** : un développeur déjà familier de VVL n'a besoin d'apprendre que cinq builtins (`class`, `field`, `method`, `new`, `mixin`) pour utiliser pleinement l'OO.

6.2. Classes, champs et méthodes

La définition d'une classe utilise la commande `class`, suivie du nom puis d'une liste de blocs `[field name default]` et `[method name [params] body]`. Si la classe définit un constructeur — la méthode nommée `init` — celle-ci est automatiquement appelée par `new` avec les arguments fournis. Sans `init`, les champs conservent simplement leurs valeurs par défaut.

```
# Definition d'une classe simple
[class Animal
    [field name "unknown"]
    [field age 0]

    [method init [n a]
        [do
            $self.name = $n
            $self.age = $a
        ]
    ]

    [method speak []
        [log [concat $self.name " a " $self.age " ans"]]
    ]
]

# Instanciation
$rex = [new Animal "Rex" 5]

# Acces aux champs (notation pointee)
[log $rex.name]                          # => Rex
[log $rex.age]                           # => 5

# Appel de methode
[$rex.speak]                               # => Rex a 5 ans

# Mutation d'un champ
$rex.age = 6
[$rex.speak]                               # => Rex a 6 ans
```

La variable spéciale `$self` est automatiquement injectée dans le contexte de chaque méthode et pointe vers l'instance appelante. C'est exactement le rôle du `self` explicite de Python ou du `my` de TclOO. À l'inverse du `this` implicite de Java ou JavaScript, `$self` doit être écrit explicitement — ce qui rend les accès aux champs internes plus visibles et évite toute ambiguïté avec une variable locale du même nom.

La mécanique interne du dispatch suit quatre étapes, qu'il est utile de comprendre pour anticiper le comportement dans les cas tordus : (1) la notation `$instance.propriete` déclenche `get_property` sur l'instance ; (2) `get_property` cherche d'abord dans les champs de données, puis dans les méthodes de la classe via la résolution MRO, puis dans les méta-propriétés (`.class`, `.fields`, `.id`) ; (3) si la propriété est une méthode, un objet `VVL_InstanceMethod` est créé à la volée — un wrapper callable lié à l'instance ; (4) à l'appel, la méthode clone le contexte, injecte `$self` (l'instance) et `$super` (le proxy vers la classe parente), puis exécute la lambda du corps.

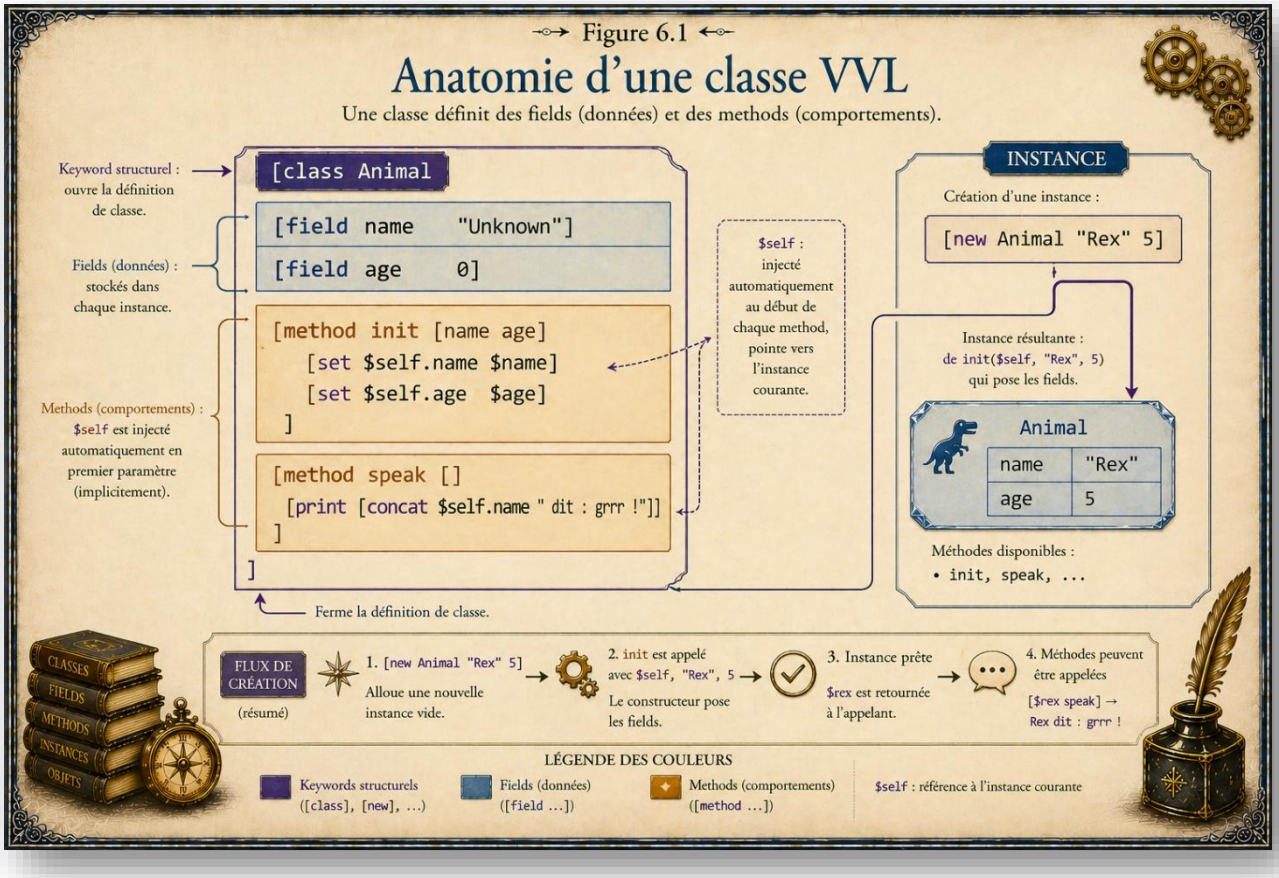


Figure 6.1 — *Définition de classe et chaîne d'instanciation*

6.3. Héritage et $super

L'héritage s'exprime avec la syntaxe `[class Enfant : Parent]`. La classe enfant hérite de tous les champs et méthodes du parent. Les champs hérités peuvent être surchargés en les redéclarant avec une nouvelle valeur par défaut. Les méthodes héritées peuvent être *overridées*, et la variable `$super` permet d'appeler la version parente d'une méthode depuis l'override.

```
[class Animal
    [field name ""]
    [field sound "..."]

    [method init [name]
        $self.name = $name
    ]

    [method speak []
        [log $self.name " dit " $self.sound]
    ]

    [method greet [other]
        [log $self.name " salue " $other.name]
    ]
]

# Sous-classe avec surcharge de champ et de methode
[class Dog : Animal
    [field sound "Woof!"]                   # surcharge du champ

    [method fetch [item]
        [log $self.name " rapporte " $item]
    ]

    [method speak []                        # override avec appel a super
        [do
            [$super.speak]
            [log "(remue la queue)"]
        ]
    ]
]

$rex = [new Dog "Rex"]
$leo = [new Animal "Leo"]
$leo.sound = "Roar"

[$rex.speak]                                # => Rex dit Woof! (remue la queue)
[$leo.speak]                                # => Leo dit Roar
[$rex.greet $leo]                           # => Rex salue Leo  (methode heritee)
[$rex.fetch "balle"]                        # => Rex rapporte balle
```

L'héritage en chaîne (A → B → C) fonctionne naturellement : `$super` dans C appelle la méthode B, et `$super` dans cette méthode B appelle la version A. Chaque niveau de la chaîne voit son propre `$self` (toujours l'instance d'origine) et son propre `$super` (toujours la classe immédiatement parente). Ce mécanisme évite les diamants problématiques de l'héritage multiple — VVL ne supporte qu'un seul parent par classe, et reporte la composition multiple sur les mixins (section 6.4).

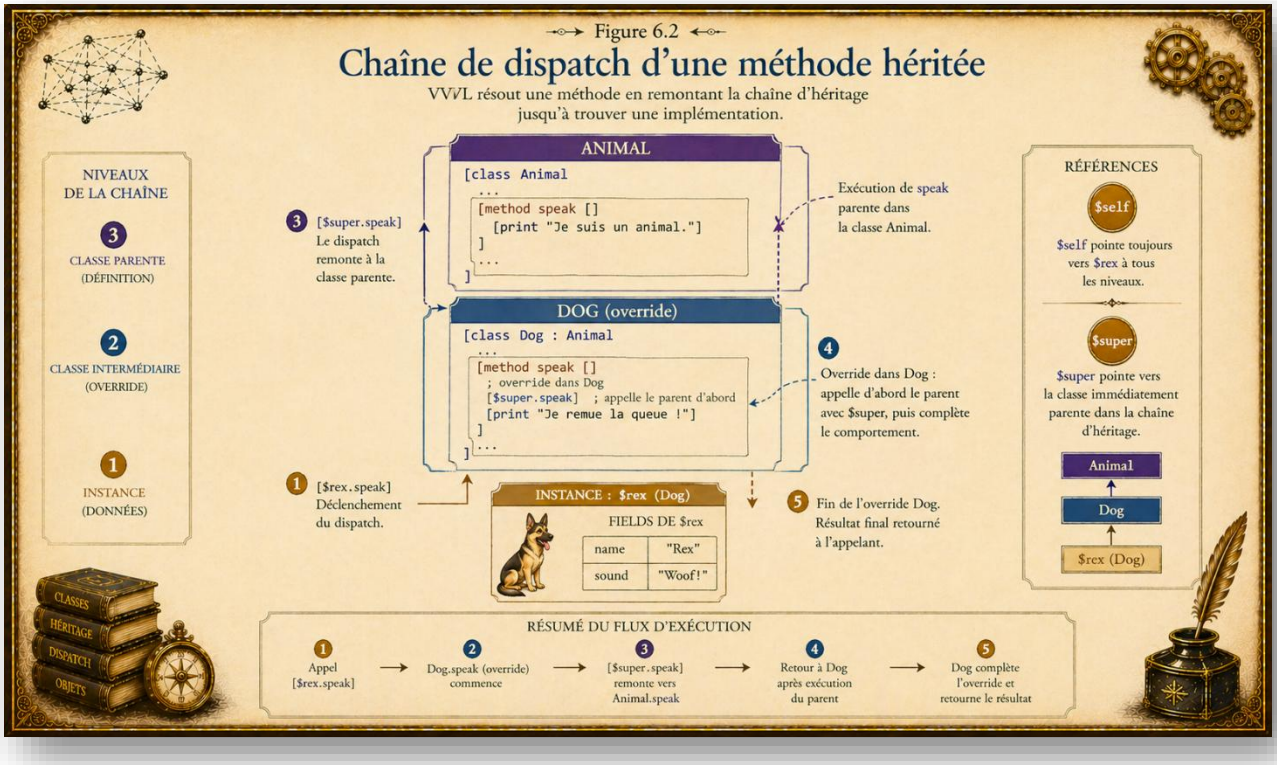


Figure 6.2 — *Override de méthode et appel à $super en cascade*

L'introspection est complète : la commande `[is-a $obj NomClasse]` teste l'appartenance à une classe ou à l'un de ses ancêtres ; `[class-of $obj]` retourne le nom de la classe directe ; `[methods-of $obj]` liste toutes les méthodes accessibles par dispatch.

```
# Introspection
[log [class-of $rex]]                    # => Dog
[log [is-a $rex Dog]]                    # => 1
[log [is-a $rex Animal]]                 # => 1  (heritage)
[log [is-a $rex Cat]]                    # => 0
[log [methods-of $rex]]                  # => [init speak greet fetch]
```

6.4. Mixins et résolution MRO

Les **mixins** permettent d'ajouter des méthodes à une classe **sans créer de relation d'héritage**. Ils sont déclarés avec `[mixin NomClasse]` dans le corps de la classe. C'est l'outil canonique de la composition de comportements : on assemble les capacités voulues (sérialisation, journalisation, datation, comparaison) sans imposer une hiérarchie d'héritage rigide.

La résolution MRO (Method Resolution Order) suit une règle stricte : **méthode propre > mixin (ordre de déclaration) > parent**. Cette règle élimine les ambiguïtés classiques de l'héritage multiple — le

diamant Java, le `super()` ambigu de Python pré-MRO C3 — tout en offrant la souplesse de la composition.

```
# Trois mixins pour des comportements composables
[class Serializable
    [method to-json []
        [log "[JSON] " $self.class " serialise"]
    ]
]

[class Loggable
    [field log_level "INFO"]
    [method log [msg]
        [log "[" $self.log_level "] " $self.class " : " $msg]
    ]
]

[class Timestamped
    [field created_at ""]
    [field updated_at ""]
    [method touch []
        $self.updated_at = "now"
    ]
]

# Classe combinant heritage classique + deux mixins
[class Document : Loggable
    [mixin Serializable]
    [mixin Timestamped]
    [field title ""]

    [method init [t]
        $self.title = $t
    ]
]

$doc = [new Document "Mon Rapport"]
[$doc.to-json]                          # mixin Serializable
[$doc.touch]                            # mixin Timestamped
[$doc.log "Document cree"]              # parent Loggable
[log [is-a $doc Serializable]]          # => 1  (is-a verifie aussi les mixins)
```

L'illustration la plus parlante de la règle MRO est le cas de méthodes homonymes définies à plusieurs niveaux. Dans l'exemple ci-dessous, `MixinRenderer` et `BaseWidget` définissent tous deux la méthode `render`. Selon que la classe finale définit ou non sa propre version, le résultat diffère :

```
[class MixinRenderer
    [method render [] "mixin-render"]
]

[class BaseWidget
    [method render [] "base-render"]
]

# Cas 1 : Widget definit sa propre methode -> elle gagne
[class Widget : BaseWidget
    [mixin MixinRenderer]
    [method render [] "widget-render"]
]
```

```
14
15  $w = [new Widget]
16  [log [$w.render]]                          # => widget-render  (propre > mixin > parent)
17
18  # Cas 2 : Button n'a pas de render propre -> mixin gagne sur parent
19  [class Button : BaseWidget
20      [mixin MixinRenderer]
21      [field label "Click"]
22  ]
23
24  $b = [new Button]
25  [log [$b.render]]                          # => mixin-render  (mixin > parent)
```

Niveau	Priorité	Cas de Widget	Cas de Button
Méthode propre	1er (le plus fort)	render = "widget-render"	— absent —
Mixin (déclaration)	2e	render = "mixin-render"	render = "mixin-render"
Parent	3e	render = "base-render"	render = "base-render"
Résultat	—	"widget-render"	"mixin-render"

Figure 6.3 — *Règle MRO de VVL : propre > mixin > parent*

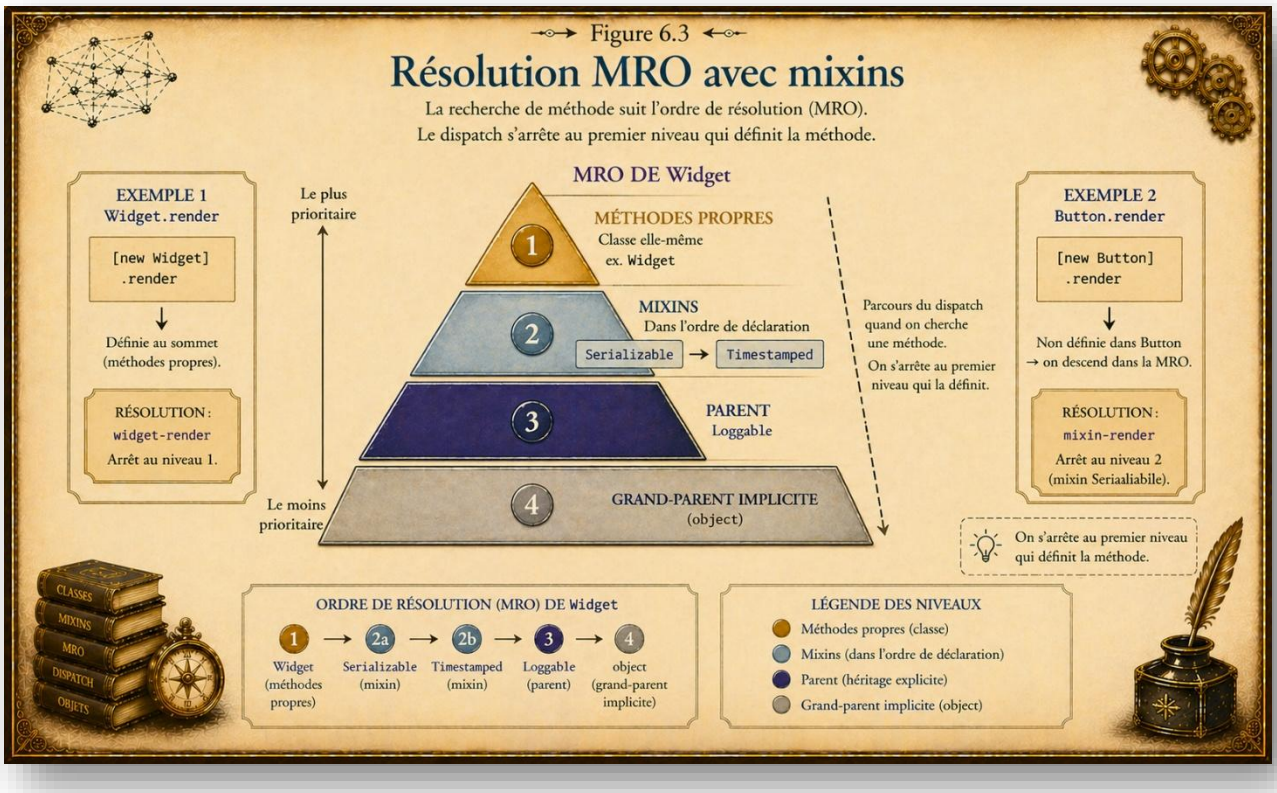


Figure 6.4 — *La règle propre > mixin > parent appliquée à la résolution de méthodes*

6.5. Protocols : interfaces contractuelles

Un **protocol** définit un contrat que les classes s'engagent à respecter. Il déclare une liste de méthodes requises avec `[requires nom]`. Les classes déclarent leur conformité avec `[implements NomProtocol]` dans leur corps. À l'instanciation, le moteur vérifie que toutes les méthodes requises sont effectivement définies — sinon une erreur est levée.

Les protocols sont l'équivalent VVL des *interfaces* de Java, des *protocols* de Swift ou des *traits* de Rust : ils découplent la spécification du contrat de son implémentation. À la différence d'un parent abstrait, un protocol n'apporte aucun champ ni méthode héritée — il ne fait qu'imposer des obligations. Une même classe peut implémenter plusieurs protocols, ce qui permet de classer les objets par capacité plutôt que par hiérarchie.

```
[protocol Logable
    [requires to-string]
]

[protocol Comparable
    [requires compare-to]
]

[class Product
    [implements Logable]
    [implements Comparable]

    [field name ""]
    [field price 0]

    [method init [n p]
        [do $self.name = $n $self.price = $p]
    ]

    [method to-string []
        $self.name
    ]

    [method compare-to [other]
        ($self.price - $other.price)
    ]
]

$p1 = [new Product "Laptop" 999]
$p2 = [new Product "Mouse"  29]

[log [$p1.to-string]]                    # => Laptop
[log [$p1.compare-to $p2]]               # => 970

# Verification de conformite
[log [is-a $p1 Logable]]               # => 1
[log [implements-of $p1 Comparable]]     # => 1
```

6.6. Annotations de validation et surcharge d'opérateurs

Annotations de validation

VVL enrichit la déclaration de champs avec des **annotations de validation** héritées du système ontologique du chapitre 12. Ces annotations sont vérifiées automatiquement à chaque assignation — pas seulement à la création. C'est un apport unique de VVL : les contraintes métier sont déclarées au niveau de la classe et appliquées systématiquement, sans qu'aucun code de vérification explicite ne soit nécessaire dans les setters.

```
# Classe avec annotations de validation
[class Product
    [field name "" :required]
    [field price 0 :min 0 :max 99999]
    [field email "" :type Email]
    [field qty 1 :min 1 :type Integer]
    [field status "" :enum [active inactive draft]]
]

$p = [new Product]
$p.name   = "Widget"                    # OK
$p.price  = 42                          # OK
$p.price  = -5                          # ERREUR : price (-5) < min (0)
$p.email  = "bad"                       # ERREUR : Invalid Email format
$p.status = "maybe"                     # ERREUR : valeur hors enum
```

Annotation	Syntaxe	Description
:required	[field name "" :required]	Le champ ne peut pas être vide
:min	[field price 0 :min 0]	Valeur numérique minimale
:max	[field score 0 :max 100]	Valeur numérique maximale
:type	[field email "" :type Email]	Validation de format (Email, Integer, Float, Currency)
:enum	[field status "" :enum [a b c]]	Valeur restreinte à un ensemble
:embed	[field lines [] :embed]	Composition inline (persistance ontologique)
:ref	[field customer "" :ref]	Référence externe par UUID (persistance)

Figure 6.5 — *Les annotations de validation des champs*

Surcharge d'opérateurs

VVL supporte la **surcharge des opérateurs arithmétiques et de comparaison** sur les classes. Trois syntaxes de nommage sont acceptées pour les méthodes d'opérateur : *quoted* (`"+"`), *style VVL* (`op-add`) et *style Python* (`__add__`). Lorsque le moteur évalue `($v1 + $v2)`, il détecte que `$v1` est une instance et appelle la méthode `"+"` de sa classe — exactement comme Python invoque `__add__`.

```
[class Vector
    [field x 0] [field y 0]
    [method init [x y]
        [do $self.x = $x $self.y = $y]]

    [method "+" [other]
        [new Vector ($self.x + $other.x) ($self.y + $other.y)]]

    [method "-" [other]
        [new Vector ($self.x - $other.x) ($self.y - $other.y)]]

    [method "==" [other]
        (($self.x == $other.x) && ($self.y == $other.y)) ]

    [method to-string []
        [concat "(" $self.x ", " $self.y ")"]]
]
$v1 = [new Vector 3 4]
$v2 = [new Vector 1 2]
$v3 = ($v1 + $v2)                       # appelle la methode "+"
[log [$v3.to-string]]                   # => (4, 6)
[log ($v1 == [new Vector 3 4])]         # => 1
```

Figure 6.6 — *Trois conventions de nommage pour les méthodes d'opérateur*

Opérateur	Style quoted	Style VVL	Style Python
+	"+"	op-add	__add__
-	"-"	op-sub	__sub__
*	"*"	op-mul	__mul__
==	"=="	op-eq	__eq__
>	">"	op-gt	__gt__
<	"<"	op-lt	__lt__

Cette mécanique de surcharge n'est pas un détail d'élégance syntaxique : c'est **le pont architectural qui prépare le système d'opérateurs Zalta** du chapitre 13. Les huit opérateurs Zalta (`==`, `~~`, `*`, `**`, `+`, `++`, `/`, `//`) sont implémentés comme un **mixin** nommé `ZaltaOperators`, injecté automatiquement dans les classes ontologiques et les concepts réifiés. Lorsque l'on évalue `($soleil ~~ "Etoile")`, le moteur applique exactement la même règle MRO décrite à la section 6.4 : il cherche d'abord une éventuelle surcharge utilisateur de `op-eq`, puis le mixin `ZaltaOperators`, puis l'héritage classique.

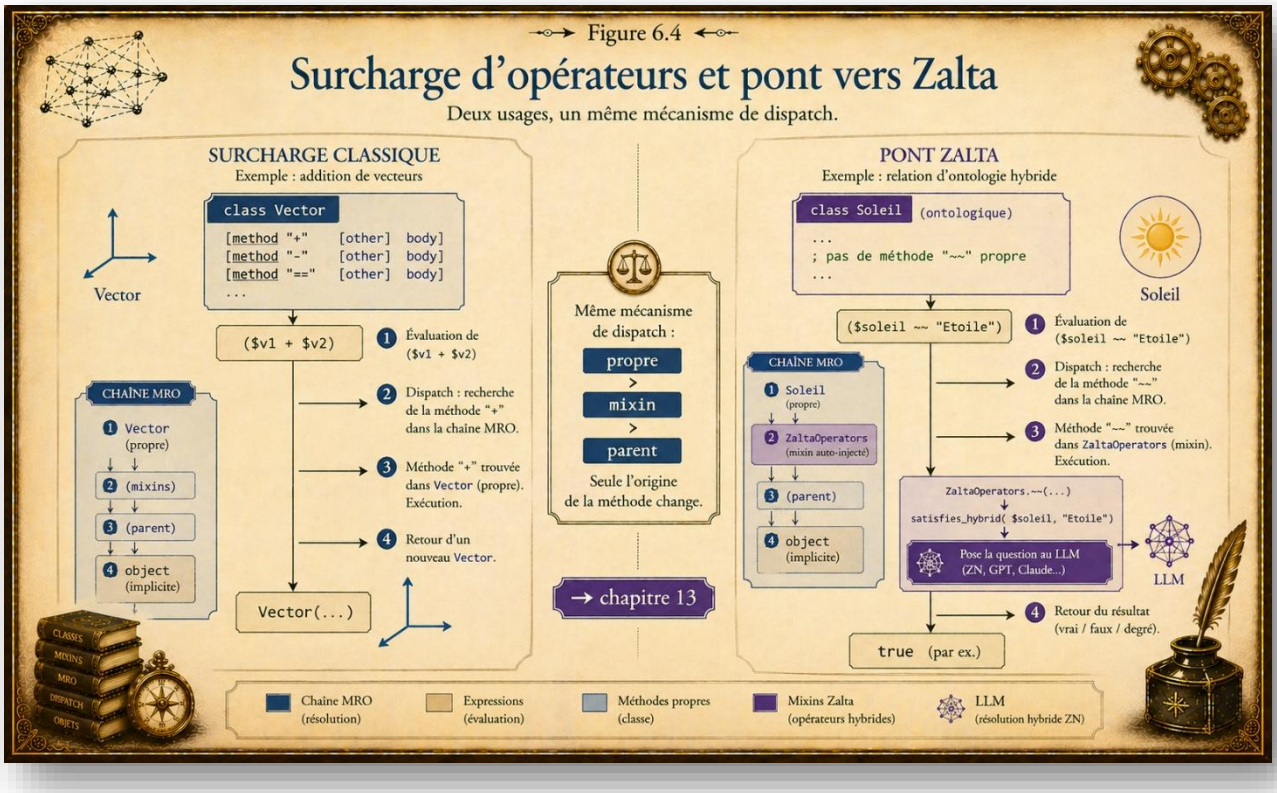


Figure 6.7 — *La surcharge d'opérateurs est le mécanisme qui rend possibles les opérateurs Zalta*

Avec ce chapitre, la Partie II du livre est complète. Le lecteur dispose de tous les outils syntaxiques de VVL : commande entre crochets, types primitifs, listes/JSON/XML natifs, structures de contrôle et pattern matching algébrique, lambdas et fonctions d'ordre supérieur, fibers et générateurs, classes et héritage avec mixins. La Partie III, qui s'ouvre au chapitre 7, va maintenant exposer la spécificité de

VVL : le LLM intégré comme co-processeur du langage, et les vingt opérateurs sémantiques qui transforment chaque expression VVL en une porte d'entrée potentielle vers le raisonnement neuronal.

Partie III — Le LLM comme co-processeur

Chapitres 7 à 10 — opérateurs sémantiques, génération de code, réification, §§

7. Les opérateurs sémantiques

Mots-clés : VVL_Semantic, opérateur sémantique, dispatch, comparaison floue, algèbre conceptuelle, projection, propriété, extraction QA, cache déterministe, spécification XML

Le chapitre 7 ouvre la Partie III du livre, qui constitue le cœur conceptuel de cet ouvrage : le LLM intégré comme co-processeur du langage. Là où les Parties I et II ont posé respectivement le contexte théorique et la grammaire du langage, cette partie expose la spécificité radicale de VVL — la transformation du LLM, système probabiliste imprévisible, en un ensemble d'**opérateurs du langage** aussi fiables que `+` ou `==`.

Ce chapitre couvre les **vingt opérateurs sémantiques** que VVL met à disposition. La première section introduit le mécanisme général — le type `VVL_Semantic` et le dispatch automatique vers le moteur LLM (7.1). Les quatre sections suivantes parcourent les catégories d'opérateurs : comparaisons floues (7.2), algèbre conceptuelle et projection (7.3), extraction QA et logique sémantique (7.4), appartenance et inclusion (7.5). La section 7.6 explique les quatre garanties — typage, cache, retry, spécifications XML — qui rendent ce système reproductible. La section 7.7 fournit la référence complète. L'opérateur `@`, à part, fait l'objet du chapitre 8 entier.

7.1. Le type VVL_Semantic et la syntaxe [§ "..."]

Le type `VVL_Semantic` est la **porte d'entrée** vers le monde neuro-symbolique. Il est créé avec la syntaxe `[§ "texte"]` et encapsule une valeur textuelle qui sera interprétée **sémantiquement** — par le sens, pas par la syntaxe — lorsqu'elle participe à une opération. Le caractère `§` (section, U+00A7) a été choisi pour sa rareté typographique : il ne risque jamais de collision avec une syntaxe utilisateur.

La mécanique est élégante : dès qu'un opérateur binaire détecte qu'**au moins un de ses opérandes est de type `VVL_Semantic`**, il abandonne sa sémantique arithmétique ou logique standard. À la place, il consulte la spécification de l'opérateur dans le fichier `neuro-symbolic.xml`, construit un prompt enrichi des exemples few-shot définis pour cet opérateur, envoie le tout au LLM, et convertit la réponse dans le type de sortie spécifié (`BOOLEAN`, `STRING`, `SEMANTIC` ou `LAMBDA`).

```
# Creer une valeur semantique
$concept = [§ "voiture electrique"]

# Les operateurs deviennent semantiques automatiquement
[log ([§ "chat"] == [§ "felin"])]         # => 1  (synonymes)
[log ([§ "chaud"] != [§ "froid"])]        # => 1  (antonymes detectes)
[log ([§ "Paris"] % [§ "pays"])]          # => France

# Melange avec des valeurs classiques
$texte = [§ "La tour Eiffel mesure 330 metres."]
$hauteur = ($texte ? [§ "Quelle est sa hauteur ?"])
[log $hauteur]                            # => 330 metres
```

Le contraste avec un usage classique du LLM est radical. Sans `VVL_Semantic`, comparer deux concepts en équivalence imposerait de construire manuellement un prompt (« GPT, dis-moi si X est synonyme de Y, réponds par 1 ou 0 »), de gérer le retry en cas de réponse mal formée, de parser le résultat, de gérer les exceptions. En VVL, tout cela disparaît derrière un simple `==`. La syntaxe ne change pas — c'est le **dispatch** qui change selon le type.

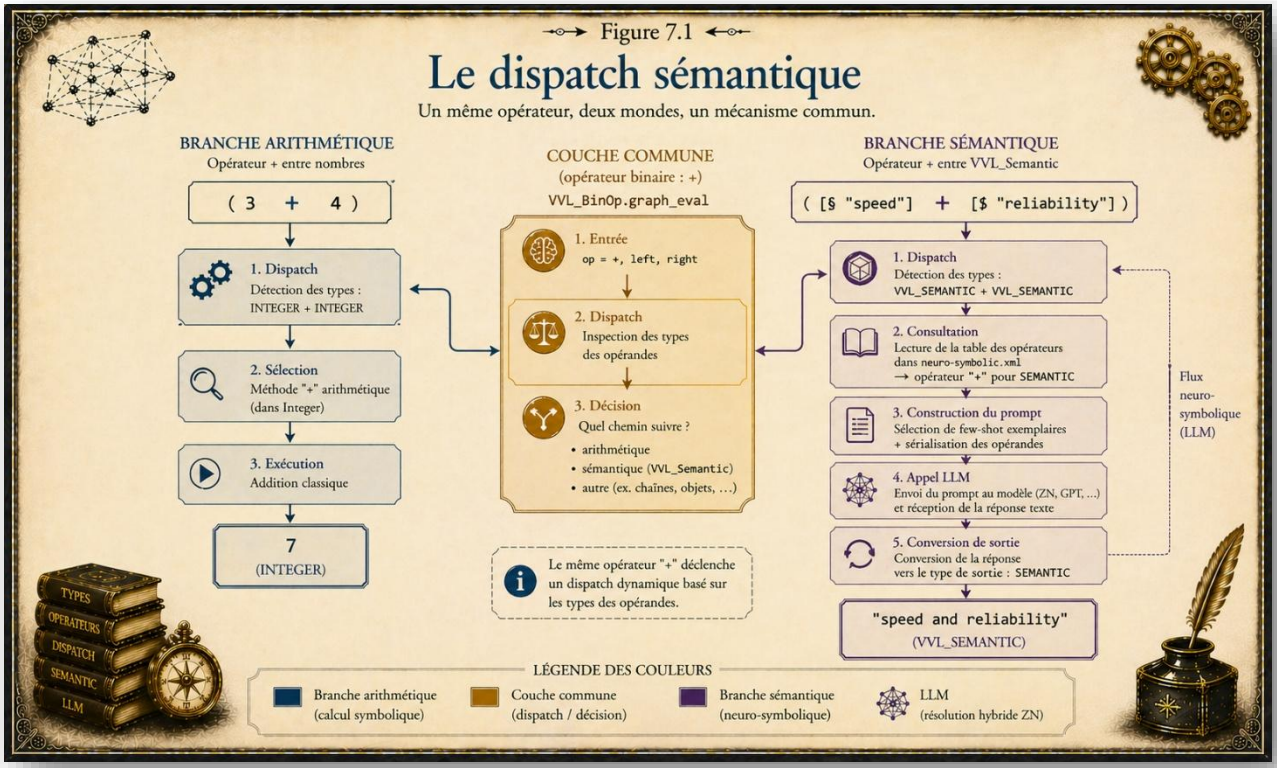


Figure 7.1 — *Le même symbole, deux sémantiques selon le type des opérandes*

La syntaxe `[§ ...]` est en fait polyvalente. Elle accepte une chaîne de caractères pour créer un concept textuel ; elle accepte aussi une image (`[§ $img]`) pour produire une description sémantique via le modèle vision ; et elle accepte une description suivie du mot sentinelle `IMG` pour générer une image via DALL·E (`[§ "un chat futuriste" IMG "1024x1024"]`). Cette polyvalence est cohérente avec la philosophie générale : un point d'entrée unique vers les capacités sémantiques du LLM, qu'elles concernent le texte ou l'image.

7.2. Comparaisons floues : ==, !=, <, >, <=, >=

Les opérateurs `==` et `!=` illustrent parfaitement la transformation probabiliste → déterministe. Dans un programme classique, comparer deux chaînes exige une identité syntaxique parfaite : `"chat" != "félin"`, `"5" != "five"`, `"2024-08-04" != "4 août 2024"`. La comparaison floue de VVL teste l'**équivalence conceptuelle** : synonymie, traduction, format alternatif, voire tolérance orthographique.

```
1  # Synonymes detectes
2  [log ([§ "chat"] == [§ "felin"])]         # => 1
```

```
3
4  # Traduction cross-language
5  [log ([§ "seven"] == [§ "Sieben"])]       # => 1
6
7  # Formats de date equivalents
8  [log ([§ "August 4, 1961"] == [§ "1961-08-04"])]  # => 1
9
10 # Tolerance orthographique
11 [log ([§ "we hav teh systeem"] == [§ "We have the system"])]  # => 1
12
13 # Faux positifs evites
14 [log ([§ "Hello World"] == [§ "Apples"])]  # => 0
15
16 # Inegalite : antonymes
17 [log ([§ "chaud"] != [§ "froid"])]         # => 1
```

Les opérateurs ordinaux (<, >, <=, >=) sont encore plus surprenants : ils comparent deux concepts sur leur **dimension naturelle implicite**. Le LLM choisit la dimension contextuellement appropriée — taille pour des objets physiques, séniorité pour des intitulés de poste, intensité pour des adjectifs.

```
1  # Comparaison sur la taille
2  [log ([§ "minuscule"] < [§ "enorme"])]    # => 1
3
4  # Comparaison sur la seniorite professionnelle
5  [log ([§ "junior"] < [§ "senior"])]       # => 1
6  [log ([§ "stagiaire"] < [§ "directeur"])] # => 1
7
8  # Comparaison sur l'intensite
9  [log ([§ "tiede"] < [§ "brulant"])]       # => 1
10
11 # Comparaison numerique sur des descriptions textuelles
12 [log ([§ "petit"] <= [§ "petit"])]        # => 1  (identite)
```

7.3. Algèbre conceptuelle : +, -, *, /, %

Les opérateurs arithmétiques deviennent, dans le contexte sémantique, des opérations sur les **concepts**. L'addition fusionne, la soustraction retire, la multiplication intersecte, la division projette. Le résultat est de type `VVL_Semantic` (sauf pour `/` et `%` qui retournent une chaîne), donc directement composable avec d'autres opérateurs sémantiques.

```
1  # Fusion (+) : assemblage de concepts
2  [log ([§ "voiture"] + [§ "electrique"])]              # => voiture electrique
3  [log ([§ "intelligence"] + [§ "artificielle"])]       # => intelligence artificielle
4
5  # Difference (-) : retrait conceptuel
6  [log ([§ "chat noir"] - [§ "noir"])]                  # => chat
7  [log ([§ "voiture sport rouge"] - [§ "rouge"])]       # => voiture sport
8
9  # Intersection (*) : ce qui est commun aux deux
10 [log ([§ "voiture electrique"] * [§ "bus electrique"])]  # => vehicule electrique
11 [log ([§ "medecin"] * [§ "enseignant"])]                  # => educateur medical
12
13 # Projection (/) : extraction d'une dimension d'une description
14 [log ([§ "voiture rouge a Paris"] / [§ "couleur"])]   # => rouge
15 [log ([§ "ingenieur senior"] / [§ "niveau"])]         # => senior
16
17 # Propriete (%) : interrogation sur un attribut d'une entite
```

```
[log ([§ "Paris"] % [§ "pays"])]                    # => France
[log ([§ "Python"] % [§ "categorie"])]              # => langage de programmation
```

La distinction entre `/` (projection) et `%` (propriété) mérite qu'on s'y arrête. **La projection** opère sur une description complexe pour en extraire une dimension donnée : `[§ "voiture rouge à Paris"] / [§ "couleur"]` retourne `"rouge"`. La description contient déjà l'information ; l'opérateur l'extrait. **La propriété** interroge sur un attribut d'une entité connue : `[§ "Paris"] % [§ "pays"]` retourne `"France"`. L'information n'est pas dans l'opérande gauche ; elle est dans la connaissance générale du LLM. Cette distinction permet de coder finement le type d'opération attendue.

La composabilité de ces opérateurs est cruciale. Comme le résultat de `+`, `-` ou `*` est lui-même un `VVL_Semantic`, on peut chaîner des opérations conceptuelles aussi naturellement qu'on chaînerait des opérations arithmétiques :

```
# Composition d'operateurs semantiques
$concept = (([§ "intelligence artificielle"] + [§ "explicable"])
            - [§ "boite noire"])
[log $concept]                          # => intelligence artificielle transparente

# Combinaison avec comparaison
$proche = (([§ "voiture electrique"] * [§ "Tesla"]) == [§ "vehicule premium"])
[log $proche]                           # => 1
```

7.4. Extraction QA et composition logique : ?, &&, ||, ^^, ~

Extraction question-réponse (?)

L'opérateur `?` permet d'**interroger un texte** pour en extraire une information précise. Il est l'équivalent sémantique de `getValue` ou `jsonGetValue`, mais où la clé est une question en langage naturel et la source un texte non structuré. C'est l'outil canonique pour transformer du texte en données exploitables.

```
# Extraction d'une donnee precise depuis un texte libre
$bio = [§ "Alan Turing est ne en 1912 a Londres et a travaille a Bletchley Park."]
[log ($bio ? [§ "Annee de naissance ?"])]   # => 1912
[log ($bio ? [§ "Lieu de naissance ?"])]    # => Londres
[log ($bio ? [§ "Ou a-t-il travaille ?"])]  # => Bletchley Park

# Sur une description technique
$specs = [§ "Le serveur tourne sur 16 coeurs avec 64 Go de RAM et 2 To SSD."]
[log ($specs ? [§ "Combien de coeurs ?"])]  # => 16
[log ($specs ? [§ "Stockage total ?"])]     # => 2 To
```

Composition logique (&&, ||, ^^, ~)

Les opérateurs logiques produisent des **expressions conceptuelles composées**. Ils ne retournent pas un booléen comme dans le cas classique, mais un `VVL_Semantic` qui exprime la conjonction, la disjonction, l'exclusion ou la négation des concepts opérandes. Ces résultats sont conçus pour être réutilisés dans d'autres expressions sémantiques.

```
# Conjonction (&&) : fusion par ET
[log ([§ "vitesse"] && [§ "fiabilite"])]      # => vitesse et fiabilite

# Disjonction (||) : alternative par OU
[log ([§ "the"] || [§ "cafe"])]               # => the ou cafe
[log ([§ "remote"] || [§ "presentiel"])]      # => remote ou presentiel

# XOR (^^) : exclusivite
[log ([§ "jour"] ^^ [§ "nuit"])]              # => jour ou nuit (exclusif)
[log ([§ "online"] ^^ [§ "offline"])]         # => online ou offline (exclusif)

# Negation (~) : operateur unaire
[log ~[§ "Je suis content"]]                   # => Je ne suis pas content
[log ~[§ "L'acces est autorise"]]              # => L'acces n'est pas autorise
```

7.5. Appartenance et inclusion : in, <|, |>

Les trois derniers opérateurs sémantiques testent les **relations conceptuelles d'appartenance**. Ils sont fondamentaux pour la programmation neuro-symbolique parce qu'ils encodent la relation IS-A — la pierre angulaire de toute classification ontologique — directement comme une opération du langage.

```
# in : appartenance a une categorie
[log ([§ "chat"] in [§ "animaux"])]                # => 1
[log ([§ "Paris"] in [§ "capitales europeennes"])] # => 1
[log ([§ "concombre"] in [§ "fruits"])]            # => 0
[log ([§ "Python"] in [§ "langages de programmation"])]  # => 1

# <| : sous-concept (le gauche est un cas particulier du droit)
[log ([§ "chat noir"] <| [§ "animaux"])]           # => 1
[log ([§ "voiture"] <| [§ "fruits"])]              # => 0

# |> : super-concept (le gauche contient le droit)
[log ([§ "animaux"] |> [§ "chat noir"])]           # => 1
[log ([§ "fruits"] |> [§ "voiture"])]              # => 0
```

La distinction entre `in`, `<|` et `|>` est subtile mais importante. **`in`** teste si un *individu* est membre d'une catégorie (« le chat est un animal »). **`<|`** teste si un *concept* est subsumé par un autre (« la notion de chat noir est subsumée par la notion d'animal »). **`|>`** est la relation inverse de `<|`. Dans la pratique, `in` et `<|` se confondent souvent, mais ils diffèrent quand le concept gauche est lui-même intensionnel : `[§ "voiture électrique"] <| [§ "véhicule"]` est plus naturel que `[§ "voiture électrique"] in [§ "véhicule"]`, qui supposerait qu'on traite *voiture électrique* comme un individu.

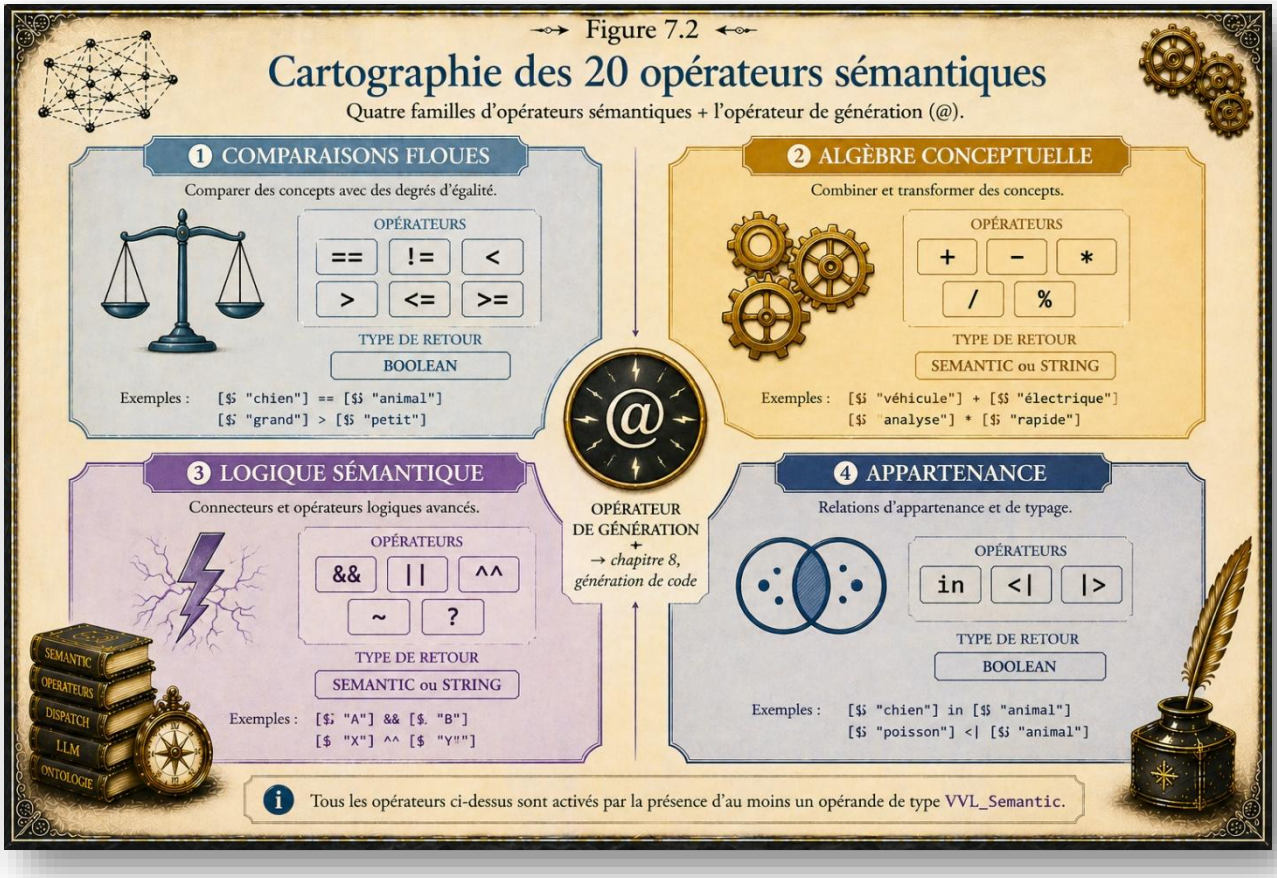


Figure 7.2 — *Les vingt opérateurs sémantiques classés par famille fonctionnelle*

7.6. Garanties de type, cache déterministe et retry automatique

Comment un système probabiliste comme un LLM peut-il fournir le comportement reproductible qu'on attend d'un opérateur de langage ? La réponse de VVL repose sur **quatre garanties** qui transforment l'imprévisibilité brute du LLM en un contrat d'interface stable. Ces garanties sont implémentées dans le moteur d'évaluation, pas laissées à la charge du développeur.

Garantie 1 — Typage de sortie

Chaque opérateur sémantique a un type de sortie spécifié dans la spécification XML. Ce type détermine la conversion automatique appliquée à la réponse du LLM avant qu'elle ne réintègre le programme. Le développeur n'a jamais à parser une chaîne ni à gérer un cas où le LLM aurait répondu autre chose qu'attendu.

OUTPUT-TYPE	Conversion VVL	Opérateurs concernés	Garantie
BOOLEAN	VVL_Num (1 ou 0)	==, !=, <, >, <=, >=, in, <|, |>	Utilisable dans if, when, filter
STRING	VVL_String	%, /, ?	Exploitable par mid, split, concat
SEMANTIC	VVL_Semantic	+, -, *, &&, ||, ^^, ~	Composable avec autres opérateurs sémantiques

OUTPUT-TYPE	Conversion VVL	Opérateurs concernés	Garantie
LAMBDA	AST VVL parsée	@	Exécutable par filter, map, ou appel direct

Figure 7.3 — *Types de sortie garantis par opérateur*

Garantie 2 — Cache déterministe persistant

Chaque appel sémantique est mis en cache par hash de l'expression complète (opérateur + opérandes + spec). Un même appel produit le même résultat dans toutes les sessions, transformant un système probabiliste en composant reproductible. Le cache est persistant sur disque, donc une expression déjà rencontrée dans une session antérieure retournera son résultat sans aucun appel LLM. Cela a deux conséquences pratiques : le coût API d'un programme tend asymptotiquement vers zéro à mesure qu'il s'exécute, et les tests automatisés deviennent reproductibles.

Cette garantie appelle une précision que les versions récentes du runtime ont rendue explicite. Un résultat mis en cache n'est reproductible que relativement au prompt et au modèle qui l'ont produit : éditer la spécification XML d'un opérateur ou changer de modèle sous-jacent modifie la fonction calculée, et une valeur mémorisée sous l'ancienne configuration ne décrit plus le comportement courant. Le runtime enregistre donc, à côté de chaque entrée, l'empreinte du prompt effectif — préambule, instruction, exemples, gabarit et type de sortie — et l'identité du modèle qui a réellement répondu, repli compris. À la lecture, une entrée produite sous une autre empreinte ou un autre modèle est traitée comme un défaut de cache et recalculée. L'invalidation est paresseuse et par opérateur : les entrées antérieures sont recalculées une fois puis estampillées, sans purge globale. Sans ce mécanisme, la trace d'exécution d'un programme pouvait attester une version de l'opérateur qui n'avait jamais produit le résultat affiché — ce qui ruinait précisément l'auditabilité que le cache est censé servir.

Garantie 3 — Retry automatique avec validation

Pour les opérateurs dont la sortie doit respecter une structure (notamment `@` qui produit une lambda VVL), le moteur valide systématiquement la réponse du LLM. Si la validation échoue — par exemple, le LLM a produit une lambda syntaxiquement incorrecte — VVL relance avec un message d'erreur enrichi, jusqu'à un nombre configurable de tentatives. Le développeur n'écrit jamais de boucle de retry ; elle est intégrée à l'opérateur.

Garantie 4 — Spécifications XML avec few-shot

Chaque opérateur est défini par un fragment XML dans `neuro-symbolic.xml` qui inclut quatre éléments critiques :

Élément XML	Rôle	Impact sur la fiabilité
<INSTRUCTION>	Prompt système envoyé au LLM	Cadre la réponse attendue (format, contraintes)
<EXAMPLES>	Exemples few-shot (entrée → sortie)	Aligne le LLM sur le format exact
<QUERY>	Template avec @1, @2 remplacés par les opérandes	Garantit l'uniformité des appels
<OUTPUTTYPE>	Type de retour attendu	Conversion automatique vers le type VVL

Figure 7.4 — *Anatomie d'une spécification XML d'opérateur sémantique*

Figure 7.3

Anatomie d'une spécification XML

Structure d'un opérateur dans neuro-symbolic.xml

neuro-symbolic.xml

```
<OPERATOR symbol="==" output="BOOLEAN">
```

1 <INSTRUCTION>

Prompt système : cadre et consignes générales.

```
Vous êtes un moteur d'inférence neuro-symbolique.
Répondez VRAI ou FAUX.
Basez-vous sur la connaissance générale,
le bon sens et les relations conceptuelles.
Soyez concis et justifiez brièvement.
...
```

```
</INSTRUCTION>
```

2 <EXAMPLES>

Exemples few-shot (entrée → sortie attendue).

#	ENTRÉE (@1, @2)	SORTIE ATTENDUE
1	"dog", "canine"	TRUE
2	"cat", "feline"	TRUE
3	"dog", "cat"	FALSE
4	"hot", "cold"	FALSE
5	"computer", "PC"	TRUE
...		

```
</EXAMPLES>
```

3 <QUERY>

Template de requête utilisateur.
Les placeholders seront remplacés par les opérandes.

```
@1 et @2 sont-ils des synonymes
(ou la même chose) ?
Répondez uniquement VRAI ou FAUX.
```

```
</QUERY>
```

4 <OUTPUTTYPE>

Type de sortie déclaré pour cet opérateur.

```
BOOLEAN
```

```
</OUTPUTTYPE>
```

```
</OPERATOR>
```

UTILISATION PAR LE MOTEUR
(neuro-symbolic engine)

MOTEUR
NEURO-SYMBOLIC

1 INSTRUCTION + EXAMPLES
↓
CONSTRUCTION DU PROMPT SYSTÈME
Cadre + consignes + exemples (few-shot)

2 EXAMPLES
↓
INTÉGRÉS DANS LE PROMPT SYSTÈME
Apprentissage par l'exemple (few-shot)

3 QUERY
↓
CONSTRUCTION DU PROMPT UTILISATEUR
@1 et @2 remplacés par les opérandes au runtime

4 OUTPUTTYPE
↓
CONVERSION AUTOMATIQUE DU RÉSULTAT
Vers le type déclaré (ici BOOLEAN)

RÉSUMÉ DU FLUX

INSTRUCTION + EXAMPLES → PROMPT SYSTÈME | QUERY → PROMPT UTILISATEUR | LLM (ZN, GPT, ...) | OUTPUTTYPE → RÉSULTAT TYPÉ

Chaque opérateur est défini de façon déclarative.
Le moteur assemble le prompt et gère la conversion.

Figure 7.5 — *La spec XML : quatre éléments qui transforment le LLM en opérateur reproductible*

Ces quatre garanties forment système. Sans le typage, le développeur devrait parser. Sans le cache, le coût et la latence seraient prohibitifs. Sans le retry, la moindre variation du modèle casserait le programme. Sans les spécifications XML, chaque appel serait sujet à la dérive du modèle. **C'est l'ensemble de ces quatre mécanismes** qui transforme le LLM en un opérateur du langage utilisable en production, et c'est cette ingénierie qui distingue VVL des bibliothèques d'orchestration LLM qui se contentent de wrapper l'API.

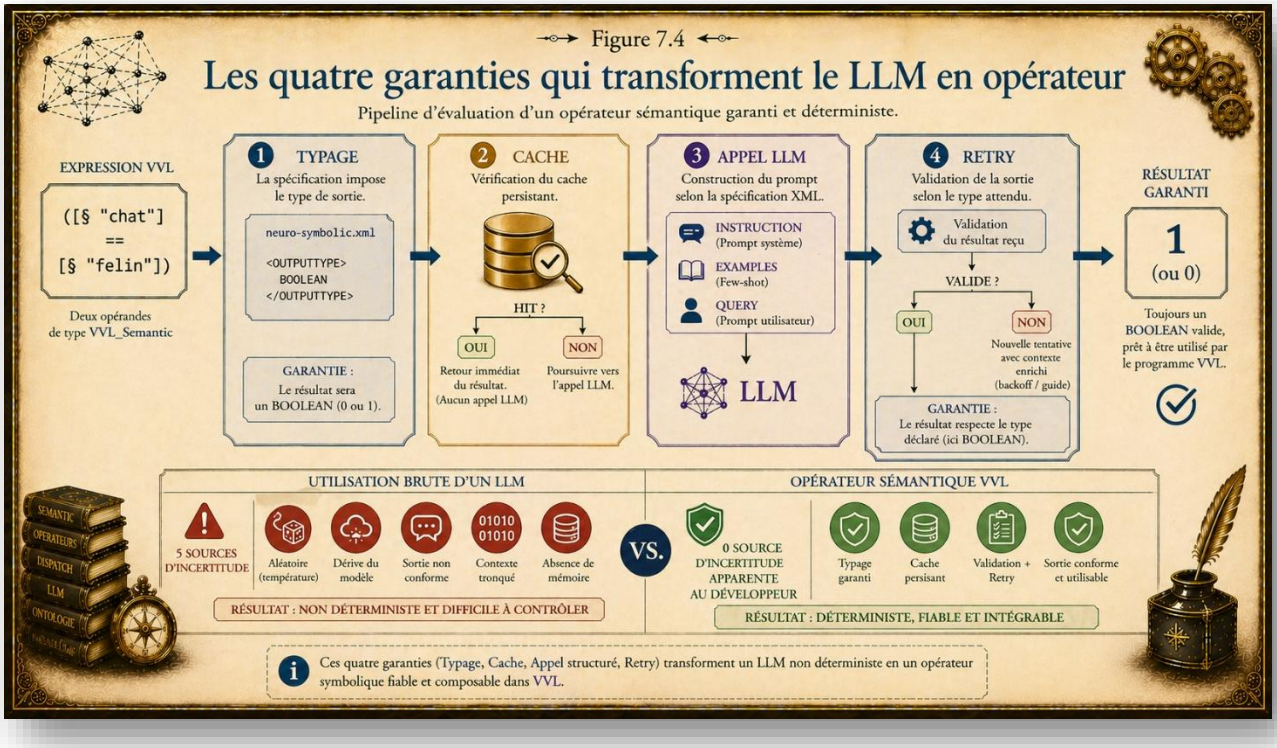


Figure 7.6 — *De l'imprévisibilité brute du LLM à la fiabilité d'un opérateur*

7.7. Référence des 20 opérateurs sémantiques

Le tableau suivant fournit la référence complète des opérateurs sémantiques de VVL. Cette référence est le complément naturel des sections précédentes — on s'y reportera pour la syntaxe exacte d'un opérateur, son type de retour, et un exemple canonique.

Symbole	Nom	Sortie	Exemple	Résultat
==	Égalité floue	BOOLEAN	[§ "chat"] == [§ "felin"]	1
!=	Différence	BOOLEAN	[§ "chaud"] != [§ "froid"]	1
<	Inférieur	BOOLEAN	[§ "tiny"] < [§ "enormous"]	1
>	Supérieur	BOOLEAN	[§ "senior"] > [§ "junior"]	1
<=	Inf. ou égal	BOOLEAN	[§ "small"] <= [§ "small"]	1
>=	Sup. ou égal	BOOLEAN	[§ "large"] >= [§ "small"]	1
+	Fusion	SEMANTIC	[§ "electric"] + [§ "car"]	electric car
-	Différence sém.	SEMANTIC	[§ "black cat"] - [§ "black"]	cat

Symbole	Nom	Sortie	Exemple	Résultat
*	Intersection	SEMANTIC	[§ "doctor"] * [§ "teacher"]	medical educator
/	Projection	STRING	[§ "red car in Paris"] / [§ "color"]	red
%	Propriété	STRING	[§ "Paris"] % [§ "country"]	France
?	Question (QA)	STRING	[§ "..."] ? [§ "question"]	réponse
&&	Conjonction	SEMANTIC	[§ "speed"] && [§ "reliability"]	both
||	Disjonction	SEMANTIC	[§ "tea"] || [§ "coffee"]	alternative
^^	XOR	SEMANTIC	[§ "day"] ^^ [§ "night"]	exclusif
~	Négation (unaire)	SEMANTIC	~[§ "I am happy"]	I am not happy
in	Appartenance	BOOLEAN	[§ "cat"] in [§ "animals"]	1
<|	Sous-concept	BOOLEAN	[§ "dog"] <| [§ "pets"]	1
|>	Super-concept	BOOLEAN	[§ "pets"] |> [§ "dog"]	1
@	Génération code	LAMBDA	$data @ [§ "filter by age"]	lambda

Figure 7.7 — *Référence complète des 20 opérateurs sémantiques de VVL*

Ce chapitre a posé le mécanisme général. Les chapitres 8, 9 et 10 vont approfondir trois opérateurs particuliers qui méritent un traitement détaillé : `@` (génération de code), `[@ "..."]` (réification de concept) et `§§` (compilation de question en programme VVL). Chacun de ces opérateurs est une porte d'entrée vers une capacité neuro-symbolique distinctive : la méta-programmation par LLM, la matérialisation paresseuse de concepts, et l'évaluation de questions en langage naturel.

8. Génération de code par LLM : l'opérateur @

Mots-clés : opérateur @, génération de code, lambda, méta-programmation, validation syntaxique, retry, cache sémantique, sandbox de builtins, opérateur =>, génération libre

Le chapitre 7 a présenté les vingt opérateurs sémantiques de VVL et a brièvement évoqué l'un d'eux — `@` — sans en détailler le fonctionnement. Ce chapitre lui est entièrement consacré, parce que l'opérateur `@` est l'**aboutissement de la philosophie homoiconique** de VVL : il transforme une description en langage naturel en code VVL exécutable, qui s'intègre directement au programme appelant sans aucun pipeline d'intégration. Il est, en un sens, l'opérateur qui rend le LLM véritablement *programmeur* du langage.

Avant d'entrer dans le mécanisme, une **mise au point critique** s'impose : le caractère `@` apparaît dans plusieurs constructions distinctes du langage, qu'on confond facilement. Ce chapitre commence par dissiper toute ambiguïté (8.1), puis traite uniquement l'**opérateur binaire infixé** `$data @ [§ "..."]`. Les autres usages sont renvoyés vers leurs chapitres respectifs.

8.1. Mise au point : les quatre usages du caractère @ en VVL

Le caractère `@` (arobase) apparaît dans la syntaxe de VVL sous quatre formes que rien ne distingue typographiquement, mais qui ont des sens radicalement différents. Cette polysémie est volontaire — elle reflète le fait que VVL utilise `@` partout où il y a *référence indirecte à autre chose* — mais elle exige une lecture attentive.

Forme syntaxique	Catégorie	Sens	Type de retour	Chapitre
$data @ [§ "..."]	Opérateur binaire infixé	Génération de code par LLM	LAMBDA (AST)	Ce chapitre
[@ "nom"]	Commande préfixée	Réification d'un concept en objet	VVL_ConceptEntity	Chapitre 9
@1, @2, @3...	Marqueur de keyword	Référence au paramètre positionnel	(selon contexte)	Chapitre 5
@priority: 100	Annotation Gherkin	Métadonnée de règle d'inférence	— (compile-time)	Chapitre 16

Figure 8.1 — *Désambiguïsation : quatre usages distincts du même caractère*

Le piège classique est de confondre les deux premières formes. **`$data @ [§ "..."]`** est un opérateur binaire — il prend des données à gauche, une description à droite, et retourne une lambda exécutable. **`[@ "Soleil"]`** est une commande préfixée — elle prend un nom de concept et retourne un objet réifié dont les propriétés se découvrent par lazy-load. Les deux invoquent le LLM, mais leur sortie et leur usage diffèrent complètement. Ce chapitre traite **uniquement le premier**. Le second fait l'objet du chapitre 9, où la théorie des objets abstraits de Zalta éclaire pleinement son rôle.

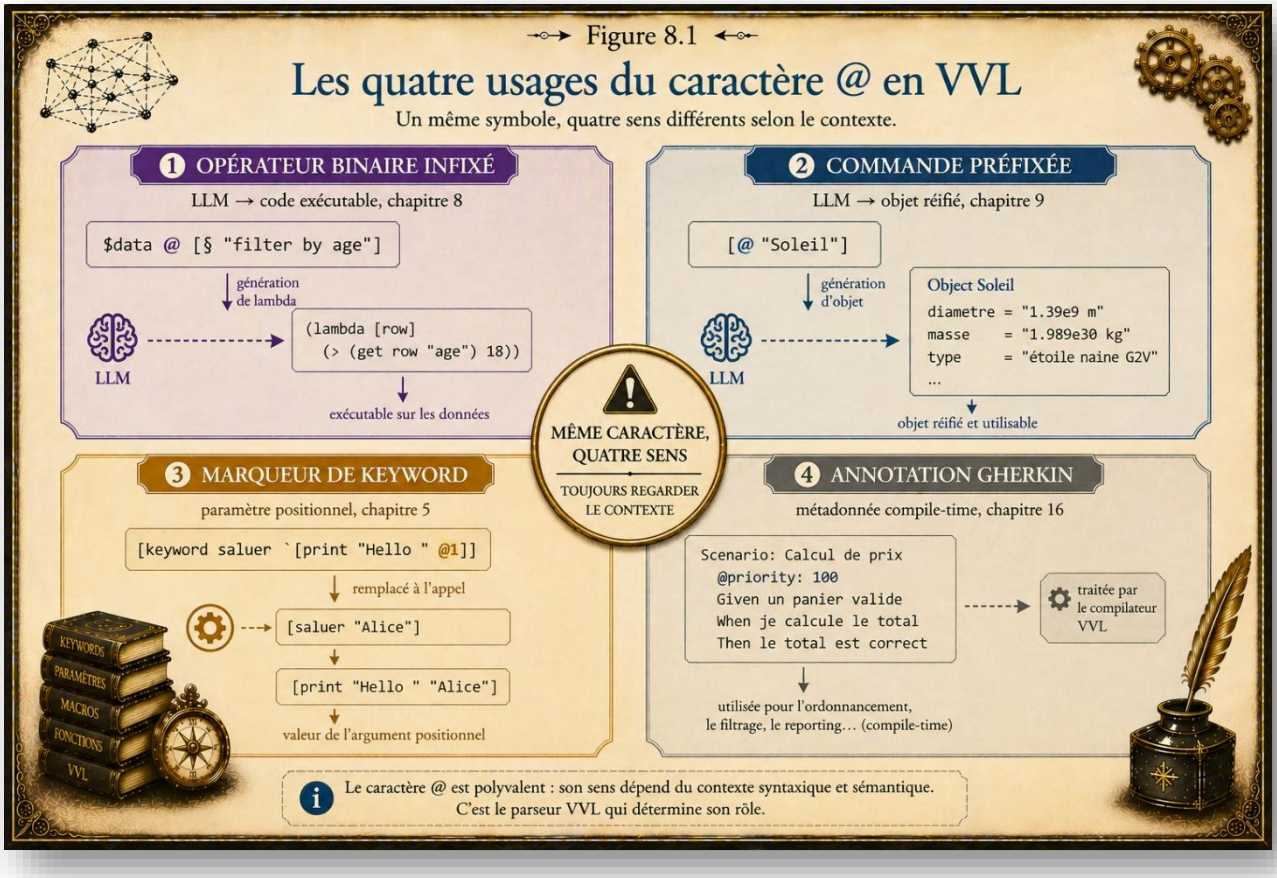


Figure 8.2 — *Le caractère @ en VVL : un quadruple usage à ne pas confondre*

8.2. L'opérateur @ : du langage naturel à la lambda

L'opérateur `@`, dans sa forme binaire infixée `$data @ [§ "description"]`, demande au LLM de produire une **lambda VVL** qui réalise le traitement décrit en langage naturel sur les données fournies. Le résultat est une fonction de première classe — au sens du chapitre 5 — qui peut être passée à `filter`, `map`, ou appelée directement.

La syntaxe canonique est en deux étapes : génération de la lambda, puis application. Le séparer ainsi a l'avantage de pouvoir réutiliser la même lambda sur plusieurs collections sans relancer la génération.

```
# Donnees JSON
$people = <[
    {"name": "Alice",   "age": 25, "city": "Paris"},
    {"name": "Bob",     "age": 32, "city": "Lyon"},
    {"name": "Charlie", "age": 19, "city": "Paris"},
    {"name": "Diana",   "age": 41, "city": "Marseille"}
]>

# Etape 1 : generer le filtre par description naturelle
$fn_age = ($people @ [§ "keep only people older than 27"])
[log $fn_age]
# La lambda generee ressemble a :
# [lambda [o] [if ([INT [jsonGetValue "age" $__o]] > 27) 1 0]]

# Etape 2 : appliquer la lambda generee a la liste
$adults = [filter $fn_age $people]
```

```
[log $adults]
# => [{"name":"Bob","age":32,...}, {"name":"Diana","age":41,...}]

# Reutilisation de la meme lambda sur une autre liste
$other_people = <[
    {"name": "Eve",   "age": 28},
    {"name": "Frank", "age": 17}
]>
$adults_other = [filter $fn_age $other_people]
```

Cinq catégories d'usages couvrent l'essentiel des cas pratiques. Le tableau suivant en donne un aperçu — chaque ligne est un pattern réel que l'opérateur @ traite naturellement :

Catégorie	Exemple	Lambda générée (équivalent)
Filtrage numérique	$people @ [§ "age > 27"]	compare INT(age) à 27
Filtrage textuel	$people @ [§ "from Paris"]	compare ville à "Paris"
Filtrage sur prix	$products @ [§ "under 100$"]	extrait FLOAT du prix
Pattern matching	$files @ [§ "CSV files"]	instr(".csv", name)
Multi-critères	$records @ [§ "active AND verified"]	AND de deux conditions

Figure 8.3 — *Cinq catégories d'utilisation de l'opérateur @*

```
# Filtrage par prix
$products = <[
    {"product": "Laptop",   "price": "999.99"},
    {"product": "Mouse",    "price": "29.99"},
    {"product": "Keyboard", "price": "79.50"}
]>
$cheap_fn = ($products @ [§ "keep products under 100 dollars"])
$cheap = [filter $cheap_fn $products]
[log $cheap]                                # => Mouse, Keyboard

# Pattern matching sur extension
$files = <[
    {"name": "report.pdf",  "size": "2.5 MB"},
    {"name": "data.csv",    "size": "150 KB"},
    {"name": "image.png",   "size": "3.1 MB"},
    {"name": "backup.csv",  "size": "500 KB"}
]>
$csv_fn = ($files @ [§ "keep only CSV files"])
$csvs = [filter $csv_fn $files]
[log $csvs]                                 # => data.csv, backup.csv

# Conditions multiples
$records = <[
    {"status": "active",   "verified": true,  "score": 85},
    {"status": "inactive", "verified": true,  "score": 92},
    {"status": "active",   "verified": false, "score": 67}
]>
$valid_fn = ($records @ [§ "keep only active AND verified items"])
$valid = [filter $valid_fn $records]
[log $valid]                                # => 1 element
```

8.3. Pipeline interne : génération, validation, retry, cache

Comprendre ce qui se passe à l'intérieur de l'opérateur `@` est essentiel pour en faire bon usage en production. Le pipeline d'exécution comporte **six étapes** dont les développeurs ne voient habituellement que l'entrée et la sortie.

Étape 1 — Sérialisation de la structure. L'opérateur reçoit l'opérande de gauche (les données) et en extrait la structure : noms de champs, types, premier exemple. Cette structure est intégrée au prompt sous une forme compacte, pour orienter le LLM vers les bons accesseurs.

Étape 2 — Construction du prompt. Le moteur charge la spécification de l'opérateur `@` depuis le fichier `neuro-symbolic.xml`. Cette spécification contient l'instruction système (« génère une lambda VVL conforme aux règles suivantes... »), les exemples few-shot (trois à cinq lambdas canoniques), et la liste des builtins autorisés. La description sémantique fournie par le développeur est insérée dans le template.

Étape 3 — Appel au LLM. Le prompt complet est envoyé au modèle configuré dans la clé `<agent>` de `vvl.config` — par défaut, un modèle plus puissant que le modèle de chat (typiquement un modèle codex), parce que la génération de code exige une discipline syntaxique plus stricte que la conversation.

Étape 4 — Validation par le parser VVL. La chaîne retournée par le LLM est passée au parser VVL. Si le parser réussit à construire un AST valide — donc si la chaîne est une lambda VVL syntaxiquement correcte — l'étape suivante peut commencer. Sinon, l'étape de retry s'enclenche.

Étape 5 — Retry avec feedback (max 3 tentatives). Si la validation a échoué, le moteur renvoie au LLM le message d'erreur du parser, en lui demandant de corriger sa sortie. Cela se répète jusqu'à trois fois maximum. Au-delà, une erreur est levée et propagée au programme appelant. Dans la pratique, plus de 95 % des générations réussissent à la première tentative.

Étape 6 — Mise en cache et retour. La lambda validée est stockée dans le cache sémantique (clé MD5 de l'opérateur + opérandes). Tout appel ultérieur avec les mêmes paramètres retourne instantanément la lambda mise en cache, sans aucun appel LLM. C'est cette propriété qui rend l'opérateur `@` utilisable en production : **le LLM réfléchit une fois, le code tourne N fois**.

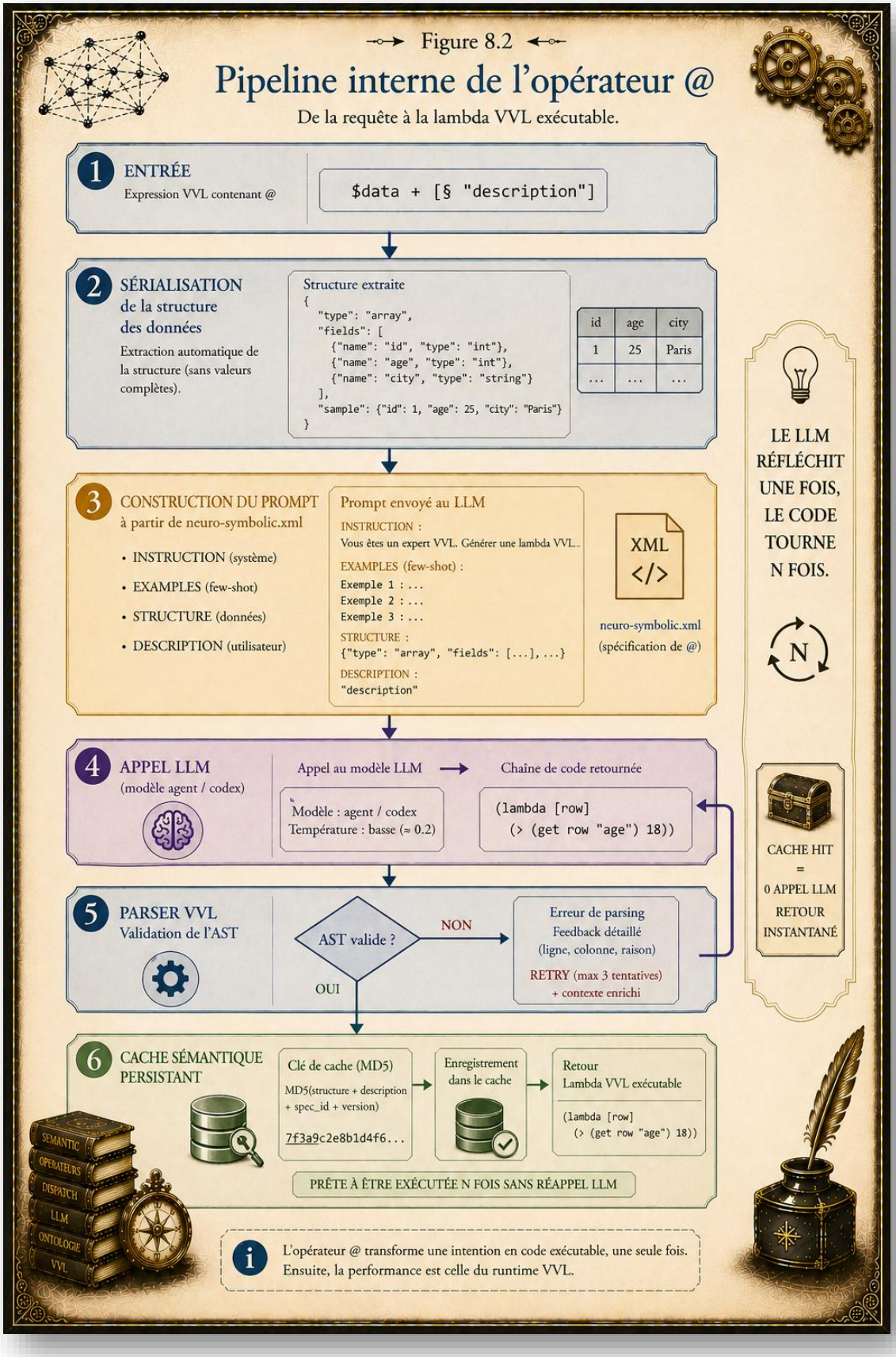


Figure 8.4 — *Les six étapes du pipeline interne de l'opérateur @*

8.4. Sandbox des builtins autorisés dans le code généré

Une question légitime se pose dès qu'on évoque l'évaluation directe de code produit par un LLM : **quel niveau de risque** cela introduit-il ? Si le LLM peut écrire n'importe quel code VVL et que ce code est ensuite évalué, ne risque-t-on pas une injection — par exemple, une lambda qui appellerait `[exec "rm -rf /"]` ou qui ferait fuiter des données via `httpGET` ?

La réponse de VVL est une **liste blanche stricte** : la spécification XML de l'opérateur `@` indique au LLM la liste exacte des builtins qu'il a le droit d'utiliser. Tout autre builtin déclenche une erreur de validation au moment du parsing. Le code généré est ainsi cantonné à un sous-ensemble du langage qui couvre les besoins légitimes (manipulation de données, accès JSON, conversions de types, conditions) sans donner accès aux capacités sensibles (exécution système, accès réseau, accès fichier, modification de la base de connaissances).

Builtin autorisé	Usage
jsonGetValue	Accès à un champ JSON par clé
INT, FLOAT, STRING, BOOL	Conversions de type intelligentes (« 150 EUR » → 150)
instr, instrrev	Position d'une sous-chaîne (1-based)
strToUpperCase, strToLowerCase	Manipulation de casse
length	Longueur d'une liste ou d'une chaîne
if	Condition (le seul builtin de contrôle autorisé)
Opérateurs arithmétiques (+, -, *, /, %)	Calculs sur les valeurs numériques extraites
Opérateurs de comparaison (==, !=, <, >, <=, >=)	Comparaisons stricte (non sémantiques)
Opérateurs logiques (&&, ||)	Combinaison de conditions

Figure 8.5 — *Liste blanche des builtins utilisables dans le code généré par @*

Cette restriction n'est pas une simple convention : elle est **active à deux niveaux**. D'abord côté LLM, par les instructions du prompt et les exemples few-shot qui ne montrent que ces builtins. Ensuite côté validation, par le parser qui rejette toute lambda contenant un identifiant ne figurant pas dans la liste. Un développeur attentif notera que ni `evaluate`, ni `exec`, ni `httpGET`, ni `kb`, ni les commandes ontologiques ne sont autorisés — par construction, le code généré ne peut pas faire de méta-évaluation, d'I/O, ni modifier l'état persistant du système.

Figure 8.6 — *Le sandbox actif à deux niveaux : prompt (souple) + parser (strict)*

8.5. Patterns d'utilisation et bonnes pratiques

Pattern 1 — Génération unique, application multiple

La règle de base est : **générer la lambda une seule fois**, l'appliquer autant de fois que nécessaire. Grâce au cache, même si on omet cette précaution, l'appel sera servi sans coût après la première exécution ; mais structurer le code en deux étapes améliore la lisibilité.

```
# Bon : la lambda est nommee, son intention est lisible
$is_adult = ($people @ [§ "keep people older than 27"])
$adults_paris  = [filter $is_adult $paris_people]
$adults_lyon   = [filter $is_adult $lyon_people]
$adults_marsei = [filter $is_adult $marseille_people]

# A eviter : on inline la generation a chaque fois (cache OK mais lisibilite mediocre)
$adults_paris  = [filter ($paris_people @ [§ "older than 27"]) $paris_people]
```

Pattern 2 — Composition avec map

L'opérateur `@` n'est pas limité au filtrage : il sert aussi à la transformation. Une lambda qui retourne une valeur transformée plutôt qu'un booléen est utilisable avec `map`.

```
# Generation de transformation, pas de filtre
$products = <[
    {"name": "Laptop", "price": "999.99 USD"},
    {"name": "Mouse",  "price": "29.99 USD"}
]>

# Lambda de transformation
$to_eur = ($products @ [§ "convert price from USD to EUR (rate 0.92), return new object"])

$products_eur = [map $to_eur $products]
[log $products_eur]
```

Pattern 3 — Description précise plutôt que vague

La qualité de la lambda générée dépend directement de la précision de la description. Une description vague produit du code qui peut fonctionner sur les exemples d'entraînement mais échouer sur les cas limites. Une description précise — qui mentionne les noms de champs exacts, les unités, les seuils — produit une lambda robuste.

```
# Mediocre : description ambigue
$fn = ($orders @ [§ "expensive orders"])
# Le LLM doit deviner ce qui est cher.

# Mieux : seuil precise
$fn = ($orders @ [§ "orders with total > 1000"])
# Mais le LLM doit deviner le nom du champ.

# Excellent : champ et seuil precises
$fn = ($orders @ [§ "orders where field 'total_amount' > 1000"])
```

Pattern 4 — Échec de validation : que faire ?

Si après trois tentatives le parser refuse toujours la lambda générée, l'opérateur `@` lève une erreur que le programme appelant doit gérer. Trois stratégies sont disponibles selon la criticité : `try/catch` pour ignorer ou logger, fallback vers une lambda écrite à la main, ou demande de re-formulation à l'utilisateur.

```
# Strategie 1 : fallback vers lambda manuelle
$fn = [try
    ($orders @ [§ "complex condition that might fail"])
    [lambda [o] 0]                          # tout rejeter par defaut
]

# Strategie 2 : log et propagation
$fn = [try
    ($orders @ [§ "..."])
    [do [log "@ generation failed"] (raise "FILTER_GEN_FAILED")]
]
```

8.6. L'opérateur => : génération libre de texte, JSON, code

L'opérateur `@` produit une lambda VVL avec validation syntaxique et sandbox. C'est puissant mais contraignant : on ne peut pas l'utiliser pour générer du JSON libre, du Markdown, du SQL ou n'importe quel autre format que des lambdas VVL. C'est pourquoi VVL fournit un opérateur complémentaire, `=>`, qui réalise un **appel LLM brut** sans validation ni typage du retour.

La syntaxe canonique sépare l'instruction des données : `($prompt => [§ "contenu"])`. Cette séparation permet de réutiliser le même prompt sur des entrées différentes, comme on réutilise une lambda. Quand il n'y a pas de données distinctes, on utilise `[§ ""]` à droite.

```
# Application d'une instruction sur differentes donnees
$traduire = "Traduis en francais :"
[log ($traduire => [§ "Hello world"])]   # => Bonjour le monde
[log ($traduire => [§ "Good morning"])]  # => Bonjour

# Generation sans donnee distincte (prompt auto-suffisant)
$prompt = "Retourne un JSON avec un champ 'name' contenant un prenom aleatoire."
$json = ($prompt => [§ ""])
[log $json]                              # => {"name": "Alice"}

# Resume d'un long texte
$summarize = "Resume en une phrase :"
$resume = ($summarize => [§ $long_texte])
```

La différence essentielle entre `@`, `=>` et `[agent]` (présenté au chapitre 22) est leur **mode d'exécution**. Le tableau suivant clarifie les cas d'usage :

Opérateur	Mode	Sortie	Validation	Cas d'usage
@	Génération + parsing	Lambda VVL	AST validé, retry	Filtres et transformations sur données
=>	Appel LLM unique	Texte brut	Aucune	Texte, JSON, Markdown, code source
[agent]	Boucle ReAct	Résultat exécuté	Validation + exécution	Tâches autonomes multi-étapes (ch. 22)

Figure 8.7 — *Trois opérateurs de génération, trois cas d'usage*

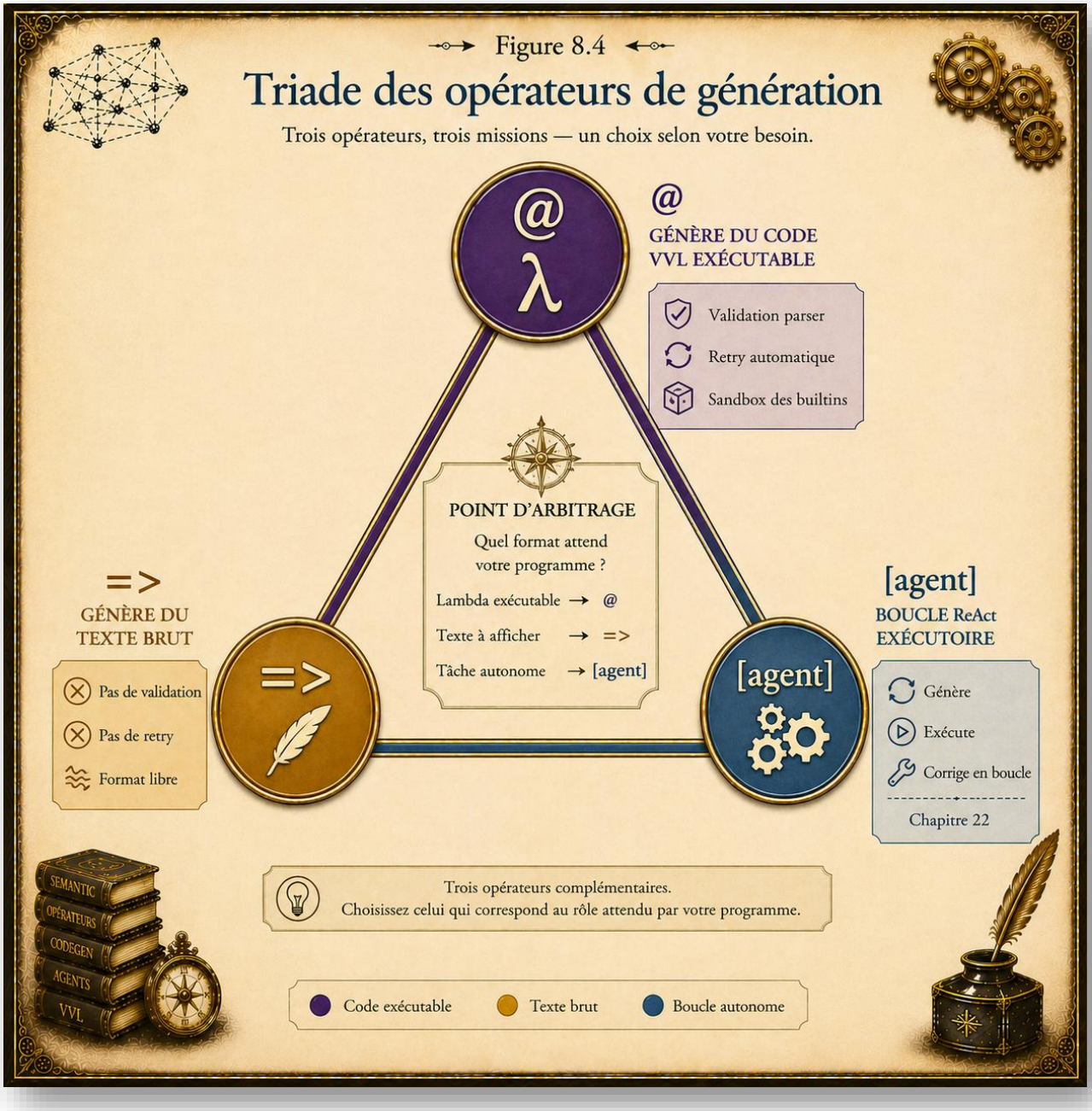


Figure 8.8 — *La triade des opérateurs de génération et leur arbitrage*

8.7. Synthèse et transition

L'opérateur `@` est l'expression la plus complète de la philosophie homoiconique de VVL : un programme qui se génère lui-même via le LLM, valide la syntaxe du résultat, le met en cache, et l'exécute dans un sandbox sûr. Les six étapes de son pipeline interne — sérialisation, prompt, génération, validation, retry, cache — résolvent ensemble le problème de fiabilité que pose la génération de code par LLM dans les systèmes classiques.

Trois choses sont à retenir avant d'aborder le chapitre 9. **Premièrement**, l'opérateur `@` (binaire infixé) ne se confond pas avec la commande `[@ "nom"]` (préfixée), même s'ils partagent le même caractère ;

le chapitre suivant traite la seconde. **Deuxièmement**, le code généré est cantonné à une liste blanche de builtins inoffensifs — il ne peut pas faire d'I/O ni de méta-évaluation, ce qui le rend utilisable en production sans risque d'injection. **Troisièmement**, `@` n'est pas un substitut universel : pour générer du texte libre on utilise `=>`, pour orchestrer une tâche autonome multi-étapes on utilise `[agent]`.

Le **chapitre 9** introduit l'autre opérateur du caractère `@` — la commande de réification `[@ "nom"]` — qui relie pour la première fois dans cet ouvrage le langage VVL à la théorie des objets abstraits de Zalta présentée au chapitre 2. Là où `@` produit du code, `[@ "nom"]` produit un **objet** dont les propriétés se découvrent paresseusement au fil des accès. C'est la porte d'entrée vers le système ontologique du chapitre 12.

9. Réification de concepts : l'opérateur [@ "..."]

Mots-clés : réification, principe de compréhension, Zalta, VVL_ConceptEntity, lazy load, cache trois niveaux, bootstrap, désambiguïsation contextuelle, typage automatique

Le chapitre 8 a traité l'opérateur binaire infixé `$data @ [§ "..."]` qui produit du **code**. Ce chapitre 9 traite la commande préfixée `[@ "nom"]` — typographiquement très proche, conceptuellement très différente — qui produit un **objet**. Là où l'opérateur `@` du chapitre précédent demande au LLM d'écrire une lambda, la commande `[@ "..."]` lui demande de matérialiser un concept en objet manipulable, dont les propriétés se découvriront paresseusement au fil des accès.

Cette commande est d'une importance théorique particulière : elle est l'**opérationnalisation directe du principe de compréhension d'Edward Zalta** (chapitre 2). Pour tout concept nommable — Soleil, Mozart, Démocratie, BMW Série 3, Acier 316L —, le LLM peut nous dire quelles propriétés le définissent. La commande `[@ "..."]` matérialise cette connaissance latente sous forme d'objet formel, et c'est ce qui permettra au chapitre 13 d'appliquer aux concepts l'algèbre complète des opérateurs Zalta. Sans réification, pas d'opérateurs Zalta ; sans opérateurs Zalta, pas de pont entre le neuronal amorphe du LLM et le symbolique manipulable de Prolog.

Le chapitre suit la séquence suivante : du nom au modèle paramétrique (9.1), le cycle de vie en trois phases (9.2), le cache à trois niveaux (9.3), le typage automatique des propriétés (9.4), la désambiguïsation contextuelle par auto-réification (9.5), et enfin le concept comme citoyen de première classe du système OO (9.6).

9.1. Du nom au modèle paramétrique

L'idée fondatrice est d'une simplicité provocante : un LLM, par son entraînement sur l'ensemble du Web et de la littérature numérisée, a accumulé une connaissance implicite de millions de concepts. Demander à GPT « quelle est la masse du Soleil » produit la réponse `1,989 × 10³⁰ kg` ; demander « quelle est l'année de naissance de Mozart » produit `1756` ; demander « quelle est la résistance à la traction de l'acier inoxydable 316L » produit `~515 MPa`. Cette connaissance est **déjà là, encodée dans les poids du modèle**, mais elle reste *amorphe* : on ne peut l'interroger qu'en posant des questions une à une, en parsant les réponses textuelles, en gérant les hallucinations.

La commande `[@ "..."]` matérialise cette connaissance amorphe en objet formel. Quand on écrit `$soleil = [@ "Soleil"]`, on demande au LLM de produire en un seul appel un noyau de propriétés essentielles du Soleil — typiquement type, diamètre, masse, âge, température. Le résultat est un objet de type `VVL_ConceptEntity` qui hérite de `VVL_Instance` et qui supporte donc tout le système OO du chapitre 6 : accès aux champs par notation pointée, dispatch de méthodes, persistance, intégration Prolog.

```
# Reification d'un concept simple
$soleil = [@ "Soleil"]
[log $soleil]
# => <Concept:Soleil [type, diameter_km, mass_kg, age, temperature_k, ...]>

# Acces aux proprietes par notation pointee (style OO classique)
[log $soleil.type]              # => "G-type main-sequence star"
[log $soleil.diameter_km]       # => 1392700
[log $soleil.mass_kg]           # => 1.989e30

# La reification fonctionne pour tout concept nommable
$mozart  = [@ "Mozart"]
$acier   = [@ "Acier inoxydable 316L"]
$demo    = [@ "Democratie"]
$bmw     = [@ "BMW Serie 3 diesel 2024"]

# Meme pour des concepts specifiques avec attributs dans le nom
[log $bmw.consommation_l_100km]  # => 5.1  (typique pour cette variante)
```

Le contraste avec `[§ "..."]` du chapitre 7 est essentiel à comprendre. **`[§ "texte"]`** crée une **valeur sémantique** — un texte qui sera interprété par le LLM lorsqu'il participera à une opération binaire (==, +, %, etc.). C'est une *valeur*, pas un objet. **`[@ "nom"]`** crée un **objet réifié** — une instance de `VVL_ConceptEntity` avec ses champs, ses méthodes, sa persistance, son intégration au système OO. C'est un *objet*, pas une valeur.

Critère	[§ "texte"]	[@ "nom"]
Type de retour	VVL_Semantic (valeur)	VVL_ConceptEntity (objet)
Usage canonique	Opérande d'opérateur sémantique	Stocké dans une variable, manipulé
Appel LLM initial	Aucun	1 appel (bootstrap des propriétés)
Persistance	Aucune (valeur immédiate)	Cache disque .concept_cache.json
Compatibilité OO	Non (valeur primitive)	Oui (hérite de VVL_Instance)
Exemple typique	[§ "chat"] == [§ "felin"]	$soleil = [@ "Soleil"]

Figure 9.1 — *Distinction critique entre [§ "..."] (valeur sémantique) et [@ "..."] (objet réifié)*

*

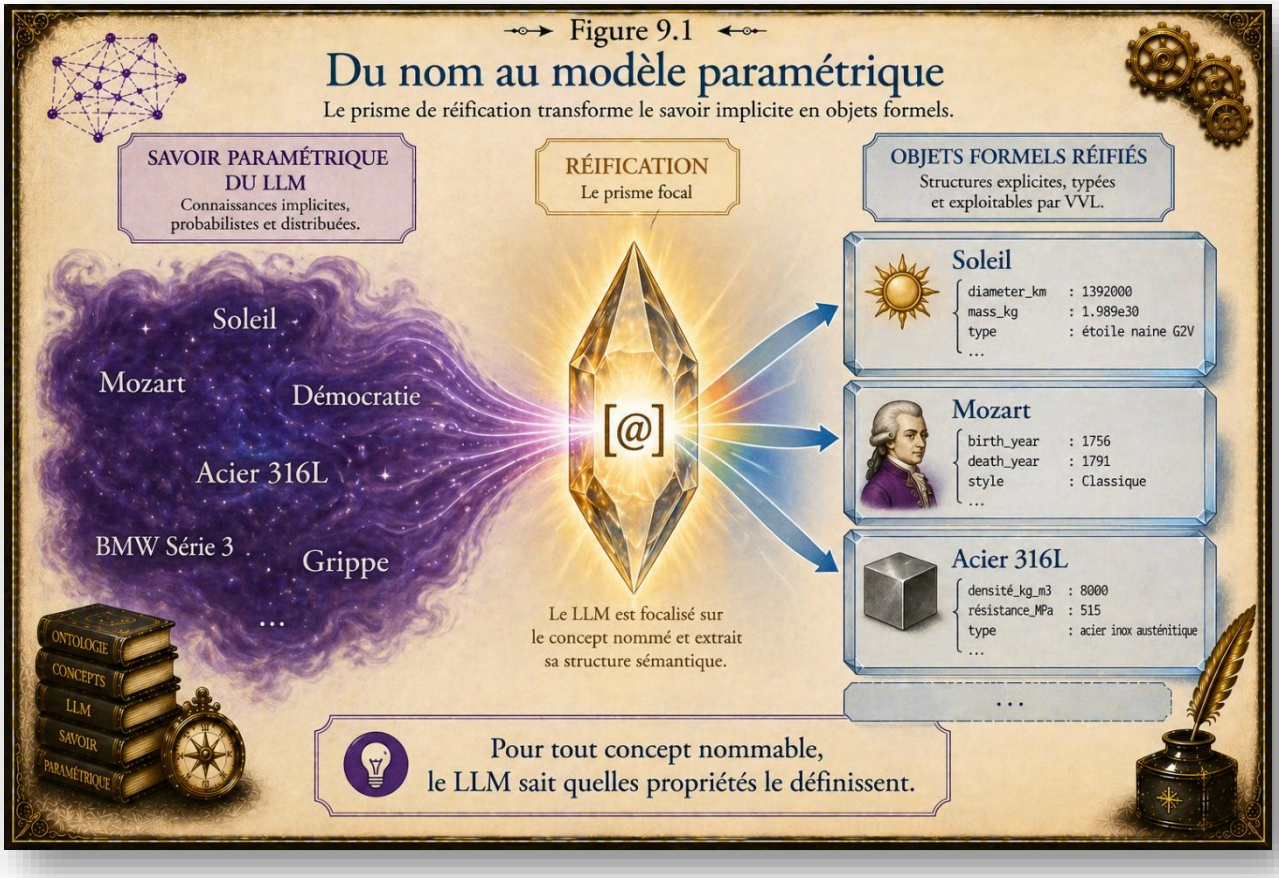


Figure 9.2 — *La réification : le prisme qui transforme l'amorphe paramétrique en objet formel*

9.2. Le cycle de vie en trois phases : bootstrap, lazy load, manipulation

Le cycle de vie d'un concept réifié se décompose en trois phases dont l'enjeu commun est de **réduire l'usage du LLM au strict nécessaire**. La première phase est inévitable : il faut au moins un appel pour obtenir un noyau de propriétés. Les phases suivantes sont une descente progressive vers le purement symbolique.

Phase 1 — Bootstrap

À la création de l'objet, un **appel LLM unique** demande les huit à dix propriétés les plus essentielles du concept. Le LLM retourne un JSON structuré dont les clés sont des identifiants `snake_case` et les valeurs sont automatiquement typées (nombres, chaînes, listes). Ce bootstrap peuple l'objet avec un noyau de propriétés factuelles immédiatement disponibles.

```
# Phase 1 : Bootstrap (1 appel LLM)
$soleil = [@ "Soleil"]

# L'objet est immediatement peuple
[log $soleil.properties]
# => [type, diameter_km, mass_kg, age, temperature_k, luminosite, distance_terre, ...]

# Mode "lazy" : pas de bootstrap, decouverte purement a la demande
$truc = [@ "Truc" "lazy"]
[log $truc.properties]                    # => []  (vide au depart)
```

Phase 2 — Découverte paresseuse

Tout accès à une propriété **non encore connue** déclenche un appel LLM ciblé. La réponse est typée, stockée dans l'objet (champ ajouté à `self.fields`), et persistée sur disque. Les accès suivants à la même propriété sont **instantanés** — ils lisent simplement le cache. Ce mécanisme transforme le LLM en une base de connaissances interrogeable propriété par propriété, où chaque interrogation devient un investissement définitif.

```
# Phase 2 : Decouverte paresseuse
$soleil = [@ "Soleil"]

# Premier acces a "luminosite" : la propriete n'a pas ete bootstrap
[log $soleil.luminosite]                  # appel LLM => "3.828e26 W"

# Second acces a la meme propriete : cache, 0 appel
[log $soleil.luminosite]                  # => "3.828e26 W" (instantane)

# Acces a une propriete typee comme liste
$france = [@ "France"]
[log $france.villes_principales]          # => ["Paris", "Lyon", "Marseille", ...]
[log $france.langues_officielles]         # => ["Francais"]
```

Phase 3 — Manipulation formelle

Une fois ses propriétés matérialisées, l'objet supporte l'**intégralité des opérations formelles** sans plus aucun appel LLM. C'est la phase où interviennent les opérateurs Zalta du chapitre 13 : satisfaction (==, ~~), intersection (*, **), enrichissement (+, ++), projection (/, //). Le LLM n'intervient plus — seul le moteur symbolique opère, ce qui rend ces manipulations rapides, déterministes, et sans coût API.

```
# Phase 3 : Manipulation purement formelle (0 appel LLM)
$soleil = [@ "Soleil"]
$terre  = [@ "Terre"]

# Intersection des proprietes communes (operateur Zalta, ch. 13)
$commun = ($soleil * $terre)
[log $commun.properties]                  # proprietes partagees

# Projection symbolique
[log ($soleil / "type")]                  # => "G-type main-sequence"

# Comparaison hybride : satisfies (un seul appel LLM pour la comparaison
#  globale, pas pour chaque propriete)
[log ($soleil ~~ "Etoile")]               # => ~0.95
```

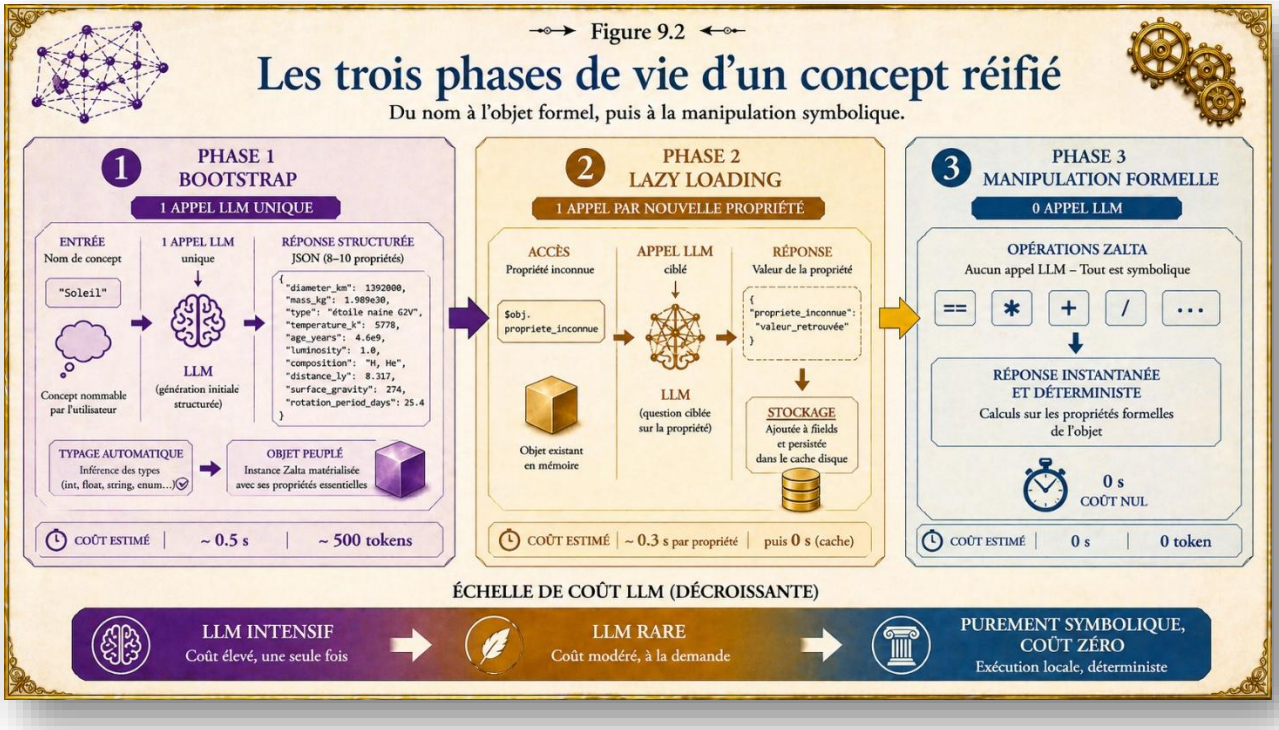


Figure 9.3 — *Du LLM à la manipulation purement symbolique : trois phases, un coût décroissant*

9.3. Le cache à trois niveaux : session, objet, disque

L'efficacité du système de réification repose sur un **cache à trois niveaux** qui garantit qu'aucune propriété n'est jamais redemandée au LLM une fois découverte. Cette pyramide de cache est ce qui transforme un système probabiliste coûteux en un investissement définitif : la première session avec un concept demande quelques appels LLM, mais toutes les sessions suivantes sont purement symboliques — zéro appel LLM, zéro coût API, zéro latence réseau.

Niveau 1 — Mémoire de session

Le premier niveau opère dans `ctx.variables` : deux invocations de `[@ "Soleil"]` dans la même session retournent le même objet Python. La clé de cache est de la forme `__concept_soleil`. Durée : la session REPL ou l'exécution du script. C'est le cache le plus rapide (~0 ms) mais le plus volatile.

Niveau 2 — Champs de l'objet

Le deuxième niveau opère sur l'objet lui-même : `self.fields` stocke chaque propriété après son premier accès. Une propriété déjà résolue se lit instantanément (~0 ms). Durée : la vie de l'objet en mémoire.

Niveau 3 — Disque

Le troisième niveau opère sur disque : un fichier JSON `.concept_cache.json` à la racine du workspace persiste toutes les propriétés découvertes entre les sessions. Durée : permanent, jusqu'à un `[concept:reset]` explicite. Latence : ~1 ms (lecture disque). C'est ce niveau qui transforme l'investissement initial en bénéfice durable.

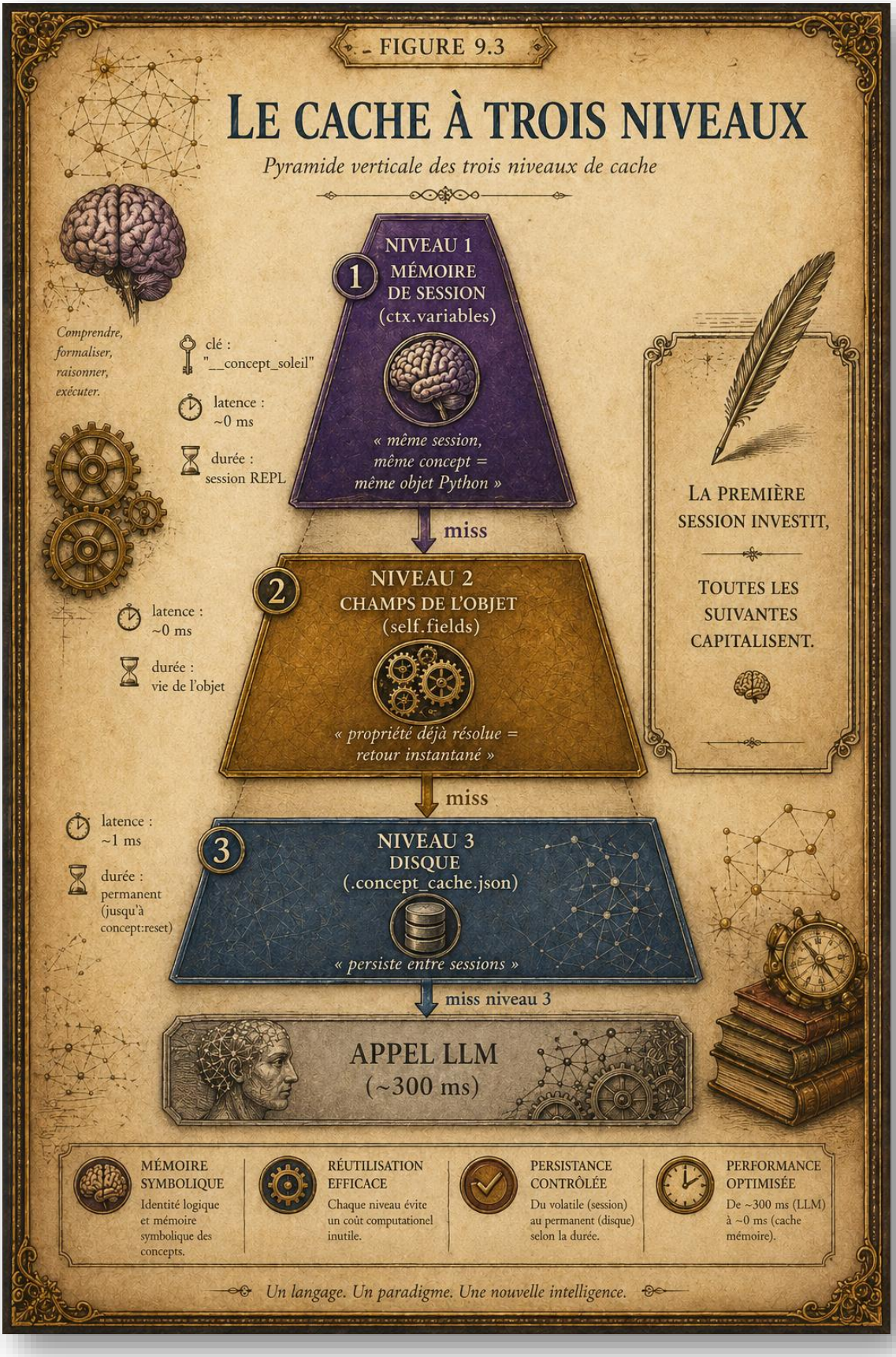


Figure 9.4 — *La pyramide du cache : session, objet, disque — puis seulement le LLM*

Une note importante concerne les **concepts auto-réifiés avec contexte** (par exemple `"Étoile"` réifié automatiquement dans l'expression `$soleil ~~ "Étoile"`, voir section 9.5). Ceux-ci sont stockés en cache de session **uniquement** — pas sur disque — pour éviter de polluer le cache avec une version désambiguïsée qui ne serait pas correcte dans un autre contexte. La clé de cache de session inclut un hash du hint de contexte, ce qui signifie que `"Étoile"` réifié dans le contexte du Soleil et `"Étoile"` réifié dans le contexte d'un restaurant sont deux objets distincts.

9.4. Typage automatique des propriétés

Le LLM retourne du **texte**. Pour que les propriétés soient utilisables dans des opérations arithmétiques, des comparaisons numériques ou des itérations, le moteur de réification effectue une **conversion de type automatique** sur chaque réponse. La fonction `_auto_type` tente, dans l'ordre, plusieurs conversions et retient la première qui réussit.

Cette conversion est essentielle : sans elle, `$soleil.diameter_km > 1000000` serait une comparaison de chaînes (qui échouerait silencieusement), et `[for $v $france.villes_principales ...]` itérerait sur les caractères d'une chaîne au lieu d'éléments d'une liste. Le typage automatique rend les propriétés directement utilisables comme s'il s'agissait de valeurs natives.

Réponse LLM	Type VVL	Description
"42"	VVL_Num(42)	Entier pur
"3.14"	VVL_Num(3.14)	Flottant
"1 392 700"	VVL_Num(1392700)	Nombre avec séparateurs
"8.0 g/cm3"	VVL_Num(8.0)	Nombre + unité (l'unité est ignorée)
"5.972 x 10^24"	VVL_Num(5.972e24)	Notation scientifique
"1.989e30"	VVL_Num(1.989e30)	Notation E
["Paris", "Lyon"]	VVL_List	Tableau JSON
"true" / "false"	VVL_Num(1/0)	Booléen
"N/A"	VVL_String("")	Marqueur d'inapplicabilité
(autre)	VVL_String	Texte libre

Figure 9.5 — *Conversions automatiques appliquées aux réponses du LLM*

Les listes sont gérées récursivement : chaque élément d'un tableau JSON est lui-même auto-typé individuellement. Ainsi `"villes_principales": ["Paris", "Lyon"]` devient une `VVL_List` de `VVL_String`, mais `"populations": [2100000, 500000]` devient une `VVL_List` de `VVL_Num`. Le typage est suffisamment robuste pour gérer les variations de format que le LLM produit naturellement (avec ou sans séparateurs de milliers, avec ou sans unité, en notation scientifique ou décimale).

9.5. Désambiguïsation contextuelle par auto-réification

Le mécanisme le plus subtil — et le plus puissant — du système de réification est l'**auto-réification contextualisée**. Quand un opérateur Zalta du chapitre 13 reçoit à gauche un concept réifié et à droite une chaîne (par exemple `($soleil ~~ "Étoile")`), le système doit réifier `"Étoile"` à la volée pour effectuer la comparaison. La question délicate est : *quel* `"Étoile"` ? L'étoile astronomique ? L'étoile de mer ? L'étoile Michelin ? L'étoile de cinéma ?

La réponse de VVL est élégante : **utiliser le contexte de l'opérande gauche pour désambiguïser**. La fonction `_auto_reify` construit un *hint* de contexte à partir des propriétés connues de l'opérande gauche — par exemple « Soleil (type=G-type star, diameter_km=1392700, mass_kg=1.989e30) » — et l'injecte dans le prompt de bootstrap du concept droit. Le LLM comprend alors qu'il faut produire des propriétés astronomiques.

```
# Cas 1 : Etoile dans le contexte du Soleil = etoile astronomique
$soleil = [@ "Soleil"]
[log ($soleil ~~ "Etoile")]              # => ~0.95
# Le concept "Etoile" auto-reifie a recu le hint :
#   "Soleil (type=G-type star, diameter=1392700, mass=1.989e30)"
# Le LLM produit donc : type=stellar, mass=varies, luminosity=...

# Cas 2 : Etoile dans le contexte d'un restaurant = etoile Michelin
$restaurant = [@ "Le Jules Verne Paris"]
[log ($restaurant ~~ "Etoile")]          # => ~0.80
# Le concept "Etoile" auto-reifie a recu un hint different :
#   "Le Jules Verne Paris (type=restaurant, cuisine=francaise, ...)"
# Le LLM produit donc : rating=1-3, guide=Michelin, cuisine=...

# Les deux objets "Etoile" sont distincts (cle de session avec hash du hint)
```

L'injection du contexte a un second effet, tout aussi important : elle aligne les **noms de propriétés**. Le LLM tend à réutiliser les mêmes noms (`diameter_km`, `mass_kg`) que ceux de l'opérande gauche, ce qui améliore considérablement la qualité des intersections symboliques (*) et des comparaisons. Sans cette uniformisation, deux concepts du même domaine pourraient utiliser des noms de propriétés différents (`taille` vs `diameter_km`) et apparaître artificiellement dissemblables au moteur symbolique.

Cette stratégie est conforme au **principe de compréhension de Zalta** rappelé au chapitre 2 : l'objet abstrait est défini par la condition ϕ qu'il satisfait — et le contexte fait partie de cette condition. Deux concepts du même nom dans des contextes différents *sont* deux objets abstraits différents, pas une seule entité ambiguë. C'est précisément pourquoi les concepts auto-réifiés ne sont pas persistés sur disque : le contexte étant constitutif de l'objet, le persister hors contexte introduirait une corruption sémantique.

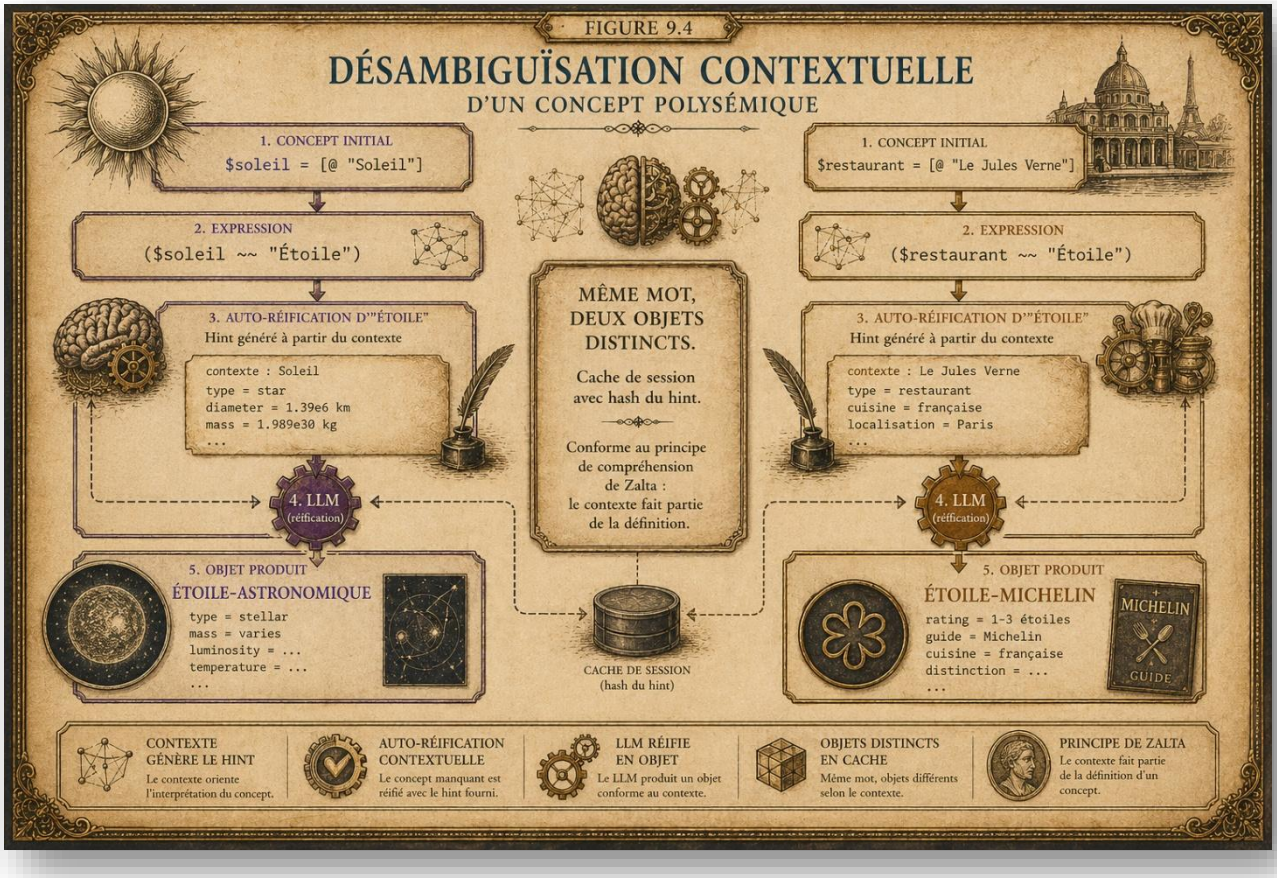


Figure 9.6 — *Le même mot produit deux objets distincts selon le contexte de l'opérande gauche*

9.6. Le concept comme citoyen de première classe

La classe `VVL_ConceptEntity` produite par `[@ "..."]` **hérite de `VVL_Instance`** — la classe de base du système OO de VVL présenté au chapitre 6. Cette filiation a une conséquence pratique majeure : tout ce qu'on peut faire avec une instance ordinaire, on peut le faire avec un concept réifié. Notation pointée pour les champs, dispatch de méthodes, surcharge d'opérateurs, persistance, intégration aux mixins.

Au-delà des propriétés découvertes par le LLM, chaque concept réifié expose un ensemble de **méta-propriétés** qui ne sont jamais envoyées au LLM — elles sont calculées localement à partir de la structure de l'objet :

```
$soleil = [@ "Soleil"]

# Meta-proprietes (jamais d'appel LLM)
[log $soleil.name]              # => "Soleil"
[log $soleil.properties]        # => liste des proprietes connues
[log $soleil.encoded]           # => dict {prop: val, ...}
[log $soleil.json]              # => dict complet avec metadonnees
[log $soleil.prolog_id]         # => "concept_a3f8b2c1..."  (identifiant Prolog)

# Compatibilite avec [is-a]
[log [is-a $soleil VVL_ConceptEntity]]   # => 1
[log [is-a $soleil VVL_Instance]]        # => 1  (heritage)

# Surcharge utilisateur d'un operateur
# (priorite sur le mixin ZaltaOperators du chapitre 13)
[class MaCarte
    [field id 0]
    [method "==" [other]                  # surcharge : compare par id
        ($self.id == $other.id)
    ]
]
```

La méta-propriété `prolog_id` est particulièrement importante : c'est l'identifiant que prendra l'objet lorsqu'il sera projeté dans la base Prolog du chapitre 11 via `obj-assert`. Cela signifie qu'un concept réifié peut être inséré dans la KB Prolog comme un fait, interrogé par `query`, manipulé par les règles d'inférence. **C'est l'aboutissement de la promesse neuro-symbolique de VVL** : un même objet vit dans le monde du LLM (par sa découverte paresseuse), dans le monde du langage (par son intégration OO), et dans le monde de la logique (par sa projection Prolog).

Le tableau ci-dessous récapitule les opérations canoniques qu'on peut appliquer à un concept réifié, en distinguant celles qui n'impliquent pas le LLM (gauche) de celles qui peuvent l'invoquer (droite) :

Opération sans LLM (rapide, déterministe)	Opération avec LLM (rare, cachée)
$obj.propriete (déjà connue)	$obj.propriete (inconnue, lazy load)
$obj.name, .properties, .json	[@ "..."] (création initiale)
$obj1 == $obj2 (Zalta symbolique)	$obj1 ~~ $obj2 (Zalta hybride, ch. 13)
$obj1 * $obj2 (intersection sym.)	$obj1 ** $obj2 (intersection hybride)
$obj / "prop" (projection exacte)	$obj // "prop" (projection sémantique)
[obj-assert $obj] (Prolog, ch. 11)	($obj1 ~~ "string") (auto-réification, sec. 9.5)

Figure 9.7 — *Opérations sur un concept réifié : avec et sans LLM*

Avec ce chapitre, le pont conceptuel entre le LLM et le système symbolique est entièrement posé. Le **chapitre 10** clôt la Partie III avec le dernier opérateur sémantique distinctif — §§, qui *compile* une question en langage naturel en programme VVL combinant réification, accès aux propriétés et calcul. Là où le chapitre 7 a montré le LLM en *opérateur* d'expression et le chapitre 8 le LLM en *générateur* de fonction, le chapitre 9 vient de montrer le LLM en *fournisseur* d'objets, et le chapitre 10 montrera le LLM en *compilateur* de questions naturelles. Quatre rôles, un seul langage.

10. L'opérateur §§ : compilation du langage naturel

Mots-clés : §§, §§?, resolve, router, classifier, dispatcher, 8 familles, arithmetic_closed, constraint_optim, relational_query, compound, rule_induction, factual_retrieval, structured_extraction, expert_system, Approximation, audit, clarification gate, fallback défensif, opérateur ^, math builtins, mode trace, [generic], catalogue, confiance

Ce chapitre clôt la Partie III du livre par le quatrième et dernier visage du LLM dans VVL. Les chapitres précédents ont montré le LLM en *opérateur* d'expression (chapitre 7), en *générateur* de fonction (chapitre 8), en *fournisseur* d'objet (chapitre 9). Reste à montrer le LLM en *compilateur et routeur neuro-symbolique* — c'est-à-dire la capacité à prendre une question en langage naturel, à en déduire la nature, à la dispatcher vers le solveur spécialisé approprié, et à emballer la réponse dans une *Approximation* auditable. C'est le rôle de la paire `[§§ "..."]` et de sa variante verbose `[§§? "..."]`. L'architecture présentée dans ce chapitre est celle du serveur courant, VAILS 7.17.2. Elle intègre les phases ROUTE-1, ROUTE-2, ROUTE-3, ROUTE-EXPERT et COMPOUND×EXPERT du routeur, les phases APX-1 à APX-3 de la propagation d'approximation, ainsi que les durcissements postérieurs : normalisation enum Zalta, opérateur de puissance ^, builtins mathématiques, et garde-fou str×num qui prévient la corruption silencieuse.

Mais résoudre une question ponctuellement ne suffit pas. Une question résolue est un investissement intellectuel ; recompiler la même question à chaque appel serait du gaspillage. C'est pourquoi VVL fournit un mécanisme complémentaire : la commande `[generic "question"]` qui prend la solution ponctuelle de §§, en extrait les paramètres, lui donne un nom, génère des tests d'amorçage, et la **persiste** sous forme de keyword nommé dans un catalogue durable. La paire §§ / [generic] est ainsi à la programmation neuro-symbolique ce que sont l'*interprétation* (§§) et la *compilation* ([generic]) en programmation classique.

Le chapitre suit la séquence suivante : la motivation (10.1), les opérateurs §§ et §§? (10.2), le router neuro-symbolique et son pipeline global (10.3), les huit familles de questions en détail (10.4), l'écosystème des experts dynamiques (10.5), l'*Approximation* comme wrapper d'audit (10.6), la configuration et les comportements défensifs (10.7), les opérateurs et builtins de support (10.8), le mode trace pour le debug (10.9), la commande [generic] et son pipeline d'industrialisation (10.10), l'écosystème du catalogue avec ses douze builtins (10.11), les limites connues (10.12), et enfin une synthèse des quatre rôles du LLM dans VVL qui ferme la Partie III (10.13).

10.1. De la question éphémère au programme réutilisable

Considérons une question apparemment simple : *" Calcule la mensualité d'un crédit de 250 000 EUR sur 20 ans à 3,5 %. "* Pour répondre, un système classique a deux options. **La première** consiste à demander directement au LLM ; la réponse arrivera sous forme de texte qu'il faudra parser, et chaque appel ultérieur consommera des tokens. Surtout, le LLM peut bricoler une formule incorrecte (intérêt simple au lieu d'amortissement). **La seconde** consiste à coder à la main la formule d'amortissement, ce qui rate le bénéfice d'une interface en langage naturel.

VVL propose une troisième voie : §§ **classifie** d'abord la question (ici en `arithmetic_closed` avec `sub_kind = amortization`), puis **dispatche** vers un spécialiste qui sait quelle formule canonique appliquer. Pour une question relationnelle (« Qui sont les grands-parents d'Anne ? »), le même opérateur §§ dispatche vers une compilation en Prolog. Pour une question de planification (« Planifie 8 tâches sur 3 machines minimisant le makespan »), il dispatche vers un compilateur CP-SAT. Pour une question composée (« Extrais le nombre d'employés de ce texte, puis calcule combien d'équipes de 5 on peut former »), il décompose et orchestre. Le résultat est toujours emballé dans une *Approximation* auditable qui trace les hypothèses, la confiance, le code source utilisé, et les avertissements.

Mais ce travail de routage et de compilation reste éphémère : la prochaine fois qu'on posera une question similaire, le moteur recompilera depuis zéro. C'est là qu'intervient `[generic]` : il prend la solution compilée, extrait les concepts comme paramètres, abstrait le code, lui donne un nom, et l'enregistre dans un catalogue. La prochaine fois, on pourra simplement appeler le keyword nommé sans aucun appel au compilateur.

10.2. Les opérateurs §§ et §§? : porte d'entrée universelle

L'opérateur `[§§ "question NL"]` (alias `[resolve "..."]`) est la porte d'entrée universelle du système. Sa sémantique est : une question en langage naturel est classifiée par un LLM, puis dispatchée vers le bon solveur spécialisé, et le résultat est emballé dans une *Approximation* auditable. La variante `[§§? "..."]` (alias `[resolveV "..."]`) est strictement équivalente côté sémantique, mais active du logging détaillé exploitable en mode trace `:t` du REPL.

Le tableau ci-dessous montre comment une même syntaxe d'appel couvre huit familles de questions de nature radicalement différente. C'est précisément ce qui fait de §§ un point d'entrée *universel* : l'utilisateur n'a jamais à choisir le solveur, le router le déduit.

Question NL	Famille déduite	Spécialiste invoqué
`[§§? "Mensualité 250k EUR / 20 ans / 3,5%"]`	`arithmetic_closed`	LLM code-gen avec formule canonique
`[§§? "Planifie 8 tâches sur 3 machines minimisant le makespan"]`	`constraint_optim`	CP-SAT via `[optimize "NL"]`
`[§§? "Qui sont les grands-parents d'Anne ?"]`	`relational_query`	LLM Prolog code-gen + `[query VVL]`
`[§§? "Parmi 2 patients, lequel est prioritaire ..."]`	`compound`	Décompose → 2× expert → compose
`[§§? "Quelle règle sépare ces fidèles des non-fidèles ?"]`	`rule_induction`	LLM induction (NL + Prolog)
`[§§? "Quelle est la capitale du Bénin ?"]`	`factual_retrieval`	LLM connaissance + branche RAG
`[§§? "Extrais nom, âge, métier de '...'"]`	`structured_extraction`	LLM JSON extraction
`[§§? "Évalue le dossier Alice, revenu 4500..."]`	`expert_system`	Keyword dynamique enregistré

Figure 10.1 — *Les huit familles couvertes par §§ et leur spécialiste associé.*

Trois propriétés caractérisent ce mécanisme. **D'abord, la question peut référer à des variables du contexte** : `[§§ "$soleil et $terre ont-ils la même densité ?"]` réutilise les concepts déjà réifiés.

Ensuite, tout est inspectable — en mode trace (commande REPL `:t` ou `ctx.trace = True`), §§ affiche le verdict du classifieur, le code intermédiaire généré, et le résultat de chaque dispatcher, ce qui rend le mécanisme transparent et déboguable. **Enfin, le résolveur ne lève jamais d'exception non capturée** : à chaque étape, un fallback ramène vers la branche `arithmetic_closed` ou vers une *Approximation* de type `_failure_approximation` (cf. § 10.7).

Le router est implémenté en `src/vails/symbolic_router/` (classifier + dispatcher + decomposer + multi + prompts). Le resolver lui-même (entrée `builtin_symbolic_query`) est en `src/vails/vvl_symbolic_query.py`.

10.3. Le router neuro-symbolique : classification et dispatch

L'exécution d'une requête §§ se décompose en deux phases : un **router** qui classifie la question (un seul appel LLM caché, prompt `CLASSIFY_SYSTEM`), suivi d'un **dispatcher** qui appelle le spécialiste de la famille identifiée. Le verdict du classifieur a la forme `{family, sub_kind, confidence, reasoning}` et est mis en cache (cf. § 10.7).

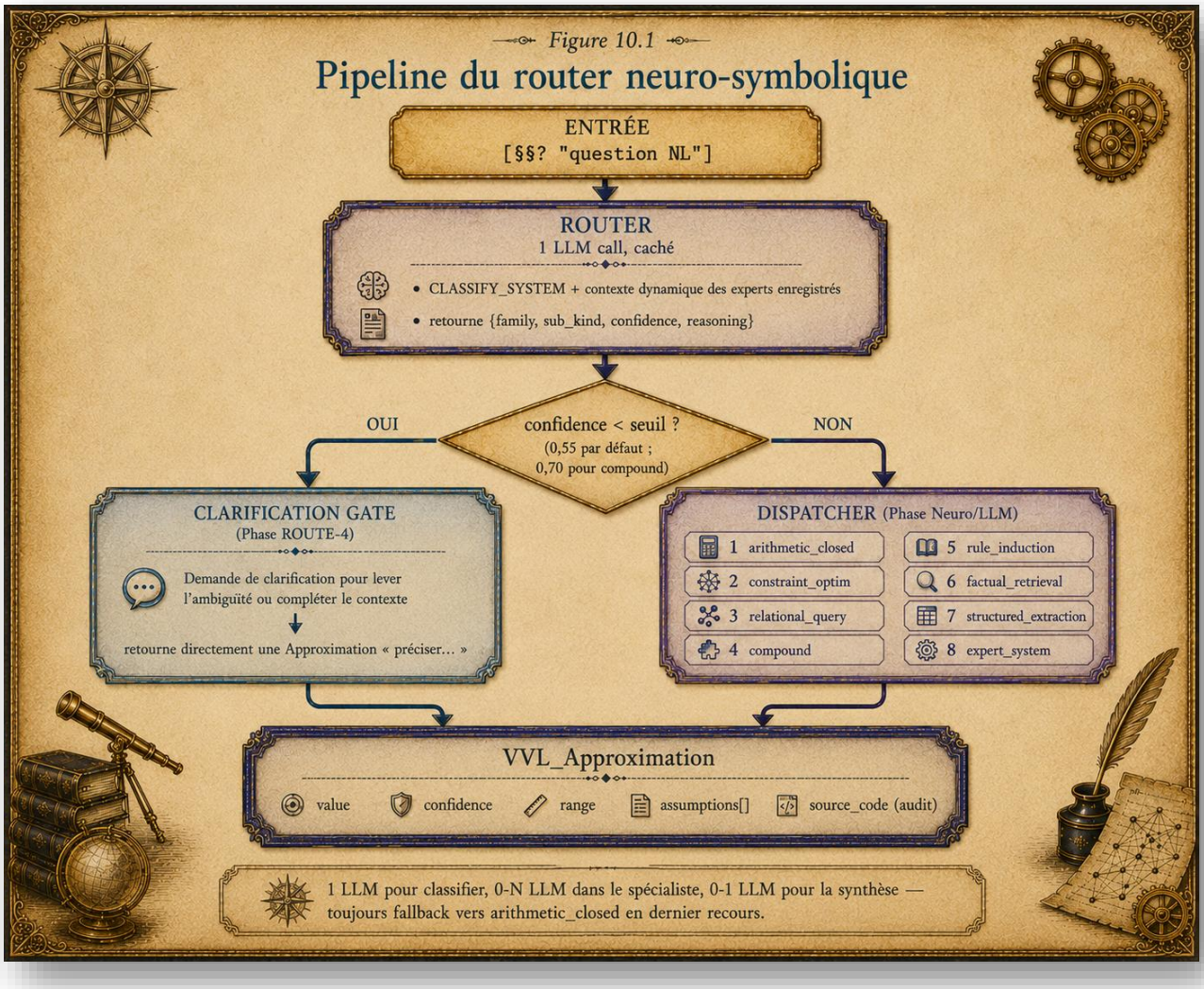


Figure 10.2 — *Pipeline global du router : classification, gate clarification, dispatch par famille, wrapping en Approximation.*

Le rôle du classifieur

Le prompt `CLASSIFY_SYSTEM` injecte dynamiquement dans le message utilisateur la liste des **experts enregistrés dans la session courante** (nom, classe principale, description tronquée), via `format_registered_experts(ctx)`. C'est cette injection qui permet au classifieur de retourner `{family: "expert_system", sub_kind: "<ExpertName>"}` quand la question matche le domaine d'un expert. Le `reasoning` retourné par le classifieur est conservé et préfixé au prompt du dispatcher arith — d'où la règle 5bis (cf. § 10.4.1).

Le gate de clarification

Si la confiance retournée par le classifieur est sous le seuil (0,55 par défaut ; 0,70 pour `compound` parce que la décomposition est coûteuse), le router court-circuite le dispatcher et retourne directement une Approximation portant `clarification_requested = True` et un message NL invitant à préciser la question. Cette phase est appelée `ROUTE-4` dans l'historique. Elle évite que le moteur dépense des appels LLM sur une question ambiguë qui pourrait être interprétée de plusieurs façons mutuellement exclusives.

10.4. Les huit familles de questions

Cette section détaille les huit familles supportées par le dispatcher. Chaque famille a son propre prompt système, son pipeline interne, ses garanties et ses limites. Toutes finissent par renvoyer une *Approximation* : c'est l'interface unifiée du resolver.

10.4.1. arithmetic_closed — calcul direct

Quand : la réponse est un nombre obtenu par arithmétique sur des inputs connus ou estimables. Couvre environ 80 % des questions. C'est aussi la branche de fallback de tous les autres dispatchers.

Pipeline : `_compile_question(question, ctx, classification)` → `SYMBOLIC_QUERY_SYSTEM_PROMPT` (avec hint `sub_kind` et `reasoning` du classifieur en préambule) → code VVL généré → eval → wrap.

Règle 5bis du prompt (phase A1-fix) : *"Pour tout problème ayant une formule canonique de manuel, utilise-la ; ne bricole pas."* Cette règle est accompagnée d'un contre-exemple explicite sur l'annuité d'amortissement vs intérêt simple. Le contexte `sub_kind` du classifieur (ex. `amortization`, `npv`, `compound_interest`) permet au LLM code-gen de choisir la bonne formule de manuel.

Opérateur de puissance : utiliser `(a ^ b)` (droite-associatif, précédence au-dessus de `*` et `/`) ou la forme préfixe `[pow base exp]`. **Jamais** `**` : ce dernier est réservé à l'intersection sémantique de Zalta entre concepts ; entre nombres il tombe en fallback multiplication silencieuse (cf. § 10.8).
Exemple : `[§§? "5000 EUR à 4% par an sur 10 ans en intérêts composés"]` → `5000 × (1 + 0.04) ^ 10 = 7401.22`.

10.4.2. constraint_optim — CP-SAT

Quand : minimiser ou maximiser un objectif sous contraintes dures (scheduling, routing, assignment, packing).

Pipeline : `_dispatch_optim` → `[optimize "NL" name]` (compile NL en modèle CP-SAT via gpt-5.3-codex) → `[optimize:solve name]` → `[optimize:report]` ou `[optimize:why-infeasible]` → wrap.
Le compilateur applique deux règles strictes : **§0 MINIMALITY** et **§0bis UNDER-SPECIFIED**. Il refuse d'inventer des dimensions (slots, capacités, durées) absentes de la question — au lieu de produire un modèle explosif, il surface une demande de clarification. C'est ce qui le distingue d'un wrapper LLM naïf qui inventerait silencieusement les contraintes manquantes.

Trace : le modèle CP-SAT (nombre de variables et de contraintes), le verdict (`OPTIMAL` / `INFEASIBLE`), les bindings bruts, et l'explication NL sont tous loggés dans la trace structurée.

10.4.3. relational_query — base Prolog

Quand : questions en qui / quel / quelle / which sur des relations entre entités nommées (généalogie, qualifications, propriétés croisées).

Pipeline : `_dispatch_relational` → un LLM avec prompt `RELATIONAL_SYSTEM` génère du VVL utilisant `[kb set]` + `[facts]` + `[rules]` + `[query VVL]` → eval → bindings bruts → un second LLM *"explique les bindings en NL"* → wrap.
KB éphémère par défaut : un KB est créé par question puis détruit. Pour réutiliser une KB persistante d'une question à l'autre, deux voies existent : la branche `expert_system` (cf. § 10.4.8) ou `[ontology:askMagic]` en mode manuel.

10.4.4. compound — multi-solveur avec data-flow

Quand : la question chaîne deux ou plus sous-tâches avec flux de données explicite OU agrège plusieurs instances du même domaine.

Pipeline : `_dispatch_compound` → `decompose(question)` (un appel LLM avec prompt `DECOMPOSE_SYSTEM`, caché) qui retourne une liste `[{name, sub_q, family, sub_kind?, depends_on}, ...]` → `run_sub_tasks` exécute en ordre topologique avec substitution `{prev_name}` dans les `sub_q` suivants → `compose` (un appel LLM) effectue la synthèse NL finale → wrap avec assumptions préfixées `<sub_task>.*`.
Deux variantes typiques. **Multi-famille** : extraction (`extract X`) puis calcul (`calc Y`) — par exemple `structured_extraction → arithmetic_closed`. **Same-family multi-instance** (apport de la phase COMPOUND×EXPERT) : `triage Marie ; triage Paul → rank` — deux fois `expert_system` puis `compose`.

Limites : 2 à 4 sous-tâches au maximum, profondeur de dépendance limitée à 2, pas de boucle ni d'itération native (utiliser `[reduce]` ou une boucle VVL manuelle hors décomposition).

10.4.5. rule_induction — induction de règle

Quand : la question fournit des exemples positifs et négatifs étiquetés et demande une règle généralisante.

Pipeline : `_dispatch_induction` → un LLM avec prompt `INDUCTION_SYSTEM` parse les exemples, induit la règle en NL, et (optionnellement) produit aussi une forme Prolog → wrap.

Audit spécifique : les champs `induced_rule_nl`, `induced_rule_prolog`, `positives_count`, `negatives_count`, et `induction_confidence` sont foldés dans l'Approximation. **À venir** : délégation à `[ilp:learn]` (Popper LFF) quand le LLM produit une bias spec exploitable.

10.4.6. factual_retrieval — lookup factuel

Quand : capitale, formule, définition, date, constante — un seul fait public sans calcul.

Pipeline : `_dispatch_factual` → LLM avec prompt `FACTUAL_SYSTEM` (réponse JSON courte avec `kb_hint` et `confidence`) → si `kb_hint` est non vide, tentative de `[ragAsk kb_hint question]` avec fallback LLM-direct → wrap.

Audit spécifique : l'hypothèse `world_knowledge_caveat` est ajoutée quand la réponse vient du LLM seul ("réponse possiblement périmée"). Un champ `freshness` apparaît dans `$r.warnings` quand le LLM lui-même signale une incertitude temporelle (cf. commit 2305a76).

10.4.7. structured_extraction — JSON typé

Quand : payload texte plus liste de champs à extraire.

Pipeline : `_dispatch_extraction` → LLM avec prompt `EXTRACTION_SYSTEM` parse en JSON avec `fields` / `missing` / `schema` / `confidence` → wrap. Les champs absents sont listés explicitement dans `missing` et jamais inventés.

Schéma utilisateur explicite (commit 4a6592f) : on peut passer `[§§? "..." [JSON '["champ1", "champ2"]']]` pour forcer un schéma spécifique. Le wrapper produit alors un champ `schema_mismatches` qui liste les divergences entre le schéma demandé et ce que le LLM a réellement pu extraire.

10.4.8. expert_system — dispatch dynamique (Phase ROUTE-EXPERT)

Quand : `[expert "domain+rules NL" "Name"]` a été appelé dans la session ET la question matche le domaine d'un expert enregistré.

C'est la famille la plus inhabituelle des huit, car elle exploite un mécanisme *introduit dans la session par l'utilisateur lui-même*. L'appel à `[expert]` bootstrap un keyword `<Name>` qui sait évaluer son domaine ; le router se met automatiquement à savoir router vers ce keyword. Le pipeline détaillé est le suivant.

Étape	Mécanisme
1. Injection dynamique au classifier	Le prompt système reçoit la liste formatée des experts via `format_registered_experts(ctx)` (nom + main_class + description tronquée).
2. Classification dispatch	Si la question matche un expert, le LLM retourne `{family: "expert_system", sub_kind: "<ExpertName>"}`.
3. Lookup registry	`_dispatch_expert_system` interroge `ctx._expert_registry[ExpertName]` et construit `[<ExpertName> "<question>"]`.
4. Eval du keyword expert	Le keyword fait : `[ontology:fromText $<Name>_schema "<MainClass>" "<input>"]` + `[ontology:commit $instance]` + `[ontology:infer $<Name>_schema]` → 1 LLM (fromText) + 0 LLM pour l'inférence (règles fuzzy déterministes).
5. Wrapping en Approximation	`router_family=expert_system`, `expert_name`, `expert_schema`, `expert_main_class`, `inference_engine=fuzzy_ontology`, `extraction_step` (LLM, weight 1.0).

Figure 10.3 — *Pipeline détaillé de la famille expert_system : injection au classifier, dispatch, lookup, eval, wrap.*

Normalisation enum (commit f676c55) : quand un champ a une contrainte enum (déclarée via `[option "enum" "v1,v2,v3"]` ou détectée heuristiquement depuis la description `(v1/v2/v3)`), la valeur extraite est normalisée vers la valeur canonique via trois niveaux successifs : **(1)** *Cheap* (strip accents + lowercase + substring exact), **(2)** *Stem* (préfixe commun d'au moins `len(option) - 2` caractères, ce qui rattrape *"état confus"* → `confuse`), **(3)** Zalta `~~` (appel `semantic_binop("~~")` pairwise contre chaque option, pick max ≥ `0.55`).
Bust de cache classifier : quand le registry des experts change, le hash de classification inclut `<question>##<expert-list-signature>` — cela garantit qu'un nouvel expert enregistré est immédiatement visible du router sans nécessité de purge manuelle (cf. § 10.7).

10.5. L'écosystème des experts dynamiques

La famille `expert_system` s'appuie sur six builtins de la famille `[expert:*]` qui gèrent le cycle de vie d'un expert, depuis son bootstrap jusqu'à sa persistance disque.

Builtin	Effet
`[expert "domain+rules NL" "Name"]`	Bootstrap one-shot : 2 appels LLM → ontologie + règles fuzzy + keyword `<Name>` enregistré dans `ctx._expert_registry`.
`[Name "input NL"]`	Le keyword auto-généré : fromText (1 LLM) + commit + infer (0 LLM) → instance ontologie.
`[expert:active]`	JSON list des experts in-memory dans la session courante (ce que le router peut dispatcher).
`[expert:save "Name"]`	Persiste l'expert sur disque (workspace jail).
`[expert:load "Name"]`	Restaure un expert sauvegardé → in-memory + keyword.
`[expert:list]`	JSON list des experts persistés sur disque (≠ in-memory).

Figure 10.4 — *Les six builtins de la famille [expert:*].*

Distinction importante : `[expert:active]` liste les experts *in-memory* (donc ce que `[§§?]` peut router), tandis que `[expert:list]` liste les experts persistés sur *disque* via `:save` (donc ce qu'on peut `:load`). Les deux vues peuvent diverger : un expert créé mais pas sauvegardé n'est pas dans `list` ; un expert sauvegardé puis serveur redémarré n'est pas dans `active` tant qu'on n'a pas fait `:load`.

Exemple complet d'un expert métier

```
001  // 1. Bootstrap (une fois, ~30s)
002  [expert "Domaine evaluation risque credit. Dossier client : nom, revenu,
003   score_credit (eleve/moyen/faible), anciennete (longue/moyenne/recente),
004   decision. ..." "Scoring"]
005
006  // 2. Appel direct (1 LLM pour fromText + 0 LLM inference)
007  [Scoring "Alice Dubois, revenu 4500, score eleve, anciennete longue"]
008  // → <Ontology:DossierClient [..., decision=approuve]>
009
010  // 3. Auto-routage via [§§?] (router découvre l'expert)
011  $r = [§§? "Évalue le dossier de Alice Dubois, revenu 4500, score élevé, ancienneté
longue"]
012  $r.value          // → {..., 'decision': 'approuve'}
```

Figure 10.5 — *Cycle complet d'un expert métier : bootstrap, appel direct nominatif, puis auto-routage par §§?.*

10.6. L'Approximation : wrapper d'audit en profondeur

Tout retour de `[§§?]` est une `VVL_Approximation` (sauf valeurs purement scalaires non-tracées). C'est l'objet pivot qui rend le système **auditable de bout en bout** : il porte non seulement la valeur, mais aussi la confiance qualitative, la fourchette estimée, la liste des hypothèses avec leur source, le code source ayant produit la réponse, et les warnings non-bloquants. C'est ce qui distingue un système neuro-symbolique sérieux d'un wrapper LLM qui produit une réponse confiante sans traçabilité.

L'API publique de l'Approximation

Accès	Effet
`$r.value`	Valeur principale (nombre, string NL, ou instance ontology).
`$r.confidence`	`"exact"` / `"high"` / `"moderate"` / `"low"` selon `Σ weight_llm`.
`$r.range`	Fourchette estimée (chaîne `"low - high"`).

Accès	Effet
`$r.trace`	Dict structuré (value, confidence, n_llm, w_llm, range, assumptions, ...).
`$r.warnings`	Liste de warnings non-bloquants (freshness, schema mismatch, ...).
`$r.sensitivity`	Per-LLM-source perturbation ±10 %, mesure d'impact.
`$r.revise`	Re-run avec overrides : `[approximation:revise $r [JSON '{"taux": 0.04}']]`.

Figure 10.6 — *L'API de la VVL_Approximation.*

Les hypothèses (assumptions) et leur pondération

Chaque hypothèse retenue lors du calcul a la forme `{name, value, source, concept, via, weight}`. Le champ `source` prend l'une des quatre valeurs `user` / `fact` / `LLM` / `computation` et pondère le calcul de confiance. Le champ `via` est un tag de sous-système (`router`, `optimize_cp_sat`, `prolog`, `expert_system`, `llm_formula`, `fuzzy_ontology`, etc.) qui permet de retracer quel composant a contribué quelle hypothèse.

Les bandes de confiance

Le passage de `Σ weight_llm` à un label qualitatif suit une grille simple :

Condition	Label
`w_llm == 0`	`exact`
`w_llm ≤ 1.0` et `w_fact > 0`	`high`
`w_llm ≤ 2.0`	`moderate`
Sinon	`low`

Figure 10.7 — *Grille de calcul du label de confiance qualitatif.*

Le cas `llm_formula` (commit 27ad88e) mérite une attention particulière : quand le LLM choisit une formule via une Concept method, l'audit tag l'hypothèse avec `source=LLM, weight=1.5` — le résultat atterrit donc par défaut en `moderate`, pas en `exact`. C'est délibéré : on évite la fausse confiance *"exact"* sur une formule bricolée par le LLM, même si la formule semble correcte syntaxiquement.

10.7. Configuration, comportement défensif, caches

Variables d'environnement et seuils

Variable	Défaut	Effet
`ROUTER_MIN_CONFIDENCE`	`0.55`	Seuil sous lequel une famille non-arith déclenche le gate clarification.
`COMPOUND_MIN_CONFIDENCE`	`0.70`	Seuil spécifique à compound (plus strict — compound est coûteux).

Figure 10.8 — *Les deux seuils de confiance configurables.*

Ces seuils peuvent être ajustés au runtime via `os.environ` (avant import) ou via `[env:set ...]` si l'instance le supporte. En production, augmenter `ROUTER_MIN_CONFIDENCE` à 0,70 réduit le bruit de dispatch mais augmente le taux de clarifications demandées à l'utilisateur.

Builtins de debug du routage

Deux builtins permettent d'inspecter le routage sans déclencher le calcul. `[resolve:classify "..."]` renvoie le verdict du classifieur *sans* dispatcher : utile pour comprendre pourquoi une question est routée comme elle l'est. `[resolve:trace $r]` (commit 22d9e23) renvoie la trace structurée JSON complète après un appel `[§§?]` : classification, events par famille, résultat final. Cette dernière est distincte du mode `:t` du REPL qui imprime au fil de l'eau sur stdout.

```
001  [resolve:classify "Évalue Marie Lefranc, 78 ans, fièvre 39.5..."]
002  // → {family: "expert_system", sub_kind: "Triage", confidence: 0.96, ...}
003
004  $r = [§§? "..."]
005  [resolve:trace $r]
006  // → JSON {classification, events: [...], result: {...}}
```

Comportement défensif — fallbacks

Le résolveur ne lève *jamais* d'exception non capturée. À chaque niveau, un fallback est prévu, ce qui garantit que **toutes les questions historiques fonctionnent** — les nouvelles familles sont un sur-ensemble strictement compatible.

Échec	Fallback
Question vide ou trop courte	`arithmetic_closed` (skip LLM classify).
LLM classifier silencieux ou timeout	`arithmetic_closed`.
Famille inconnue retournée par LLM	`arithmetic_closed`.
Spécialiste lève une exception	`arithmetic_closed` (fallback callback).
Code généré (arith / relational) ne parse pas ou lève à l'eval	Retry compile avec feedback d'erreur (max 2) puis fallback `_failure_approximation`.
Décomposeur retourne 0 sous-tâche ou une sub-task plante	Fallback `arithmetic_closed`.
Expert non trouvé dans registry / keyword eval rate	Fallback `arithmetic_closed`.
Confiance inférieure au seuil	Gate clarification (Approximation `low` avec `clarification_requested`).

Figure 10.9 — *Cascade des fallbacks défensifs du résolveur.*

Caches

Cache	Clé	Mécanisme de bust
Classifier	`sha256(question + experts_signature)`	Auto sur changement de registry, ou restart serveur.
Decomposer	Identique au classifier	Identique.
Semantic binops (`~~`, `**`, `//`, `%`)	`(op, hash(left), hash(right))`	Cache disque persistant, vider via `[semantic:clearCache]`.
LLM réplique	Géré au niveau du provider	n/a.

Figure 10.10 — *Les quatre caches du système de routage.*

10.8. Opérateurs et builtins de support

L'opérateur de puissance ^

Introduit par le commit 109ae97, l'opérateur `^` est numérique, droite-associatif, avec une précédence au-dessus de `*` et `/`. Il existe aussi une forme préfixe `[^ a b]` et un alias legacy `[pow base exp]`.

```
001  (2 ^ 10)              = 1024
002  (2 ^ 3 ^ 2)           = 512        // = 2^(3^2)
003  (2 + 3 ^ 2)           = 11         // = 2 + 9
004  ((1 + 0.04) ^ 10)     = 1.4802
005  [^ 2 10]              = 1024       // forme prefix
006  [pow 2 10]            = 1024.0     // alias legacy
```

Attention : `**` reste l'opérateur d'intersection sémantique de Zalta entre concepts ; entre nombres, il tombe en fallback multiplication (comportement legacy conservé pour rétrocompatibilité). *Toujours utiliser `^` pour la puissance* — jamais `**`, sous peine de calculer silencieusement une multiplication à la place.

Builtins mathématiques

Le commit cffa62d a ajouté un jeu de builtins mathématiques de base, qui complètent `^` et sont indispensables pour les formules de manuel utilisées par `arithmetic_closed`.

Builtin	Effet
`[sqrt x]`	√x
`[pow base exp]`	base^exp
`[floor x]`	math.floor(x)
`[ceil x]`	math.ceil(x)
`[round x [ndigits]]`	round(x, ndigits=0)
`[abs x]`	|x|

Figure 10.11 — *Les six builtins mathématiques de base.*

10.9. Mode trace `:t` et debug du routage

Le mode trace `:t` du REPL active une trace pas-à-pas du résolveur. Chaque dispatcher non-arithmétique émet des lignes préfixées `[router/<family>]` qui suivent les étapes de la résolution. Le format est symétrique entre toutes les familles : `constraint_optim`, `relational_query`, `compound`, `rule_induction`, `factual_retrieval`, `structured_extraction`, et `expert_system`. En mode trace OFF, tous les `ctx.log()` sont des no-op à coût zéro.

```
001  [router/expert_system] Step 1/3 — Looking up expert in registry...
002  [router/expert_system] matched expert='Scoring' (schema=risque_credit,
class=DossierClient)
003  [router/expert_system] Step 2/3 — Evaluating [Scoring "..."] (fromText + commit + infer)
004  [router/expert_system] VVL call :
005  [router/expert_system] ──────────────────────────────────────────
006  [router/expert_system]    1  [Scoring "Alice Dubois, revenu 4500, score eleve, anciennete
longue"]
007  [router/expert_system] ──────────────────────────────────────────
008  [router/expert_system] Step 3/3 — Wrapping the classified instance as Approximation
009  [router/expert_system] Resulting instance :
010  [router/expert_system] ──────────────────────────────────────────
011  [router/expert_system]    1  <Ontology:DossierClient [..., decision=approuve]>
012  [router/expert_system] ──────────────────────────────────────────
```

Figure 10.12 — *Exemple de trace expert_system avec le préfixe [router/<family>] et les step markers.*

10.10. [generic] : capitaliser les solutions résolues

La commande `[generic "question"]` accomplit en un seul appel ce que §§ fait — *plus* l'industrialisation. Elle compile la question, mais retient la solution dans un catalogue persistant sous forme de **keyword nommé**. La signature canonique est `[generic "question"]` ou `[generic "question" "nom_explicite"]` ; un troisième argument JSON peut spécifier des options.
Le pipeline d'industrialisation comporte douze étapes que le développeur ne voit pas, mais qu'il est essentiel de comprendre pour utiliser la commande à bon escient. Chaque étape contribue à la transformation d'une *solution* en *fonction réutilisable*.

#	Étape	Rôle
1	Compilation §§	Produit le code VVL résolvant la question.
2	Évaluation	Exécute le code et obtient le résultat témoin.
3	Détection des ancres	Trouve les `[@ "..." "lazy"]` du code généré.
4	Inférence des rôles	Détecte des patterns (référence / cible, sujet, input).
5	Inférence du contrat	Liste les propriétés requises par paramètre.
6	Abstraction	Remplace les littéraux par `[@ @1 "lazy"]`, `[@ @2 "lazy"]`, ...
7	Naming automatique	Devine un nom métier sensé à partir de la question et du code.
8	Génération du keyword	Construit le code VVL final avec retour contractuel.
9	Tests d'amorçage	Génère le cas exact + cas inversé (si symétrique).
10	Tests compatibles	Cherche d'autres concepts ayant les mêmes propriétés requises.
11	Pseudo-code Markdown	Génère une fiche lisible pour humains.
12	Persistance + évaluation	Écrit catalog.json, tests/, pseudocode/, et charge le keyword.

Figure 10.13 — *Les douze étapes du pipeline d'industrialisation [generic].*

```
001  // Premiere resolution : la question est posee comme question
002  [§§ "Si Mozart avait vecu aussi vieux que Beethoven, en quelle annee serait-il mort ?"]
003  // => 1812
004
005  // Industrialisation de la solution en keyword nomme
006  [generic "Si Mozart avait vecu aussi vieux que Beethoven, en quelle annee serait-il mort
?"]
007  // Catalog : ajoute la fonction "death-year-if-lived-like" avec :
008  //   - parametres : @1 (target = annee_naissance), @2 (annee_naissance + annee_deces)
009  //   - tests d'amorcage : cas exact (Mozart, Beethoven) + cas inverse
010  //   - pseudocode Markdown
011  //   - keyword charge dans le contexte courant
012
013  // Reutilisation de la fonction sur d'autres concepts
014  [death-year-if-lived-like "Schubert" "Bach"]    // => 1862
015  [death-year-if-lived-like "Mendelssohn" "Wagner"] // => 1879
```

Le **naming automatique** mérite un commentaire car il est l'un des aspects les plus surprenants. Au lieu de générer un nom syntaxique illisible (`gen_42`), `_derive_keyword_name` reconnaît des patterns métier et propose des noms évocateurs.

Pattern détecté	Exemple de question	Nom dérivé
2 paramètres avec annee_naissance + annee_deces + + et −	« Mozart si vécu comme Beethoven »	`death-year-if-lived-like`
2 paramètres + token compare/plus	« Qui est plus massif, Soleil ou Jupiter ? »	`compare-mass-kg-between-two-concepts`
1 paramètre + 1 propriété requise	« Quelle est la masse de Saturne ? »	`get-mass-kg-of-concept`
Calcul numérique de plusieurs propriétés	« Densité d'un acier 316L ? »	`compute-densite-resistance-from-concept`

Pattern détecté	Exemple de question	Nom dérivé
Sinon (fallback lexical)	« température de la Terre »	`temperature-de-la-terre`

Figure 10.14 — *Quelques règles de dérivation automatique de nom de keyword.*

10.11. L'écosystème du catalogue [generic]

Le catalogue créé par `[generic]` n'est pas une boîte noire : c'est un **écosystème de fichiers explorable** et **manipulable** par douze builtins dédiés. La racine est `lib/generic/` à la base du workspace courant ; elle contient un fichier `catalog.json` qui liste toutes les entrées, et deux sous-dossiers `tests/` et `pseudocode/` qui contiennent les artefacts détaillés de chaque entrée.

```
001  [generic:list]                              // toutes les fonctions
002  [generic:list "draft"]                      // filtrer par statut
003  [generic:info "death-year-if-lived-like"]   // fiche complete
004  [generic:code "death-year-if-lived-like"]   // code source du keyword
005  [generic:contract "death-year-if-lived-like"] // parametres + contrat
006  [generic:tests "death-year-if-lived-like"]  // tests d'amorcage
007  [generic:pseudocode "death-year-if-lived-like"]
008  [generic:load "death-year-if-lived-like"]   // charge en session
009  [generic:validate "death-year-if-lived-like"] // re-execute les tests
010  [generic:evolve "death-year-if-lived-like"]  // regenere depuis la question
011  [generic:forget "death-year-if-lived-like"]  // supprime + artefacts
```

Builtin	Rôle
`[generic "question" "nom"?]`	Industrialise une solution en keyword catalogué.
`[generic:list status?]`	Liste les fonctions du catalogue (filtrable par statut).
`[generic:info nom]`	Retourne la fiche complète (params, contract, tests, validator).
`[generic:code nom]`	Retourne le code source VVL du keyword.
`[generic:contract nom]`	Retourne les paramètres et le contrat de sortie.
`[generic:tests nom]`	Retourne les tests mémorisés.
`[generic:pseudocode nom]`	Régénère le pseudo-code Markdown.
`[generic:load nom]`	Charge le keyword dans la session courante.
`[generic:validate nom]`	Exécute les tests et met à jour le score de validation.
`[generic:evolve nom]`	Régénère l'entrée à partir de la question source.
`[generic:forget nom]`	Supprime l'entrée et tous ses artefacts.
`[generic:invoke nom args...]`	Exécute en préservant les approximations (helper interne).

Figure 10.15 — *Les douze builtins de l'écosystème [generic].*

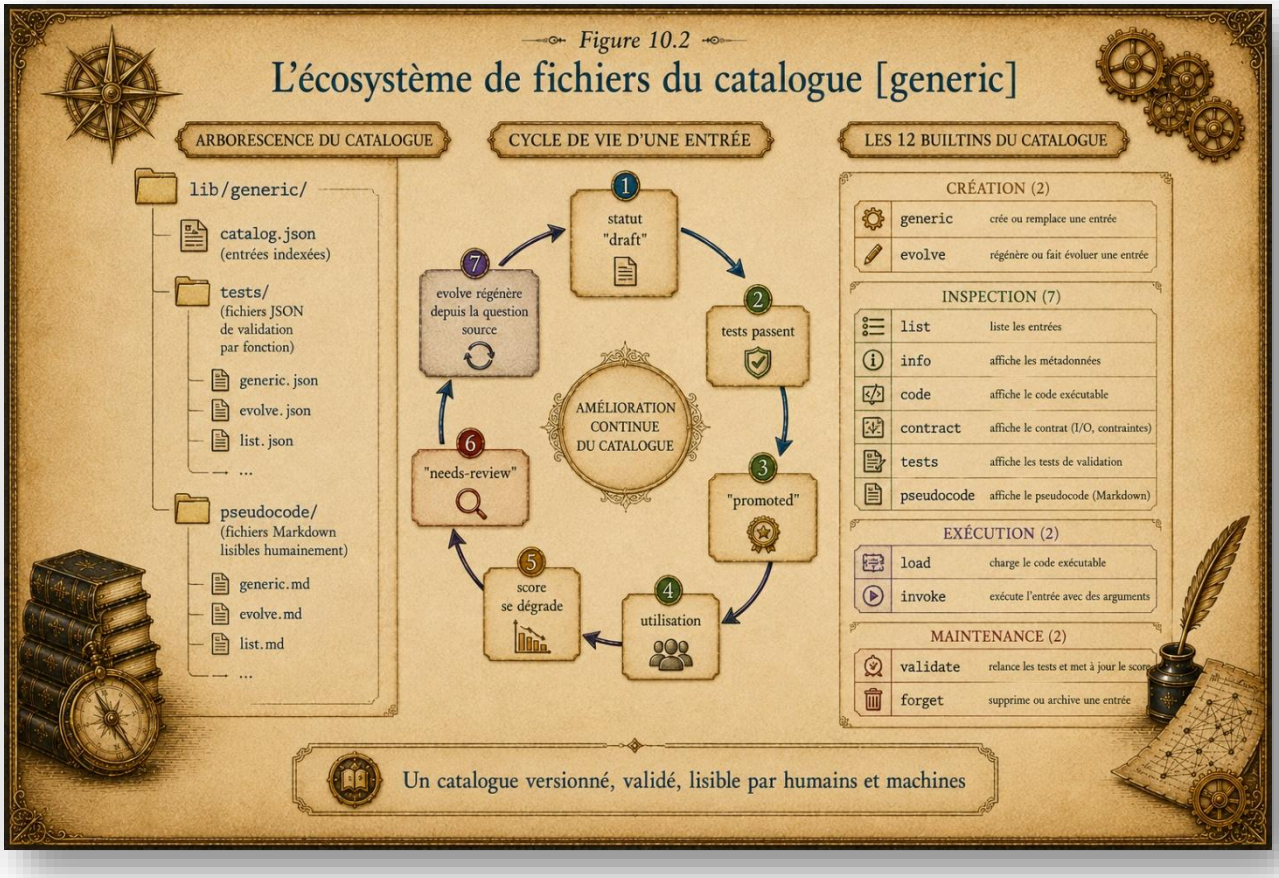


Figure 10.16 — *Le cycle de vie d'une entrée [generic] : draft, promoted, needs-review, evolved.*

10.12. Limites connues

Toute architecture porte ses limites ; il vaut mieux les expliciter pour orienter l'usage.

Limite	Détail
Décompose limité à 4 sous-tâches (cap dur)	Au-delà, le LLM decomposer émet une seule sous-tâche de clarification.
Pas d'itération native dans compound	Pour répéter un expert sur N entités hors décomposition, utiliser une boucle VVL manuelle avec `[map]` ou `[reduce]`.
Registry d'experts en mémoire	Limité par la fenêtre de contexte du classifier. Au-delà de ~15 experts simultanés, le bloc `REGISTERED EXPERTS` injecté dans le user message devient long ; un pré-filtrage par k-NN sur les descriptions devient utile.
Fuzzy rule generation (`[fuzzy:rule]`)	Les conditions complexes AND / OR parenthésées sont parfois mal traduites par le LLM en règle ontology. Vérifier `$<Name>_schema.rules` si une décision attendue ne se déclenche pas.
Ontology dump truncation	`<Ontology:Class [...]>` coupe après 5 champs avec `...`. La valeur n'est PAS perdue (visible via `$instance.field` ou `[approximation:trace]`).

Limite	Détail
Push gitlab interne	SSL cert problem sur certains réseaux, workaround `git -c http.sslVerify=false push` (à utiliser en one-shot, ne pas changer la config persistante).

Figure 10.17 — *Les limites connues du résolveur dans la version courante.*

Exemples complets de bout en bout

Pour fixer les idées, trois exemples complets qui exercent trois familles différentes.
Calcul financier canonique :

```
$r = [§§? "Calcule la mensualité d'un crédit de 250000 EUR sur 20 ans à 3,5% taux fixe."]
$r.value         // => 1449.90 (formule d'amortissement canonique)
$r.confidence    // => "moderate" (la formule est choisie par le LLM)
[approximation:trace $r]
```

Question composée avec data-flow :

```
$r = [§§? "Extrais le nombre d'employés de 'Acme Corp compte 87 employés',
           puis calcule combien d'équipes de 5 personnes on peut former."]
$r.value         // => "17 équipes de 5"
```

Comparaison multi-instances (compound × expert) :

```
[expert "Domaine triage medical urgences. Patient : nom, age,
 fievre_celsius, douleur (faible/moderee/intense),
 conscience (alerte/confuse/inconsciente), priorite. ..." "Triage"]

$r = [§§? "Parmi ces deux patients quel est le plus prioritaire :
           (Marie Lefranc, 78 ans, fievre 39.5, douleur intense, etat confus)
           et (Paul Durand, 65 ans, fievre 38.7, douleur modérée, alerte) ?"]
$r.value         // => "Marie (rouge) est plus prioritaire que Paul (orange)..."
```

10.13. Synthèse : les quatre rôles du LLM dans VVL

Avec ce chapitre, la Partie III du livre est complète. Le tableau suivant offre une vue synthétique des **quatre rôles que le LLM joue dans VVL**, chacun correspondant à un chapitre de la Partie III. Ce tableau résume aussi pourquoi VVL n'est pas un wrapper d'API LLM mais un véritable langage neuro-symbolique : le même modèle de langage est successivement opérateur, générateur, fournisseur d'objets et compilateur-routeur, sans qu'aucune de ces facettes ne nécessite d'API supplémentaire ni de pipeline d'intégration.

Rôle	Chapitre	Syntaxe	Sortie	Usage typique
Opérateur d'expression	7	`[§ "..."]` dans une expression	BOOLEAN, STRING, SEMANTIC	Comparaisons floues, algèbre conceptuelle.
Générateur de fonction	8	`$data @ [§ "..."]`	LAMBDA validée	Filtres et transformations.
Fournisseur d'objet	9	`[@ "nom"]`	VVL_ConceptEntity	Réification, lazy-load, opérateurs Zalta.

Rôle	Chapitre	Syntaxe	Sortie	Usage typique
Compilateur-routeur de question	10	`[§§ "..."]` / `[§§? "..."]` / `[generic "..."]`	Approximation auditable ou keyword	Questions NL, 8 familles, industrialisation.

Figure 10.18 — *Synthèse des quatre rôles du LLM dans VVL.*

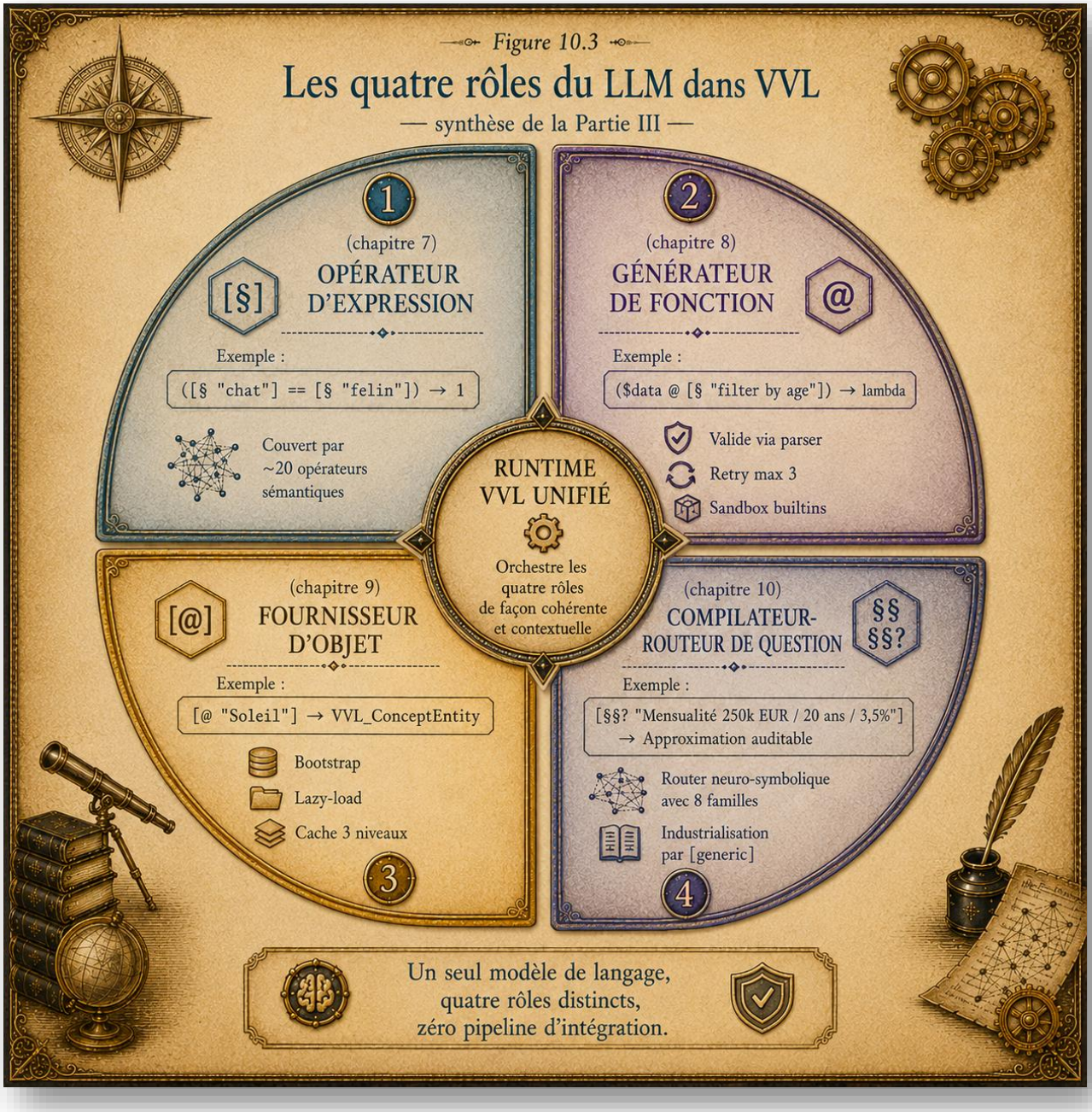


Figure 10.19 — *Le LLM dans VVL : opérateur, générateur, fournisseur d'objet, compilateur-routeur.*

La **Partie IV** qui s'ouvre au chapitre 11 marque un changement de polarité. Là où la Partie III a montré le versant **neuronal** du langage — le LLM intégré comme co-processeur —, la Partie IV explore le versant **symbolique** : la base Prolog (chapitre 11), le système ontologique (chapitre 12), et les opérateurs Zalta (chapitre 13) qui réalisent le pont entre les deux mondes. Ce mouvement de balancier

entre neuronal et symbolique, appelé *oscillation neuro-symbolique* au chapitre 18, est la signature de la programmation neuro-symbolique authentique. Les deux versants ne sont pas des bibliothèques juxtaposées ; ils sont les deux faces d'un même langage, et le programme VVL passe de l'un à l'autre par simple changement de syntaxe.

Partie IV — Le pont symbolique

Chapitres 11 à 13 — base Prolog, système ontologique, algèbre de Zalta

11. La base de connaissances Prolog

Mots-clés : Prolog, faits, règles, requêtes, unification, backtracking, kb, query, prologSolutions, obj-assert, pont OO-Prolog, raisonnement déductif, déclaratif

La Partie IV du livre s'ouvre par un changement de polarité. Là où la Partie III a montré le versant **neuronal** de VVL — le LLM intégré comme co-processeur sous quatre rôles distincts —, cette Partie IV explore le versant **symbolique**. Le chapitre 11 introduit le moteur Prolog intégré à VVL, qui est l'un des trois piliers de la couche symbolique avec le système ontologique (chapitre 12) et les opérateurs Zalta (chapitre 13).

Pourquoi un livre sur la programmation neuro-symbolique consacre-t-il un chapitre entier à Prolog, langage inventé en 1972 par Alain Colmerauer à Marseille ? La réponse est triple. **Premièrement**, Prolog est l'expression la plus mûre et la plus rigoureuse de la programmation déclarative — décrire *ce qui est vrai* plutôt que *comment résoudre*. **Deuxièmement**, son moteur d'unification et de backtracking offre des garanties de correction qu'aucun système probabiliste ne peut fournir : si une requête a une solution, Prolog la trouve ; si elle n'en a pas, Prolog le démontre. **Troisièmement**, Prolog est complémentaire au LLM : là où le LLM excelle à *reconnaître* et à *interpoler*, Prolog excelle à *déduire* et à *prouver*. La force de VVL est de réunir ces deux capacités dans le même runtime.

Le chapitre suit la séquence suivante : la motivation et le triangle VVL (11.1), la grammaire Prolog de base — faits, règles, requêtes (11.2), l'intégration depuis VVL via `kb`, `facts`, `rules` (11.3), les six modes de requête (11.4), le pont OO ↔ Prolog par `obj-assert` qui transforme tout objet VVL en faits interrogeables (11.5), et enfin le cycle de vie complet d'un objet à travers les couches du langage (11.6).

11.1. Le triangle neuro-symbolique de VVL

Pour comprendre la place de Prolog dans VVL, il faut voir le langage comme un **triangle à trois sommets** dont chacun apporte une capacité que les deux autres ne peuvent pas fournir. **Le sommet neuronal**, occupé par le LLM, apporte la *reconnaissance approximative* — la capacité à comprendre du langage naturel, à interpoler entre concepts, à produire du contenu plausible. **Le sommet impératif**, occupé par le système OO et les fonctions, apporte la *manipulation d'état* — la capacité à composer des transformations, à itérer, à modifier des structures de données. **Le sommet symbolique**, occupé par Prolog, apporte la *déduction garantie* — la capacité à prouver des conclusions à partir de prémisses, à chercher exhaustivement les solutions, à vérifier des contraintes.

Aucun de ces trois sommets ne suffit seul pour la programmation neuro-symbolique authentique. Le LLM seul produit des réponses plausibles mais non vérifiables. Le système OO seul ne sait que ce qu'on lui a explicitement codé. Prolog seul est puissant pour la déduction mais incapable de comprendre du langage naturel ou de manipuler des données non structurées. **C'est leur composition** qui produit le neuro-symbolique : le LLM extrait des faits depuis du texte, l'OO les manipule comme objets, Prolog les inscrit comme faits déductibles, et le cycle reprend.

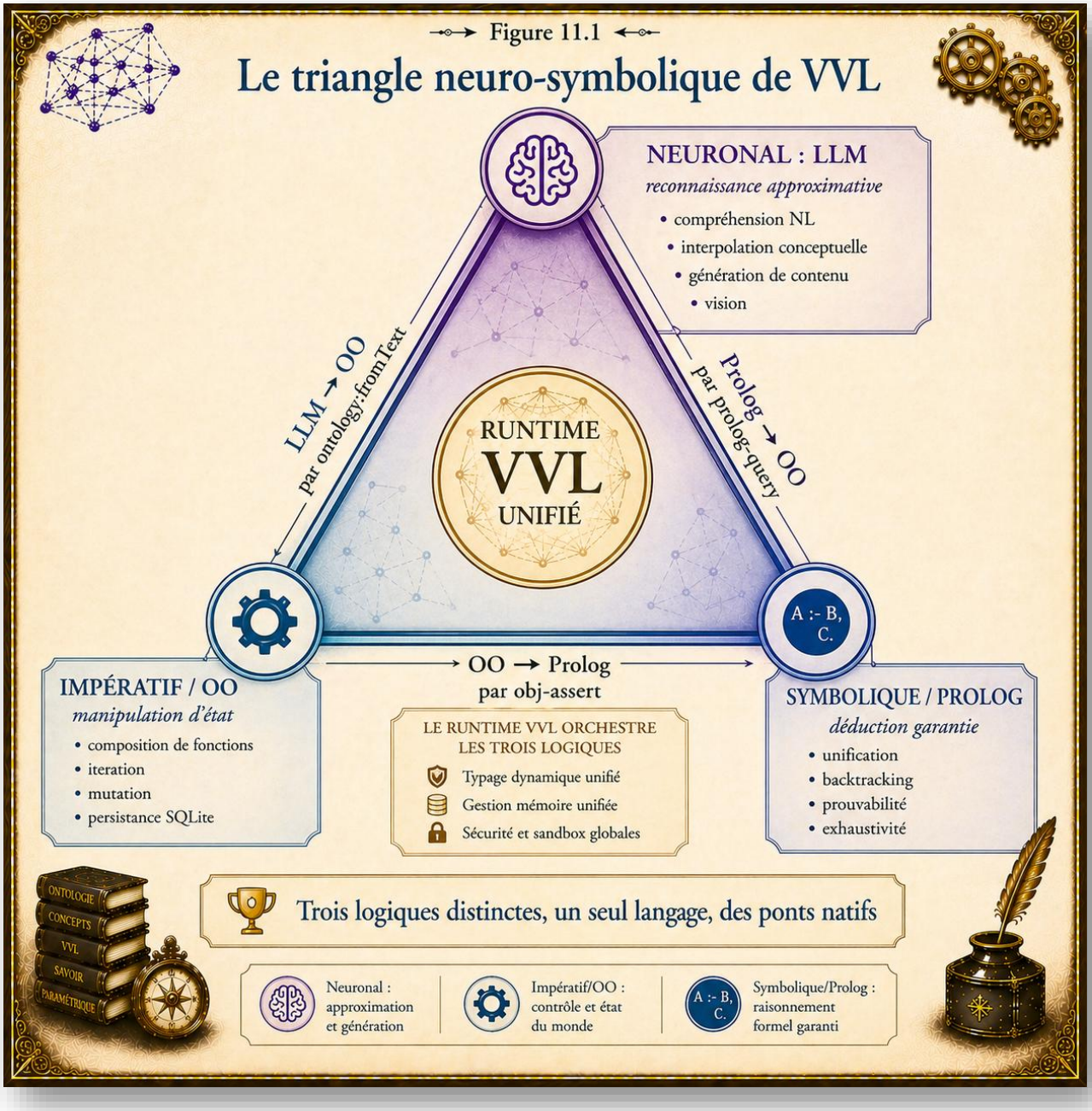


Figure 11.1 — *Le triangle des trois logiques de VVL : neuronal, impératif, symbolique*

Ce chapitre ouvre le côté symbolique du triangle. Prolog n'est pas une dépendance optionnelle de VVL — c'est un **citoyen de premier rang** du langage. Le moteur est intégré au runtime ; il est accessible depuis n'importe quel script via des builtins natifs (`kb`, `facts`, `rules`, `query`, etc.) ; et il dispose d'un mode REPL dédié (`:prolog`) qui transforme l'invite en console Prolog pure pour le développement interactif. Cette intégration profonde est ce qui rend possible le pont `obj-assert` de la section 11.5 — n'importe quel objet VVL peut devenir un fait interrogeable.

11.2. Faits, règles et requêtes : la grammaire de base

Prolog manipule trois constructions fondamentales que tout développeur doit maîtriser avant d'utiliser les builtins VVL d'intégration. Cette section présente la syntaxe Prolog standard, telle qu'elle s'utilise en mode `:prolog` dans le REPL VAILS ou dans un fichier `.pl` chargé via `consult`.

Faits : assertions déclaratives

Un **fait** est une proposition inconditionnellement vraie. La syntaxe est `predicat(arg1, arg2, ...).` avec un point final obligatoire. Les **constantes** (atomes) commencent par une lettre minuscule ; les **variables** par une majuscule (mais les variables n'apparaissent que dans les règles et les requêtes, pas dans les faits ordinaires). Le **prédicat** lui-même commence aussi par une minuscule.

```
% Faits sur les relations familiales
parent(jean, pierre).
parent(jean, marie).
parent(pierre, luc).
parent(marie, lea).

% Faits sur le genre
homme(jean).
homme(pierre).
homme(luc).
femme(marie).
femme(lea).
```

Règles : déductions conditionnelles

Une **règle** est une déduction de la forme « la tête est vraie *si* le corps est vrai ». Le séparateur `:-` se lit « si ». Les variables (en majuscules) sont liées par unification au moment de la résolution. Les conjonctions sont notées par la virgule (« et »), les disjonctions par le point-virgule (« ou ») mais sont rares. La récursion est première classe — une règle peut s'appeler elle-même, ce qui permet d'exprimer des relations transitives ou des structures inductives sans boucle explicite.

```
% Règle simple
pere(X, Y) :- parent(X, Y), homme(X).
mere(X, Y) :- parent(X, Y), femme(X).

% Règle composée
grand_parent(GP, E) :-
    parent(GP, I),
    parent(I, E).

% Règle récursive : ancêtre
ancetre(X, Y) :- parent(X, Y).
ancetre(X, Y) :- parent(X, Z), ancetre(Z, Y).
```

Requêtes : interrogation du monde déclaré

Une **requête** est une question terminée par un point d'interrogation. Le moteur Prolog explore systématiquement l'espace des solutions par **unification** (rendre deux termes identiques en liant les variables) et **backtracking** (revenir en arrière quand une branche échoue, pour essayer la suivante). Si la requête a des variables, le moteur retourne toutes les valeurs qui les satisfont ; si elle n'en a pas, il retourne `Yes` (vrai) ou `No` (non démontrable).

```
% Vérification simple (pas de variable)
parent(jean, pierre)?      % => Yes
parent(luc, jean)?         % => No

% Recherche avec variable : qui sont les enfants de Jean ?
parent(jean, X)?
% => X = pierre
% => X = marie

% Recherche multi-variable : toutes les paires parent-enfant
parent(P, E)?
% => P = jean, E = pierre
% => P = jean, E = marie
% => P = pierre, E = luc
% => P = marie, E = lea

% Recherche avec règle dérivée
grand_parent(jean, X)?
% => X = luc   (jean -> pierre -> luc)
% => X = lea   (jean -> marie -> lea)
```

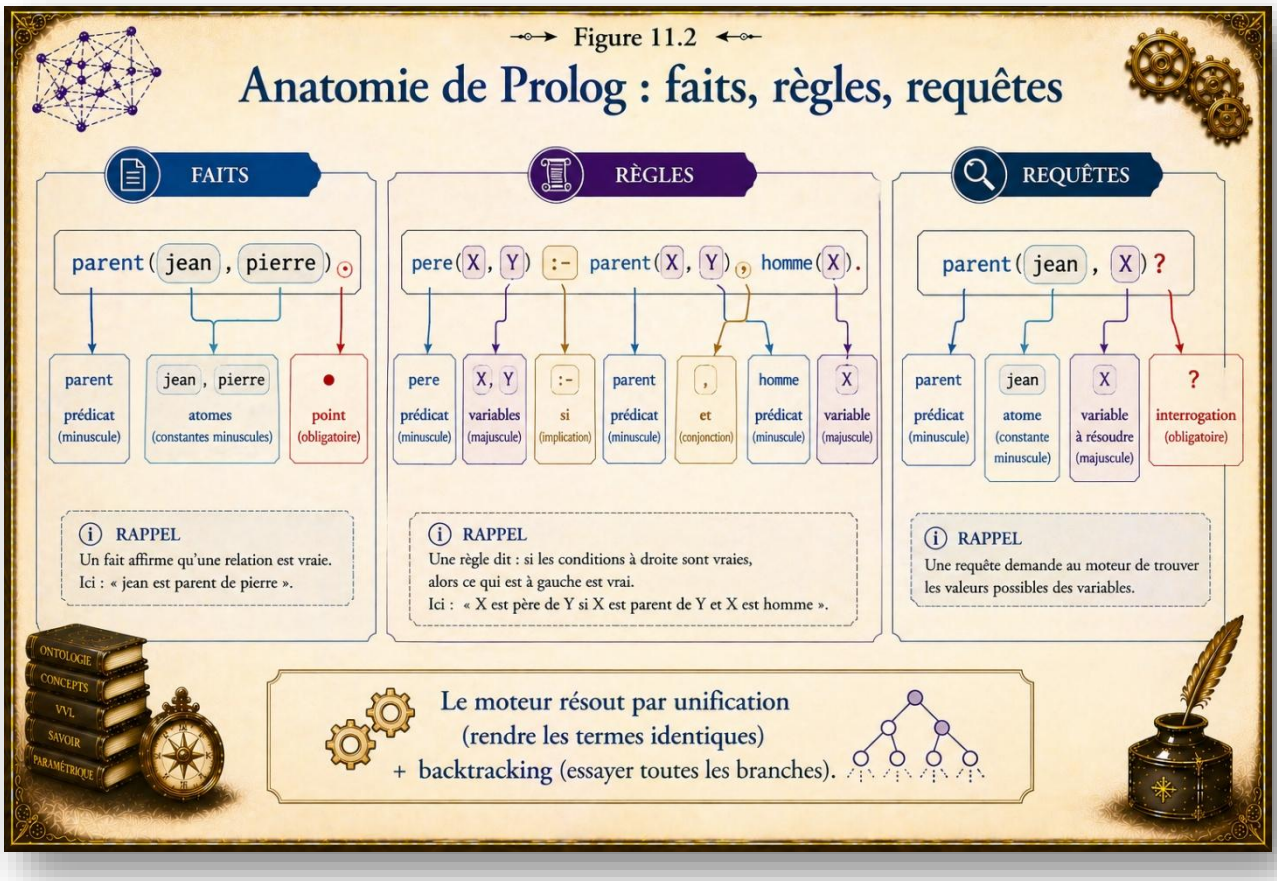


Figure 11.2 — *Les trois constructions fondamentales de Prolog et leur syntaxe*

11.3. Intégration depuis VVL : kb, facts, rules

L'intégration de Prolog dans VVL passe par trois builtins fondamentaux : `kb` pour activer une base de connaissances nommée, `facts` pour y charger des faits, `rules` pour y charger des règles. Une session VVL peut héberger plusieurs bases de connaissances simultanées, chacune avec son propre nom — pratique pour isoler des domaines (familial, médical, financier) qui partagent les mêmes prédicats sans conflit.

```
# Activer une base de connaissances nommee
[kb set "famille"]

# Charger des faits (chaque fait est une chaine Prolog standard)
[facts "famille"
    "parent(jean, pierre)."
    "parent(jean, marie)."
    "parent(pierre, luc)."
    "parent(marie, lea)."
    "homme(jean)."
    "homme(pierre)."
    "homme(luc)."
    "femme(marie)."
    "femme(lea)."
]

# Charger des regles
[rules "famille"
    "grand_parent(GP, E) :- parent(GP, I), parent(I, E)."
    "pere(X, Y) :- parent(X, Y), homme(X)."
    "mere(X, Y) :- parent(X, Y), femme(X)."
    "ancetre(X, Y) :- parent(X, Y)."
    "ancetre(X, Y) :- parent(X, Z), ancetre(Z, Y)."
]
```

Trois remarques importantes sur cette intégration. **Premièrement**, les faits et règles passés à `facts` et `rules` sont des **chaînes Prolog brutes** — la syntaxe à l'intérieur des guillemets est strictement celle de Prolog, pas celle de VVL. Cette séparation maintient la pureté des deux mondes : VVL gère le flux et l'orchestration, Prolog gère la déclaration des connaissances. **Deuxièmement**, les bases sont **nommées** — toute requête doit spécifier le nom de la KB cible. Cela évite les collisions et permet de raisonner sur plusieurs domaines indépendants. **Troisièmement**, les faits et règles **persistent en mémoire** pendant toute la session ; pour les sauvegarder entre sessions, on utilise `ontology:save` qui écrit la KB au format Prolog standard, puis `ontology:load` pour la recharger.

11.4. Six modes de requête depuis VVL

Le pont entre VVL et Prolog offre **six modes de requête** dont chacun retourne le résultat dans un format adapté à un usage particulier. Cette diversité reflète la nature double des résultats Prolog — qui peuvent être un booléen, une valeur unique, une liste de valeurs, ou un tableau de bindings complets — et évite au développeur de réécrire à chaque fois le code de transformation entre formats.

Commande	Description	Retour
[query JSON kb q]	Résultat structuré complet	JSON avec bindings
[query VVL kb q]	Résultat en liste VVL	VVL_List de bindings
[prologSolutions kb q]	Toutes les solutions	Liste de dictionnaires
[prologValues kb q var]	Valeurs d'une variable spécifique	Liste de valeurs

Commande	Description	Retour
[prologFirstValue kb q var]	Première valeur uniquement	Valeur unique
[prologExists kb q]	Test d'existence (oui/non)	1 ou 0

Le choix du bon mode est important pour la lisibilité du code. Pour un test conditionnel, `prologExists` est idiomatique. Pour itérer sur les valeurs d'une variable particulière, `prologValues` évite de manipuler des dictionnaires de bindings. Pour les requêtes complexes à plusieurs variables, `prologSolutions` retourne la structure complète. Le mode `JSON` est utile pour la sérialisation et le passage à des frontends ; le mode `VVL` reste manipulable avec les builtins de liste natifs.

```
# Test booleen : Jean est-il parent de Pierre ?
[if [prologExists "famille" "parent(jean, pierre)"]
    [log "Oui"]
    [log "Non"]
]
# => Oui

# Premiere valeur : qui est le premier enfant de Jean ?
$premier = [prologFirstValue "famille" "parent(jean, X)" X]
[log "Premier enfant : " $premier]
# => Premier enfant : pierre

# Toutes les valeurs : tous les enfants de Jean
$enfants = [prologValues "famille" "parent(jean, X)" X]
[for $e $enfants [log "  - " $e]]
# => - pierre
# => - marie

# Solutions completes : tous les couples grand-parent / petit-enfant
$gp = [prologSolutions "famille" "grand_parent(jean, X)"]
[for $s $gp [log $s]]
# => {X: luc}
# => {X: lea}

# Mode JSON pour serialisation
$json = [query JSON "famille" "pere(X, Y)"]
[log $json]
# => [{"X":"jean","Y":"pierre"}, {"X":"jean","Y":"marie"}, ...]
```

11.5. Le pont OO ↔ Prolog : obj-assert

Le mécanisme le plus puissant de ce chapitre — peut-être de toute la Partie IV — est la commande `[obj-assert $instance]` qui **projette n'importe quel objet VVL en faits Prolog**. Cette commande prend une `VVL_Instance` (objet ordinaire, instance ontologique, ou concept réifié du chapitre 9) et génère automatiquement les faits Prolog correspondants dans la KB active. Le moteur de raisonnement peut alors interroger ces objets, leur appliquer des règles, déduire de nouvelles informations — exactement comme s'ils avaient été déclarés en Prolog dès le départ.

Pour chaque instance projetée, **trois types de faits** sont générés systématiquement :

Fait Prolog	Format	Exemple
Instance	instance(prolog_id, classe)	instance(i_7f3a, contact)
Propriété	prop(prolog_id, champ, valeur)	prop(i_7f3a, nom, 'Alice')
Relation	rel(prolog_id, champ, target_id)	rel(i_7f3a, client, i_8b2c)

Figure 11.3 — *Les trois types de faits générés par obj-assert*

```
# Definition d'une classe metier
[class Product
    [field name ""]
    [field price 0]
    [method init [n p]
        [do $self.name = $n $self.price = $p]
    ]
]

# Creation et persistance d'instances
$widget = [new Product "Widget" 9.99]
$gadget = [new Product "Gadget" 49.99]
[$widget.save]
[$gadget.save]

# Projection vers Prolog
[obj-assert $widget]
# Genere :
#   instance(i_uuid_w, product).
#   prop(i_uuid_w, name, 'Widget').
#   prop(i_uuid_w, price, 9.99).

[obj-assert $gadget]
# Idem pour Gadget

# Interrogation Prolog des objets projetes
$cheap = [prologSolutions "global"
    "instance(X, product), prop(X, price, P), P < 20"
]
[for $s $cheap [log $s]]
# => {X: i_uuid_w, P: 9.99}

# Combinaison avec des regles ontologiques chargees auparavant
[rules "default"
    "produit_promo(X) :- instance(X, product), prop(X, price, P), P < 10."
]
$promo = [prologValues "global" "produit_promo(X)" X]
[for $p $promo [log "Promo : " $p]]
# => Promo : i_uuid_w
```

L'impact de ce mécanisme est considérable. **Premièrement**, n'importe quel objet VVL — qu'il soit créé manuellement avec `new`, extrait par le LLM avec `ontology:fromText`, chargé depuis SQLite avec `obj-load`, ou réifié à partir d'un concept avec `[@ "..."]` — peut être interrogé par Prolog. **Deuxièmement**, les règles de raisonnement définies dans la KB (transitivité, symétrie, héritage ontologique, règles métier) s'appliquent aux objets ordinaires sans qu'on ait à les redéfinir pour chaque type. **Troisièmement, et c'est le point le plus important pour la programmation neuro-symbolique** : la frontière entre la base de données (SQLite) et le moteur de raisonnement (Prolog) disparaît. Les deux ne sont plus que deux vues du même ensemble d'objets, l'une optimisée pour la recherche par index, l'autre pour la déduction par règles.

La commande complémentaire `[obj-retract $instance]` retire les faits d'une instance de la KB Prolog — utile pour gérer le cycle de vie ou pour simuler des scénarios sans pollution permanente. La commande `[prolog-query "..."]` est un alias bas-niveau de `prologSolutions` qui opère sur la KB partagée par défaut, sans nécessiter d'argument de nom.

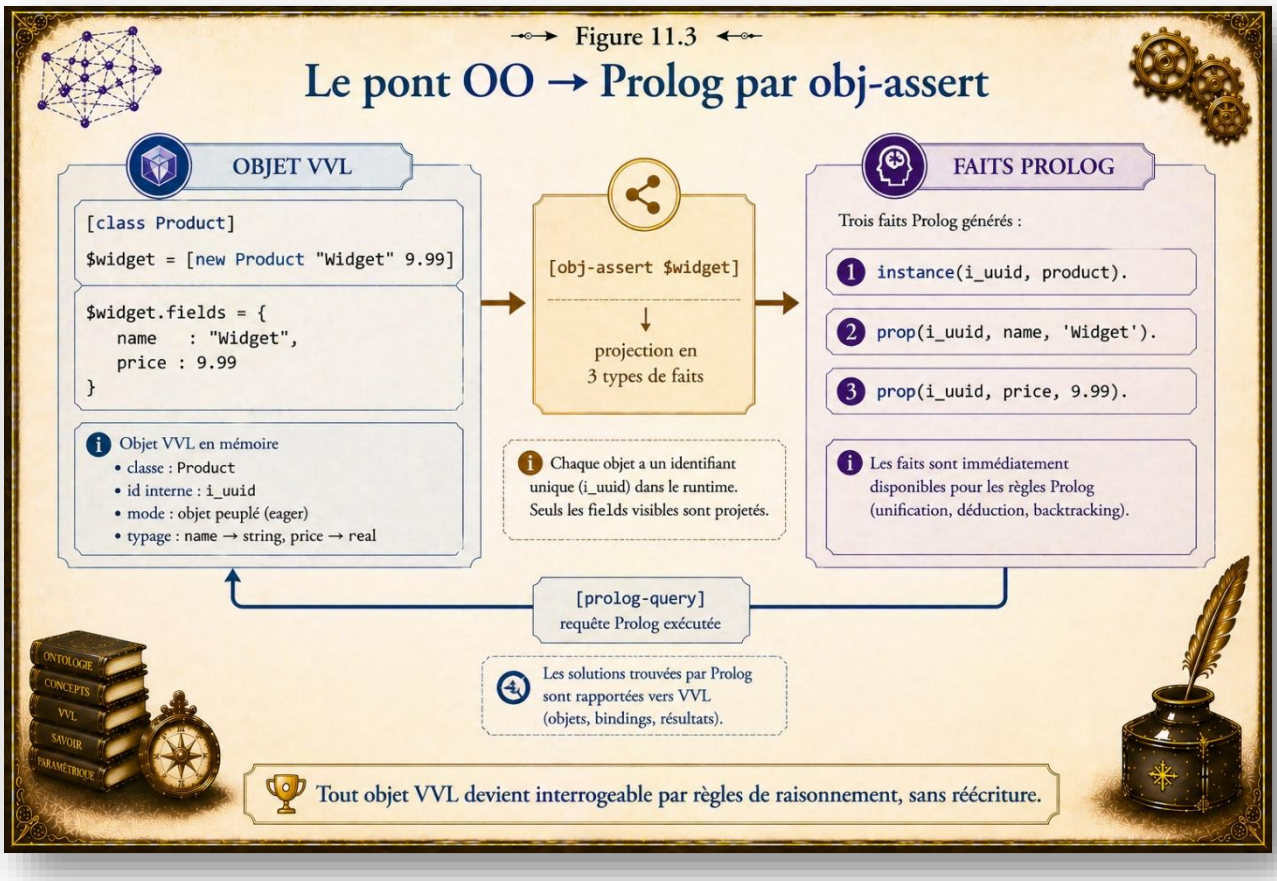


Figure 11.4 — *La projection bidirectionnelle entre objets VVL et faits Prolog*

11.6. Cycle de vie complet d'un objet à travers les couches

L'exemple suivant illustre comment un objet circule à travers **toutes les couches** de VVL : extraction depuis du texte par le LLM, manipulation comme objet VVL standard, persistance SQLite, recherche relationnelle, projection Prolog, raisonnement déductif, modification, et re-sauvegarde. Cette circulation est l'expression la plus concrète de la promesse neuro-symbolique du langage.

```
# 1. Extraction depuis le monde reel par le LLM
$cv = "Bob Durand, 35 ans, bob@tech.fr. Architecte cloud."
$candidat = [ontology:fromText $RH "Candidat" $cv]

# 2. Manipulation comme objet VVL standard
$candidat.score   = 92
$candidat.statut  = "retenu"

# 3. Persistance SQLite
[$candidat.save]
$id = $candidat.id

# 4. Recherche dans la base par index
$retenus = [obj-find Candidat statut "retenu"]
$top     = [obj-find Candidat score 80 ">="]

# 5. Projection vers Prolog pour le raisonnement
[obj-assert $candidat]

# 6. Inferences avec regles metier
[rules "global"
    "embauchable(X) :- instance(X, candidat), prop(X, score, S), S > 90,
                       prop(X, statut, retenu)."
]
$candidats_embauchables = [prologValues "default" "embauchable(X)" X]

# 7. Rechargement dans une autre session
$bob = [obj-load Candidat $id]
[log $bob.nom " - score " $bob.score]

# 8. Modification et re-sauvegarde
$bob.statut = "embauche"
[$bob.save]

# 9. Mise a jour de la KB Prolog
[obj-retract $bob]
[obj-assert $bob]
```

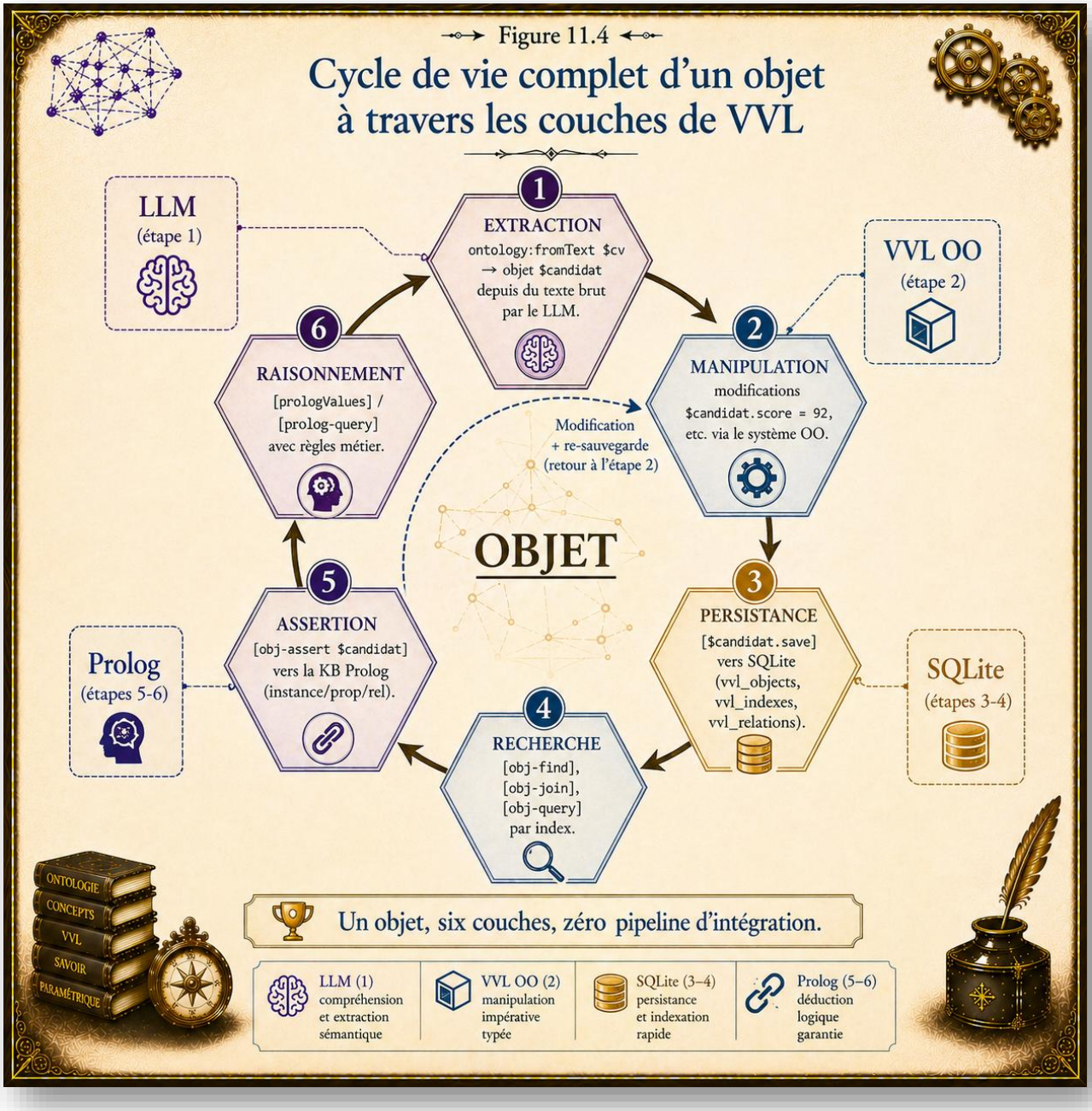


Figure 11.5 — *Le même objet circule à travers les couches : LLM, OO, SQLite, Prolog*

Cette circulation est ce qui distingue VVL des architectures classiques où chaque couche aurait sa propre représentation et nécessiterait des adaptateurs. Ici, **l'objet reste le même** à travers toutes les phases ; ce sont les opérations qu'on lui applique qui changent. La couche LLM le crée à partir de texte ; la couche OO le manipule par méthodes ; la couche SQLite le persiste comme blob JSON ; la couche Prolog le projette en faits ; et chaque transition se fait par un seul builtin natif sans qu'aucun JSON intermédiaire, aucun ORM, aucun pipeline d'intégration ne soit nécessaire.

Ce chapitre a posé les bases du raisonnement symbolique en VVL : faits déclaratifs, règles d'inférence, requêtes, et le pont entre objets et faits. Le **chapitre 12** introduit la couche supérieure — le système

ontologique de VVL —, qui ajoute à Prolog la dimension des **classes**, des **attributs typés**, des **règles métier nommées**, et qui automatise la projection des données extraites par le LLM dans la KB. Là où Prolog raisonne sur des faits *non typés*, l'ontologie raisonne sur des concepts *typés et validés*.

12. Le système ontologique

Mots-clés : ontologie, schéma, ontology:define, ontology:fromText, ontology:rule, ontology:infer, infer-fixpoint, ontology:world, attributs typés, validation, règles métier, mondes éphémères

Le chapitre 11 a posé le moteur Prolog comme couche de raisonnement déductif. Mais Prolog seul, dans sa pureté minimaliste, présente trois limites qui le rendent peu pratique pour le développement d'applications métier : pas de notion de **classe**, pas de **typage** des arguments, pas de **validation** automatique. Un fait `personne(alice, 30, 'a@b.fr')` est strictement identique pour le moteur à `personne(alice, 'invalide', 999)` ; la responsabilité du contrôle revient entièrement au développeur.

Le **système ontologique** de VVL ajoute par-dessus Prolog les couches manquantes : schémas typés (`ontology:define`), extraction structurée par LLM (`ontology:fromText`), règles métier nommées avec priorités (`ontology:rule`), inférence à point fixe (`ontology:infer-fixpoint`), et même un mécanisme de **mondes éphémères** (`ontology:in-world`) qui permet de faire des simulations de scénarios sans toucher à la KB réelle. Ce chapitre couvre l'ensemble du système comme couche intermédiaire entre Prolog (chapitre 11) et les opérateurs Zalta (chapitre 13).

Le chapitre suit la séquence suivante : du fait nu au concept typé (12.1), définition de schéma (12.2), extraction par LLM (12.3), règles métier (12.4), inférence et points fixes (12.5), mondes éphémères pour la simulation (12.6), synthèse de la pyramide symbolique (12.7).

12.1. Du fait nu au concept typé

La couche symbolique de VVL est organisée en **trois étages superposés** dont chacun ajoute un niveau d'abstraction et de garantie. La base, c'est Prolog — le moteur de déduction sur des faits non typés. Au milieu, c'est l'ontologie — Prolog enrichi de schémas, de types et de règles métier nommées. Au sommet, ce sont les opérateurs Zalta du chapitre 13 — l'algèbre conceptuelle qui opère sur les objets ontologiques et sur les concepts réifiés.

Cette stratification correspond à des cas d'usage distincts. **Pour de la pure déduction logique** sur des faits ad-hoc — relations familiales, dépendances de tâches, contraintes de planning —, Prolog seul est suffisant et même préférable pour son économie syntaxique. **Pour modéliser un domaine métier** avec validation, persistance, extraction depuis du texte et règles d'évolution, l'ontologie est l'outil naturel : c'est la couche qui rend le langage utilisable en production. **Pour comparer ou enrichir des concepts** par leur sens — savoir si un objet ontologique satisfait un archétype, intersecter deux concepts, projeter une propriété —, les opérateurs Zalta apportent la dimension neuro-symbolique manquante.

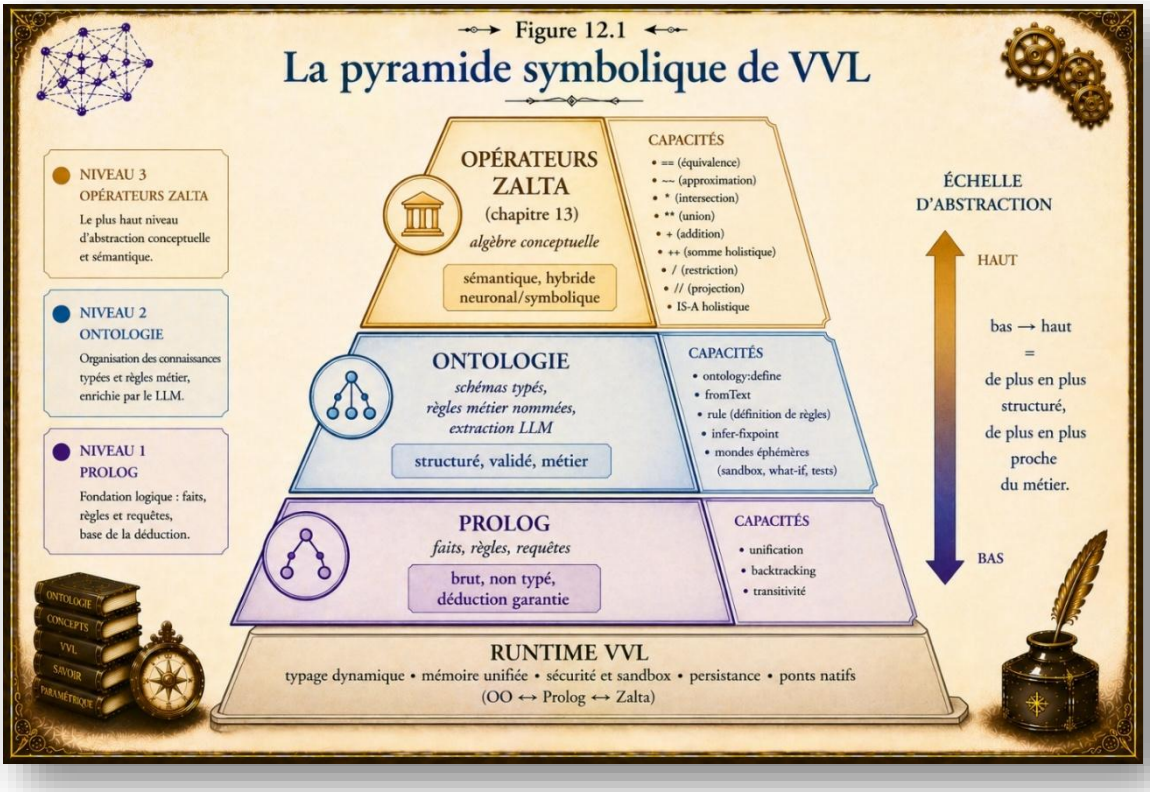


Figure 12.1 — *Les trois étages de la couche symbolique de VVL*

12.2. Définition de schéma : ontology:define

Un schéma ontologique déclare les **concepts** (équivalents des classes) et leurs **attributs typés** d'un domaine métier. La commande `ontology:define` retourne un schéma stocké dans une variable, qui sera ensuite passé aux commandes d'extraction, de définition de règles et d'inférence.

```
# Definition d'un schema "HRSystem" avec une classe Employee
$Schema = [ontology:define "HRSystem"
    [ontology:class Employee
        [attributes
            [name      String  [required true]]
            [age       Integer [min 18] [max 99]]
            [email     String  [type Email]]
            [salary    Float   [min 0]]
            [status    String  [default "Active"]
                                 [enum [Active Inactive Retired]]]
            [seniority String  [default ""]]
        ]
    ]

    [ontology:class Department
        [attributes
            [name   String [required true]]
            [budget Float  [min 0]]
        ]
    ]
]
```

Les types de base disponibles sont `String`, `Integer`, `Float`, `Boolean`, `Date`, et des types métiers spécialisés (`Email`, `URL`, `Currency`). Les annotations sont les mêmes que pour la programmation OO du chapitre 6, mais elles s'appliquent ici à la définition ontologique :

Annotation	Effet	Exemple
[required true]	Le champ doit être renseigné	[name String [required true]]
[default valeur]	Valeur initiale si non spécifiée	[status String [default "Active"]]
[min N]	Valeur numérique minimale	[age Integer [min 18]]
[max N]	Valeur numérique maximale	[age Integer [max 99]]
[type T]	Format spécifique (Email, URL, Currency)	[email String [type Email]]
[enum [v1 v2 ...]]	Valeur restreinte à une liste	[status [enum [A B C]]]

Figure 12.2 — *Annotations de validation des attributs ontologiques*

La validation est appliquée à chaque modification d'instance — pas seulement à la création. Toute valeur qui ne respecte pas les contraintes est silencieusement ignorée pour éviter la corruption de données. Cette politique conservatrice est volontaire : dans un pipeline neuro-symbolique, le LLM peut produire des valeurs aberrantes, et il est préférable que la validation joue le rôle de filtre plutôt que de propager des erreurs en cascade.

12.3. Extraction par LLM : fromText et fromImage

Le pont entre le neuronal (texte non structuré) et le symbolique (instances typées) passe par les commandes `ontology:fromText` et `ontology:fromImage`. Ces commandes prennent un schéma, le nom d'une classe cible, et une source — texte libre ou image — et retournent une instance peuplée. Le LLM se charge de lire la source, d'identifier les valeurs correspondant à chaque attribut du schéma, et de remplir l'instance en respectant les types.

```
# Extraction depuis du texte libre
$cv = "Bob Durand, 35 ans, bob.durand@tech.fr, salaire 75000, departement Engineering"
$bob = [ontology:fromText $Schema "Employee" $cv]

# L'instance est peuplee mais pas encore commitee dans la KB
[log $bob.name]                # => Bob Durand
[log $bob.age]                 # => 35
[log $bob.email]               # => bob.durand@tech.fr
[log $bob.salary]              # => 75000.0

# Extraction depuis une image (organigramme, document scanne, ...)
$alice = [ontology:fromImage $Schema "Employee" "photos/alice-cv.png"]

# Validation automatique : les valeurs invalides sont rejetees
$mauvais_cv = "Carlos Lopez, age 12, carlos@@invalid, salaire -5000"
$carlos = [ontology:fromText $Schema "Employee" $mauvais_cv]
# age=12 rejete (min=18), email rejete (format), salary rejete (min=0)
# => instance partielle avec seulement name="Carlos Lopez"

# Commit dans la KB Prolog (rend les faits interrogeables)
[ontology:commit $bob]
[ontology:commit $alice]
```

La séparation entre **extraction** et **commit** est une décision de conception importante. Une instance extraite par le LLM est *staged* — elle existe en mémoire mais n'est pas encore dans la KB. Cela laisse la possibilité de l'inspecter, de l'enrichir manuellement (par exemple en croisant avec un référentiel CSV via `ontology:set`), ou de la rejeter si certaines valeurs critiques manquent. Le `commit` n'écrit dans la KB que les instances qu'on a explicitement validées.

Cette politique est complémentaire de la validation des types : le développeur garde le contrôle final sur ce qui entre dans la base de connaissances, ce qui est important quand la source est un LLM probabiliste.

12.4. Règles métier : ontology:rule

Les règles métier ontologiques sont l'équivalent typé et nommé des règles Prolog du chapitre 11. Au lieu d'écrire une règle sur des faits bruts, on écrit une règle sur les **instances d'une classe** d'un schéma. La syntaxe `[ontology:rule "nom" $schema [target ...] [priority N] [when ...] [then ...]]` est inspirée du modèle classique des moteurs de règles métier (Drools, ILOG).

```
# Definition d'un schema avec un statut et un seuil d'age
$Schema = [ontology:define "HRSystem"
    [ontology:class Employee
        [attributes
            [name   String]
            [age    Integer]
            [salary Float]
            [status String [default "Active"]]
        ]
    ]
]

# Regle 1 : retraite automatique au-dela de 65 ans
[ontology:rule "RetirementCheck" $Schema
    [target "Employee"]
    [priority 50]
    [when ($this.age > 65)]
    [then [ontology:set $this "status" "Retired"]]
]

# Regle 2 : surveillance des hauts salaires (priorite plus haute = passe en premier)
[ontology:rule "HighSalaryAlert" $Schema
    [target "Employee"]
    [priority 100]
    [when ($this.salary > 100000)]
    [then
        [ontology:set $this "status" "UnderReview"]
        [log "Alerte salaire eleve : " $this.name]
    ]
]

# Application des regles
$updates = [ontology:infer $Schema]
[log $updates]                # => "Inference complete. N updates applied."
```

Trois éléments méritent une attention particulière. **Premièrement, la variable `$this`** est automatiquement injectée dans la portée des blocs `[when]` et `[then]` ; elle pointe vers l'instance courante en cours d'évaluation. Cette convention est la même que `$self` dans le système OO du chapitre 6, mais avec un nom distinct pour bien marquer qu'on est dans le contexte ontologique. **Deuxièmement, la commande `ontology:set`** doit être utilisée à la place d'une assignation directe

`$this.status = "..."` pour modifier un champ : elle met à jour à la fois la KB Prolog *et* le dictionnaire de fields, garantissant que le changement est visible aux deux interfaces. **Troisièmement, la priorité** détermine l'ordre d'exécution : les règles à priorité haute s'exécutent en premier, ce qui est crucial quand plusieurs règles peuvent toucher le même champ.

12.5. Inférence : infer et infer-fixpoint

Une fois les règles définies, on les applique avec `ontology:infer` (un seul passage) ou `ontology:infer-fixpoint` (jusqu'à stabilité). La distinction est essentielle quand les règles sont **en cascade** — c'est-à-dire quand l'effet d'une règle peut activer la condition d'une autre.

infer : un passage

`[ontology:infer $schema]` parcourt toutes les règles enregistrées dans l'ordre de priorité décroissante, et pour chacune, parcourt toutes les instances de la classe cible. Pour chaque instance, si la condition `[when ...]` est vraie, le bloc `[then ...]` est exécuté. Le moteur retourne un message d'état avec le nombre d'updates appliquées. Si une règle Y dépend d'un champ modifié par une règle X, et que X a une priorité inférieure à Y, l'effet de X ne sera pas vu par Y dans ce passage — il faudra attendre un nouvel `infer`.

infer-fixpoint : boucle jusqu'à stabilité

`[ontology:infer-fixpoint $schema max_iterations?]` répète l'inférence en boucle tant que des modifications continuent d'être produites. Le moteur s'arrête au **point fixe** — l'itération où aucune règle n'a plus rien à mettre à jour. Une protection optionnelle limite le nombre maximum d'itérations (10 par défaut) pour éviter les boucles infinies en cas de règles mal conçues. C'est l'outil idoine pour les systèmes de règles complexes où les changements en cascade sont attendus.

```
# Regle en cascade : l'experience determine la seniorite, qui determine le bonus
[ontology:rule "DetermineExperience" $Schema
    [target "Employee"]
    [priority 200]
    [when ($this.years_in_company > 7)]
    [then [ontology:set $this "experience" "high"]]
]

[ontology:rule "DetermineSeniority" $Schema
    [target "Employee"]
    [priority 100]
    [when ($this.experience == "high")]
    [then [ontology:set $this "seniority" "senior"]]
]

[ontology:rule "DetermineBonus" $Schema
    [target "Employee"]
    [priority 50]
    [when ($this.seniority == "senior")]
    [then [ontology:set $this "bonus_eligible" 1]]
]

# Avec infer (un passage) :
$r1 = [ontology:infer $Schema]
# => "Inference complete. 3 updates applied."
# MAIS : seule la premiere regle s'applique sur des instances frais.
# Les regles 2 et 3 ne voient pas les effets de la regle 1 !

# Avec infer-fixpoint (cascade) :
$r2 = [ontology:infer-fixpoint $Schema 5]
```

```
# => "Fixpoint inference: 9 total updates."
# Iteration 1 : regle 1 -> experience=high (3 updates)
# Iteration 2 : regle 2 voit experience=high -> seniority=senior (3 updates)
# Iteration 3 : regle 3 voit seniority=senior -> bonus_eligible=1 (3 updates)
# Iteration 4 : aucune modification -> arret au point fixe
```

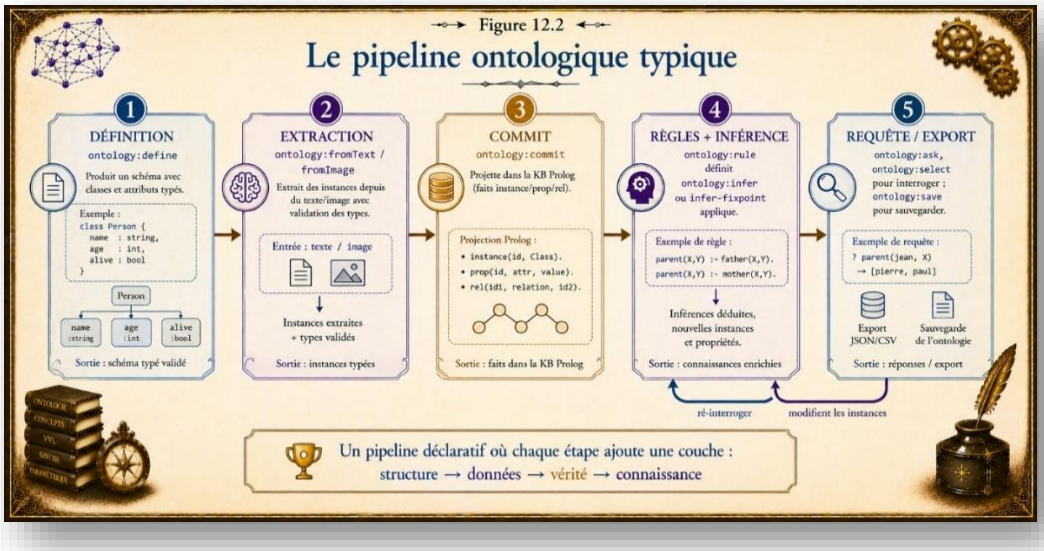


Figure 12.3 — *Les cinq étapes typiques d'un pipeline ontologique en VVL*

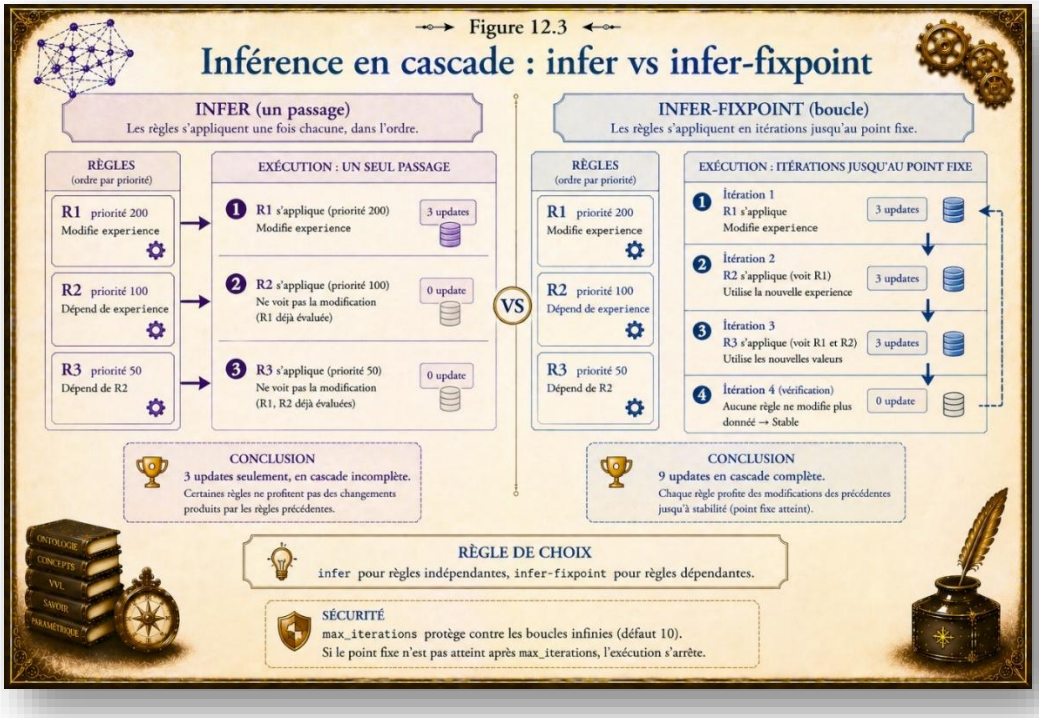


Figure 12.4 — *La différence cruciale entre un passage et le point fixe*

12.6. Mondes éphémères : ontology:in-world

La fonctionnalité la plus distinctive du système ontologique de VVL est sans doute le mécanisme de **mondes éphémères**. La commande `[ontology:in-world "NomDuMonde" body]` crée un *fork complet* de la KB courante, exécute le `body` dans ce monde isolé, puis **restaure automatiquement** la KB réelle à la sortie. Toutes les modifications faites pendant le fork — assertions, retraits, inférences, ajouts de règles — sont confinées au monde éphémère et n'affectent pas la base réelle.

Ce mécanisme ouvre la voie à trois usages que la programmation classique sait mal traiter : la **simulation de scénarios métier** (« que se passerait-il si l'on appliquait cette politique tarifaire ? »), le **diagnostic différentiel** (« quel facteur change si l'on retire cette hypothèse ? »), et l'**analyse de sensibilité** (« comment varie le résultat quand ce paramètre bouge ? »). Dans chacun de ces cas, on veut explorer plusieurs alternatives sans qu'aucune ne contamine la base de référence.

```
# Trois scenarios de prix pour un produit
$produit = [@ "Widget industriel, prix 80E, marge 15%"]

[for $prix [list 60 80 120]
    # Fork complet vers un monde nomme
    [ontology:world (concat "Scenario_" $prix)]
    [ontology:in-world (concat "Scenario_" $prix)
        [ontology:set $produit "prix" $prix]
        [ontology:infer-fixpoint $Schema 3]

        # Mesurer dans ce monde uniquement
        $premium = ($produit ~~ "ProduitPremium")
        $budget  = ($produit ~~ "ProduitBudget")
        [log (concat $prix "E -> Premium:" $premium " Budget:" $budget)]
    ]
    # Sortie du in-world : la KB est automatiquement restauree.
    # $produit.prix est revenu a sa valeur d'origine 80.
]

# Sortie console :
# 60E -> Premium:0.1 Budget:0.9
# 80E -> Premium:0.5 Budget:0.3
# 120E -> Premium:0.9 Budget:0.05
```

La commande `ontology:world` (sans `in-`) crée le monde nommé sans y entrer ; on l'utilise pour préparer plusieurs mondes et y exécuter des scripts indépendants ensuite. La commande `ontology:in-world` entre effectivement dans le monde, exécute le bloc, puis restaure la KB. La paire est conçue pour être combinée avec le pattern `for` ou `pmap` pour explorer un espace de scénarios en parallèle.

L'implémentation interne du fork est un *deep copy* de la KB Prolog au moment de l'entrée, et une *restauration totale* à la sortie. Cela a un coût mémoire proportionnel à la taille de la KB ; pour les très grandes bases, on préférera limiter la portée du `in-world` aux modifications strictement nécessaires.

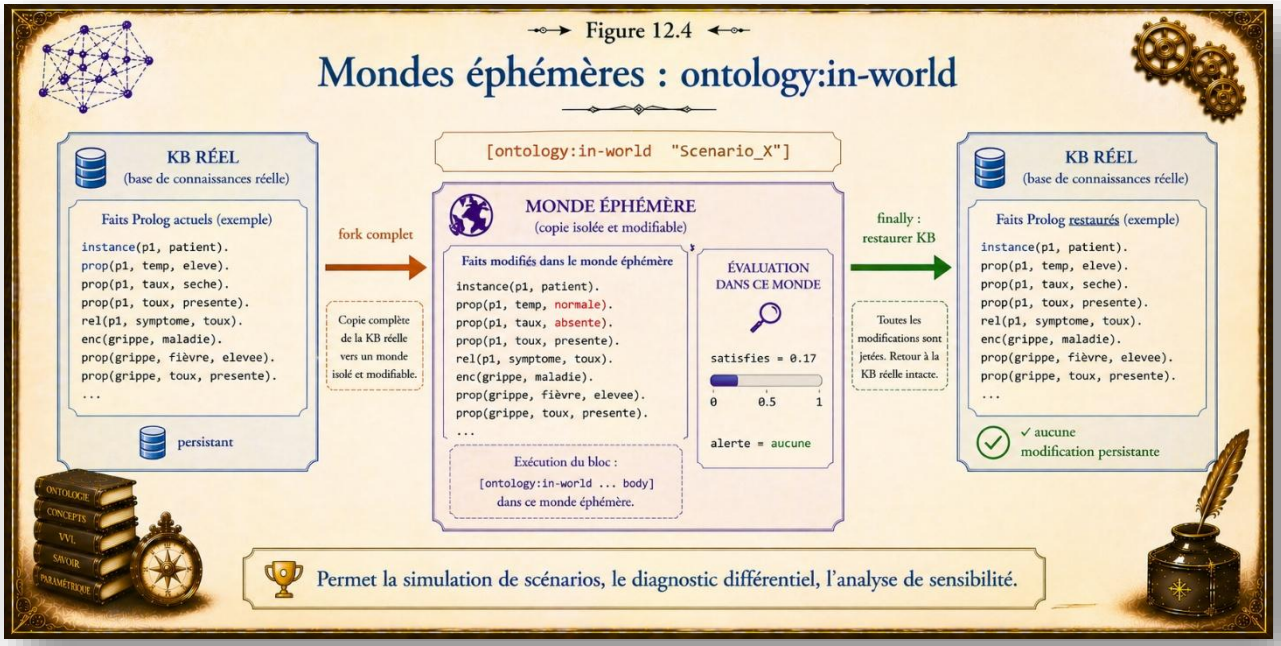


Figure 12.5 — *Le mécanisme de fork et restauration : exploration sans contamination*

12.7. Synthèse : la pyramide symbolique

Le système ontologique de VVL est l'**étage médian** de la pyramide symbolique. Il prend en entrée la robustesse déductive de Prolog (chapitre 11) et y ajoute les structures (classes, attributs typés), la validation, l'extraction depuis du texte par LLM, les règles métier nommées avec priorités, l'inférence à point fixe, et le mécanisme de mondes éphémères. C'est cette combinaison qui rend la couche symbolique utilisable pour des applications métier réelles, pas seulement pour des problèmes académiques de logique.

Couche	Apport propre	Limite
Prolog (ch. 11)	Faits, règles, requêtes, déduction garantie	Pas de typage, pas de validation, pas de classes
Ontologie (ch. 12)	Schémas typés, validation, règles métier nommées, mondes éphémères	Pas d'algèbre conceptuelle hybride
Zalta (ch. 13)	Algèbre sur concepts (==, *, +, /), versions hybrides LLM	Suppose les couches précédentes en place

Figure 12.6 — *La pyramide symbolique : chaque couche complète la précédente*

Le **chapitre 13** clôt la Partie IV en présentant les opérateurs Zalta — l'algèbre de huit opérateurs (==, ~~, *, **, +, ++, /, //) qui couronne la pyramide. Ces opérateurs sont implémentés comme un mixin injecté dans les classes ontologiques et dans les concepts réifiés, et c'est leur dispatch via le MRO du chapitre 6 qui leur donne la même fluidité que les opérateurs natifs. Avec ce dernier chapitre, la couche symbolique de VVL sera entièrement exposée, et nous pourrons aborder la Partie V — le pont neuronal — qui montrera comment cette couche symbolique est alimentée et enrichie par les capacités du LLM.

13. L'algèbre des opérateurs Zalta

Mots-clés : opérateurs Zalta, algèbre conceptuelle, satisfaction, intersection, enrichissement, projection, mixin ZaltaOperators, MRO, matching à 4 niveaux, IS-A holistique, auto-réification contextuelle, archétype

Le chapitre 13 clôt la Partie IV — le pont symbolique de VVL — par le sommet de la pyramide ontologique. Les opérateurs Zalta sont l'aboutissement de tout ce qui a été présenté jusqu'ici dans cette partie : ils opèrent sur les **objets ontologiques** typés du chapitre 12 et sur les **concepts réifiés** du chapitre 9, avec deux modes parallèles — symbolique (déterministe, zéro appel LLM) et hybride (enrichi par le LLM) — qui permettent d'arbitrer finement entre fiabilité, coût et richesse sémantique.

Ces huit opérateurs (`==`, `~~`, `*`, `**`, `+`, `++`, `/`, `//`) ne sont pas des fonctions globales : ils sont implémentés comme un **mixin VVL** nommé `ZaltaOperators`, automatiquement injecté dans toute classe ontologique et dans tout concept réifié. Le dispatch utilise le MRO standard du chapitre 6 — méthode propre > mixin > parent —, ce qui signifie que l'utilisateur peut surcharger n'importe lequel de ces opérateurs sur sa propre classe sans toucher au système. Cette élégance d'intégration est ce qui transforme une algèbre conceptuelle en un véritable composant du langage.

Le chapitre suit la séquence suivante : la motivation et la matrice (13.1), l'architecture par mixin et dispatch MRO (13.2), les quatre paires d'opérateurs avec leurs deux modes — satisfaction (13.3), intersection (13.4), enrichissement (13.5), projection (13.6) —, et enfin la synthèse de la Partie IV (13.7) qui prépare le passage à la Partie V.

13.1. Une algèbre pour les concepts

L'idée d'une algèbre sur les concepts est ancienne — Leibniz parlait d'une *characteristica universalis* qui aurait permis de calculer sur les idées comme on calcule sur les nombres. Edward Zalta, à Stanford, a donné à cette idée une formulation rigoureuse contemporaine dans son *Principia Logico-Metaphysica* (2025), avec quatre opérations fondamentales : la **satisfaction** (un objet satisfait-il les propriétés d'un archétype ?), l'**intersection** (quelles propriétés deux objets partagent-ils ?), l'**enrichissement** (la fusion de deux objets sous priorité), et la **projection** (l'extraction d'une propriété nommée). Ces quatre opérations forment un système clos — leur composition permet d'exprimer toute manipulation conceptuelle utile.

VVL implémente ce système avec une innovation propre : **chaque opération existe en deux versions**, l'une symbolique (déterministe, immédiate, zéro appel LLM) et l'autre hybride (enrichie par le LLM pour les cas où la comparaison stricte échoue). On obtient ainsi huit opérateurs organisés en une matrice 4×2 dont le tableau ci-dessous donne la vue d'ensemble :

Opération	Symbolique	Hybride (LLM)	Retour	Cas d'usage type
Satisfaction	==	~~	Score 0.0–1.0	Vérifier qu'une instance correspond à un archétype
Intersection	*	**	VVL_AbstractObject	Trouver les propriétés communes à deux objets
Enrichissement	+	++	Instance enrichie	Fusionner deux objets (gauche prioritaire)

Opération	Symbolique	Hybride (LLM)	Retour	Cas d'usage type
Projection	/	//	Valeur de la propriété	Extraire une propriété par son nom

Figure 13.1 — *Les huit opérateurs Zalta : quatre paires symbolique/hybride*

La distinction symbolique/hybride n'est pas un détail d'implémentation. **Le mode symbolique** opère par comparaison stricte des valeurs (égalité de chaînes insensible à la casse, seuils numériques `>=`) ; il est instantané, déterministe, et n'engendre aucun coût API. **Le mode hybride** invoque le LLM pour les cas où le matching strict échoue ; il coûte un appel par opération (parfois batché pour économiser), mais résout les cas d'ambiguïté sémantique — synonymes, traductions, formulations différentes — que le mode symbolique laisse passer pour des dissemblances. La règle pratique est de **préférer le mode symbolique par défaut**, et de basculer vers le mode hybride uniquement quand la qualité du matching le justifie.



Figure 13.2 — *La matrice complète des opérateurs Zalta*

13.2. Architecture : dispatch via mixin et MRO

L'élégance d'implémentation des opérateurs Zalta tient à un seul fait : ils ne sont pas du code spécial inscrit dans le moteur d'expression, mais des **méthodes d'un mixin standard** qui s'inscrit dans le mécanisme de dispatch OO du chapitre 6. Le mixin s'appelle `ZaltaOperators`, et il est automatiquement injecté dans deux types de classes : les `VVL_ClassDef` ontologiques (créées par `ontology:define`) et les `VVL_ConceptEntity` (créées par `[@ "..."]`).

Concrètement, lorsqu'une expression `($alice ~~ "ProfilSenior")` est évaluée, le moteur d'expression `VVL_BinOp.graph_eval` détecte que l'opérande gauche est une `VVL_Instance`, et au lieu d'effectuer la comparaison numérique standard, il déclenche la résolution de la méthode `op-eq` (ou son équivalent pour le symbole utilisé) sur la classe de cette instance. La résolution suit alors le **MRO standard** présenté au chapitre 6 :

```
# Dispatch d'un operateur Zalta

# 1. L'expression ($alice == "ProfilSenior") declenche
#    VVL_Equals.graph_eval qui detecte $alice : VVL_Instance.

# 2. Le moteur appelle class_def.resolve_method("op-eq", ctx).

# 3. La resolution suit le MRO :
#    a) self.methods["op-eq"]    -> si l'utilisateur a surcharge, on utilise
#    b) mixin ZaltaOperators     -> sinon, le mixin Zalta
#    c) parent.methods           -> sinon, l'heritage classique

# 4. Si Zalta est selectionne, il execute son code de matching
#    (symbolique pour ==, hybride pour ~~).

# Surcharge utilisateur : prioritaire sur le mixin
[class MonMateriau
    [field densite 0]
    # Ma logique custom pour == (ignore Zalta)
    [method "==" [other]
        ($self.densite == $other.densite)
    ]
]
# Le mixin Zalta n'est PAS appele pour MonMateriau
# car la methode propre prend la priorite.
```

Trois conséquences importantes découlent de cette architecture. **Premièrement, l'extensibilité** : un développeur peut surcharger n'importe quel opérateur Zalta sur sa propre classe en définissant la méthode correspondante (`"=="`, `"~~"`, `"*"`, etc.), sans avoir à modifier le système. **Deuxièmement, la cohérence** : les opérateurs Zalta sont dispatchés exactement de la même manière que les opérateurs surchargés de la section 6.6 ; il n'y a pas deux mécanismes en parallèle, mais un seul système OO uniforme. **Troisièmement, la composition** : les opérateurs Zalta peuvent être appelés sur n'importe quel objet qui hérite leur mixin (instance ontologique, concept réifié, ou même un objet ordinaire si on lui ajoute le mixin manuellement).

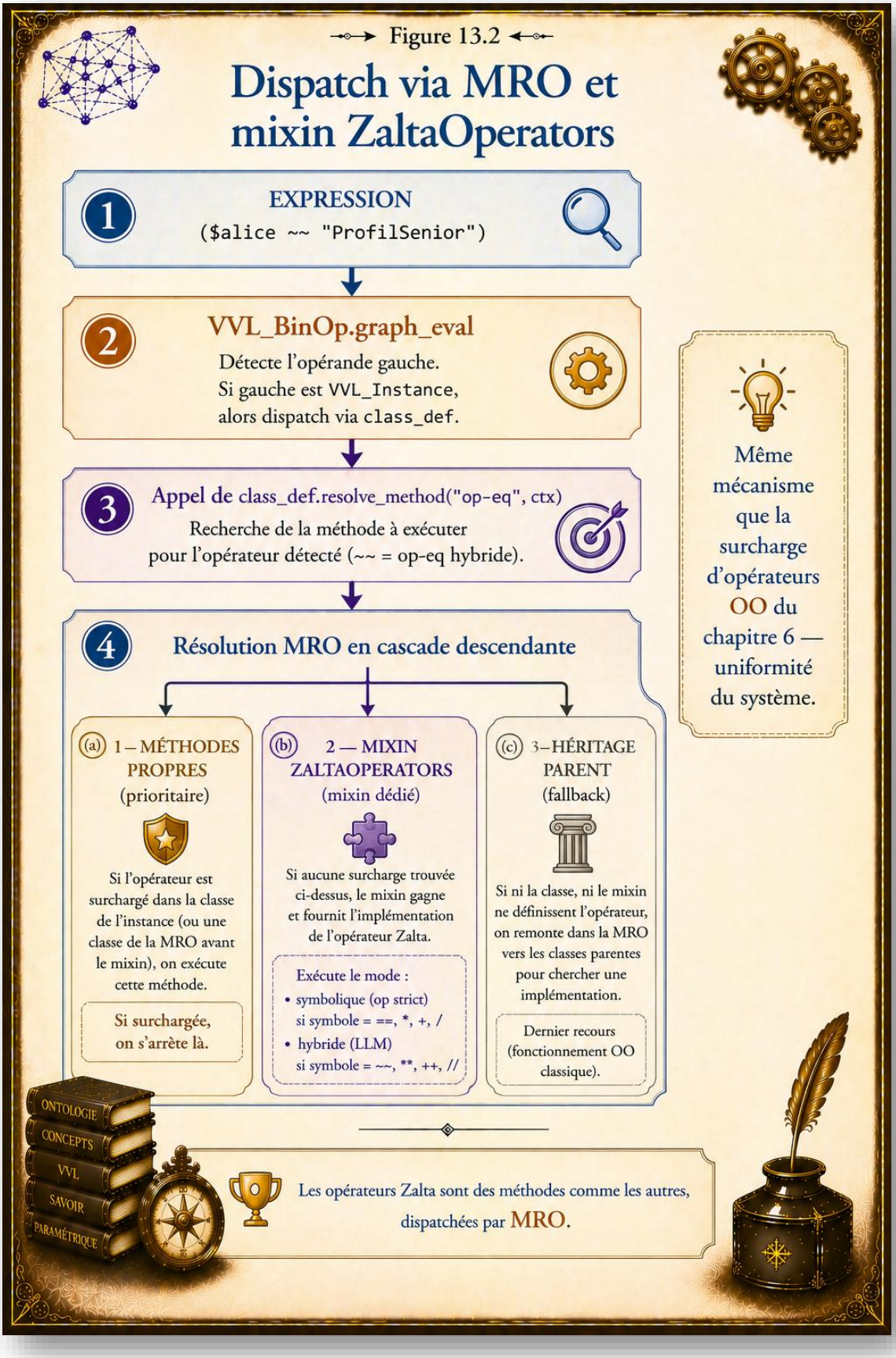


Figure 13.3 — *Le dispatch des opérateurs Zalta s'inscrit dans le système OO standard*

13.3. Satisfaction (== et ~~)

L'opérateur == est la **transposition directe de l'opération `satisfies`** de la théorie de Zalta. Il compare les propriétés *exemplifiées* d'une instance (ses valeurs réelles) avec les propriétés *encodées* d'un archétype (ses valeurs canoniques) ; il retourne un score entre 0.0 et 1.0 qui est le ratio des propriétés satisfaites sur le nombre total d'encodages. Le matching symbolique utilise l'égalité de chaînes insensible à la casse, et le seuil numérique >= (l'instance peut dépasser la valeur canonique sans cesser de la satisfaire).

```
# Definir un schema avec un archetype
$RH = [ontology:define "RH"
    [ontology:class Candidat
        [attributes [competence String] [experience Integer] [salaire Integer]]
    ]

    [ontology:abstract ProfilSenior : Candidat
        [encode competence "dev"]
        [encode experience 5]
    ]
]

# Creer une instance
$alice = [ontology:fromText $RH "Candidat" "Alice, dev senior, 9 ans, salaire 65k"]

# Satisfies symbolique : compare strict
[log ($alice == "ProfilSenior")]
# competence: "dev" == "dev"  -> match
# experience: 9 >= 5          -> match
# => 1.0  (2/2 proprietes encodees satisfaites)

# Comparaison entre deux instances
[log ($alice == $bob)]            # => score selon les fields communs
[log ($alice == $alice)]          # => 1.0 (identite)
```

L'opérateur hybride ~~ étend == par un **matching sémantique à quatre niveaux**. Les niveaux 1 à 3 sont déterministes ; le niveau 4 invoque le LLM pour les paires que les niveaux précédents n'ont pas pu résoudre. Cette stratification est conçue pour minimiser le coût LLM : si une paire est résolue par un niveau symbolique, elle ne déclenche aucun appel.

Niveau	Type de matching	Exemple	Score
1	Symbolique strict	"dev" == "dev"	1.0 (déterministe)
2	Numérique avec seuil	9 >= 5	1.0 (déterministe)
3	Substring	"dev" in "developpeuse senior"	0.9 (déterministe)
4	Sémantique LLM (batché)	"developpeuse senior" ~ "dev"	0.85 (1 appel)

Figure 13.4 — *Les quatre niveaux du matching de l'opérateur ~~*

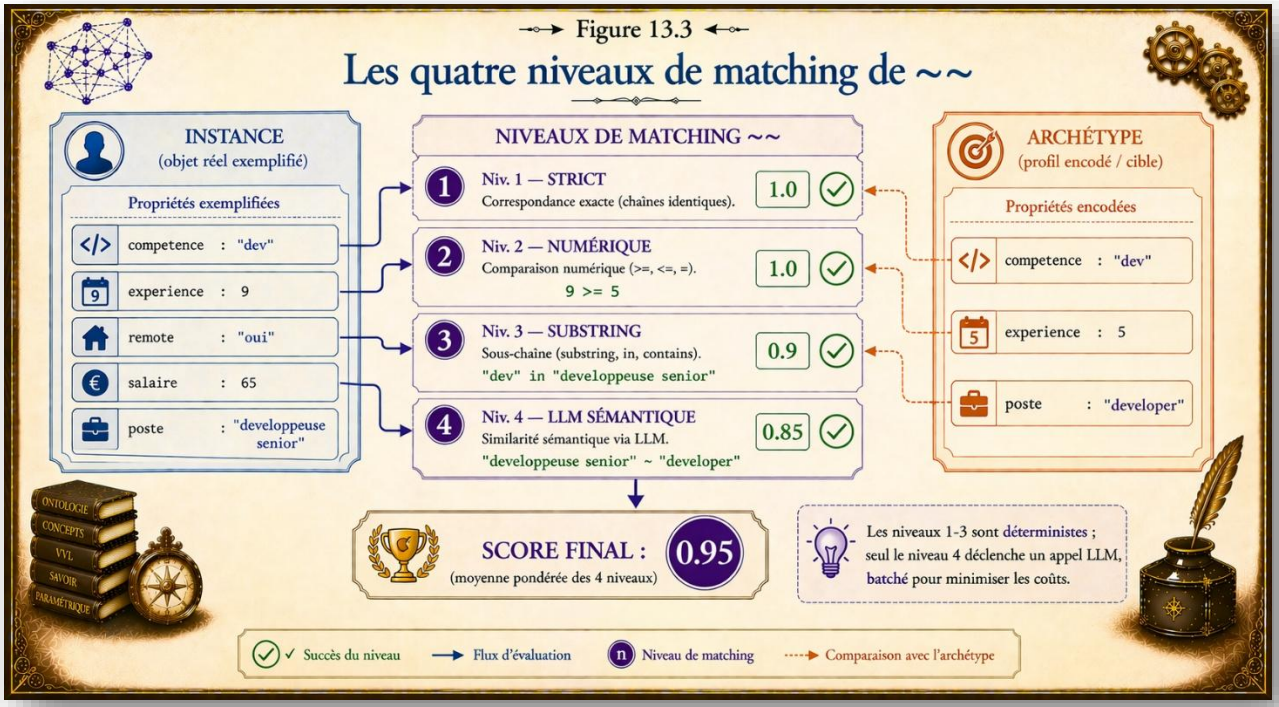


Figure 13.5 — *La descente progressive du matching : strict → sémantique*

Une **subtilité cruciale** apparaît quand ~~ est appelé entre **deux concepts réifiés** (deux objets `[@ "..."]`). Au lieu de comparer propriété par propriété — ce qui produirait souvent des faux négatifs car les concepts génériques (« Étoile ») ont peu de propriétés concrètes —, le moteur bascule en **mode IS-A holistique** : il pose au LLM la question directe « À quel point X est-il un Y ? » en un seul appel. Ce mode évite que `Soleil ~~ Étoile` retourne un score faible simplement parce que « Étoile » n'a pas de propriétés numériques exactes.

```
# IS-A holistique : un seul appel LLM, question directe
$soleil = [@ "Soleil"]

($soleil ~~ "Etoile")          # => ~0.95  (Soleil EST une etoile)
($soleil ~~ "Planete")         # => ~0.05  (Soleil n'est pas une planete)
($soleil ~~ "Naine jaune")     # => ~0.90  (classification astronomique correcte)
($soleil ~~ "Trou noir")       # => ~0.02  (categorie astronomique mais distincte)
```

13.4. Intersection (* et **)

L'opérateur * retourne un nouvel objet abstrait — de type `VVL_AbstractObject`, exactement au sens de Zalta — qui contient **uniquement les propriétés communes** aux deux opérandes. En mode symbolique, deux propriétés sont considérées identiques si elles ont le même nom et la même valeur (selon les règles de matching strict). Le résultat est immédiatement utilisable comme un objet, peut être inspecté, projeté, ou comparé à un autre archétype.

```
$alice = [ontology:fromText $RH "Candidat" "Alice, dev senior, 9 ans"]
$bob   = [ontology:fromText $RH "Candidat" "Bob, dev confirme, 5 ans"]

# Intersection symbolique : proprietes strictement identiques
$commun = ($alice * $bob)
[log $commun.properties]
# => [competence]  (les deux sont "dev")
# experience differe : 9 vs 5 => non commune

# Intersection avec un archetype
$match = ($alice * "ProfilSenior")
# => <Abstract:Intersect_ProfilSenior [competence: "dev", experience: 5]>

# Intersection hybride **
# Le LLM detecte les correspondances semantiques
$alice_fr = [@ "developpeuse senior"]
$alice_en = [@ "senior developer"]
$commun_h = ($alice_fr ** $alice_en)
# => <Abstract [role: "developpeuse"/"developer", level: "senior"/"senior"]>
# Le ** mappe les noms de proprietes potentiellement différents
```

L'opérateur hybride ** enrichit l'intersection de deux mécanismes propres au LLM. **Premièrement, le matching sémantique des valeurs** : deux propriétés peuvent être considérées proches même avec des chaînes différentes (`"developpeuse"` ≈ `"developer"`). **Deuxièmement, le cross-matching des noms de propriétés** : deux propriétés peuvent être identifiées comme équivalentes même avec des noms différents (`diameter_km` ≈ `diametre`, `birth_year` ≈ `annee_naissance`). Ce dernier point est précieux quand on intersecte des concepts réifiés dont les noms de propriétés ont été générés indépendamment par le LLM dans des contextes différents — situation fréquente dans les pipelines neuro-symboliques.

13.5. Enrichissement (+ et ++)

L'opérateur + correspond à l'**addition de concepts de Leibniz** (A ⊕ B). Il fusionne les propriétés de deux objets, l'opérande gauche étant prioritaire — ses valeurs existantes ne sont jamais écrasées. Si l'opérande droit est un archétype ou un concept réifié, ses propriétés viennent compléter les champs encore vides ou manquants à gauche. C'est l'outil canonique pour enrichir un objet partiellement extrait avec un référentiel.

```
# Alice extraite avec donnees minimales
$alice = [ontology:fromText $RH "Candidat" "Alice, dev"]
[log $alice.competence]              # => dev
[log $alice.experience]              # => 0  (defaut, non extrait)
[log $alice.salaire]                 # => 0  (defaut)

# Enrichir Alice avec un archetype "ProfilSenior" qui code des valeurs canoniques
$alice_enrichie = ($alice + "ProfilSenior")
[log $alice_enrichie.competence]     # => dev   (Alice avait deja, conservee)
[log $alice_enrichie.experience]     # => 5     (defaut de l'archetype)

# Fusion de deux extractions du meme employe depuis sources differentes
$source_a = [ontology:fromText $RH "Candidat" "Alice, 30 ans, Engineering"]
$source_b = [ontology:fromText $RH "Candidat" "Alice Dupont, performance 95"]

$alice_complete = ($source_a + $source_b)
# Resultat fusionne :
#   nom         : "Alice"          (de $source_a, prioritaire)
#   age         : 30               (de $source_a)
#   departement : "Engineering"    (de $source_a)
#   performance : 95               (de $source_b, complete)
```

L'opérateur hybride `++` étend l'enrichissement par le LLM en réalisant un **mapping sémantique des noms de propriétés**. Si l'opérande gauche a un champ `taille` et l'opérande droit un champ `diameter_km`, l'opérateur symbolique `+` les considérera comme distincts ; l'opérateur `++` les unifiera grâce à un appel LLM préalable qui établit la correspondance. C'est utile quand on fusionne des données provenant de schémas hétérogènes — base interne et API externe, par exemple.

13.6. Projection (/ et //)

L'opérateur `/` est la projection la plus simple : il **extrait une propriété par son nom exact**. Si le nom n'existe pas, il retourne une chaîne vide. C'est l'équivalent du `getattr` de Python ou de l'accès dictionnaire `obj[key]`, mais intégré comme opérateur du langage.

```
$soleil = [@ "Soleil"]

# Projection exacte
[log ($soleil / "diameter_km")]      # => 1392700
[log ($soleil / "type")]             # => "G-type main-sequence"
[log ($soleil / "taille")]           # => ""  (propriete absente)
[log ($soleil / "mass_kg")]          # => 1.989e30
```

L'opérateur hybride `//` étend la projection par une **recherche sémantique du nom de propriété le plus proche**. Si l'on demande une propriété qui n'existe pas exactement mais dont une variante sémantiquement équivalente existe, le LLM identifie la correspondance. Ce mécanisme rend les concepts réifiés interrogeables dans n'importe quelle langue ou terminologie, sans qu'on ait à connaître les noms exacts choisis par le LLM lors du bootstrap.

```
$soleil = [@ "Soleil"]

# Projection symbolique : echec si nom different
[log ($soleil / "taille")]            # => ""    (pas de propriete "taille")
[log ($soleil / "temperature")]       # => ""    (le champ s'appelle temperature_k)

# Projection semantique : trouve la propriete la plus proche
[log ($soleil // "taille")]           # => 1392700  (taille -> diameter_km)
[log ($soleil // "temperature")]      # => 5778     (-> temperature_k)
[log ($soleil // "size")]             # => 1392700  (anglais -> diameter_km)
[log ($soleil // "diametre")]         # => 1392700  (orthographe -> diameter_km)
```

Une remarque importante : `//` est plus coûteux que `/` puisqu'il invoque le LLM pour faire le matching de nom. Le résultat est cependant mis en cache pour le couple (objet, nom demandé) ; on ne paye qu'une seule fois pour chaque résolution. Sur des programmes qui interrogent répétitivement les mêmes concepts avec les mêmes alias, le coût asymptotique tend vers zéro, comme pour les autres opérateurs sémantiques de VVL.

13.7. Synthèse : un objet, trois couches symboliques

Avec ce chapitre, la Partie IV est complète. Le tableau ci-dessous résume les apports cumulés des trois couches symboliques de VVL et leurs interactions :

Couche	Chapitre	Apport propre	Exemple canonique
Prolog	11	Faits, règles, déduction garantie	[query ... "ancetre(jean, X)"]
Ontologie	12	Schémas typés, validation, règles métier, mondes	[ontology:rule "Retraite" ...]
Zalta	13	Algèbre conceptuelle (8 opérateurs sym/hybride)	($soleil ~~ "Etoile") => ~0.95

Figure 13.6 — *Récapitulatif de la pyramide symbolique de VVL*

Ces trois couches **ne sont pas trois sous-systèmes séparés** : elles partagent la même base de connaissances, le même système OO, le même runtime. Un objet `$alice` extrait par `ontology:fromText` au chapitre 12 est **simultanément** un fait Prolog interrogeable au chapitre 11 (via `obj-assert`), une instance ontologique gouvernée par des règles métier, et un opérande potentiel des opérateurs Zalta du chapitre 13. Cette unification est ce qui rend le langage *neuro-symbolique* au sens fort — pas une bibliothèque de raisonnement greffée sur un langage généraliste, mais un langage où le raisonnement est intégré au cœur du modèle d'exécution.

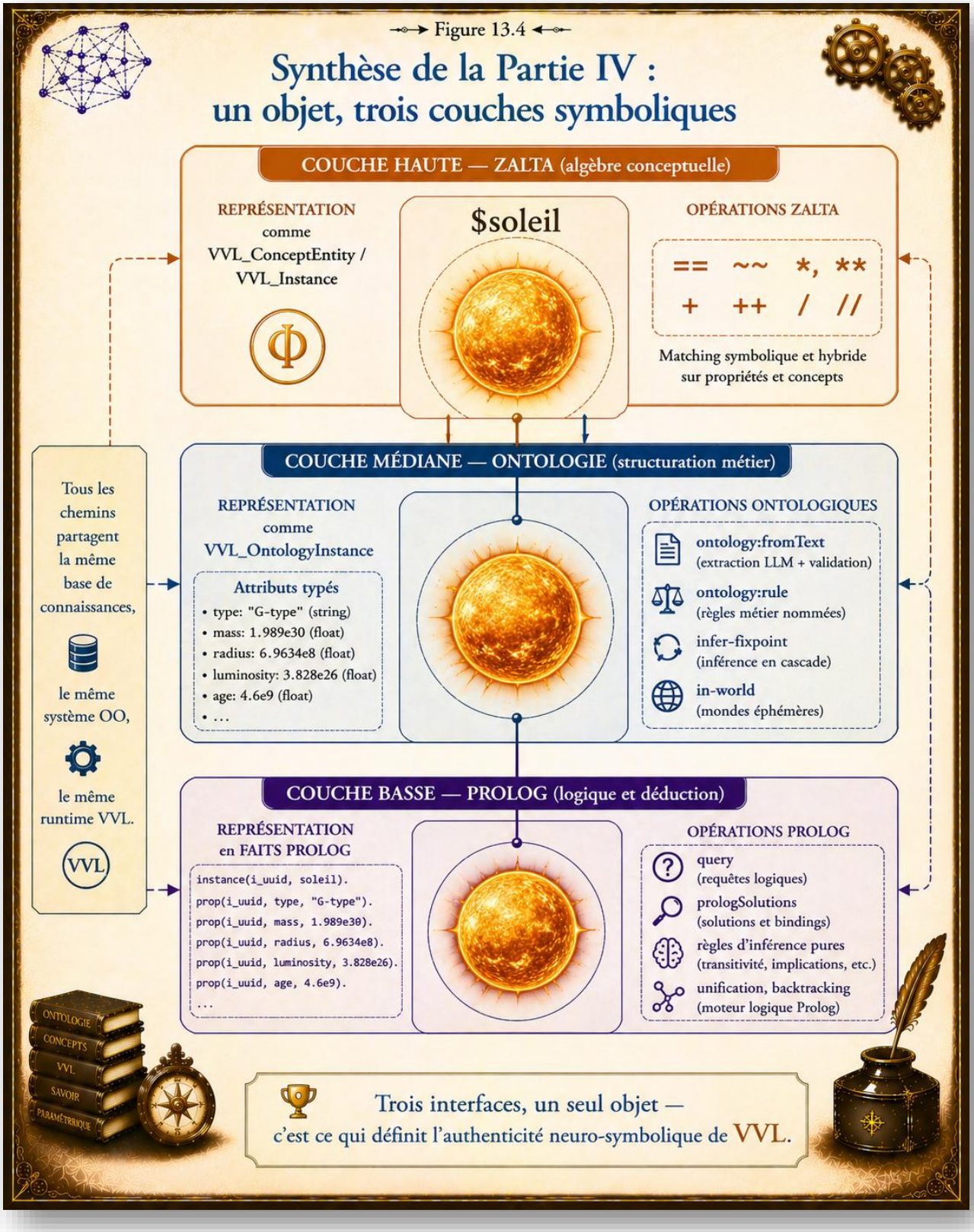


Figure 13.7 — *Un seul objet, trois couches d'interaction symbolique*

La **Partie V** qui s'ouvre au chapitre 14 marque le retour vers le neuronal. Là où la Partie III a montré le LLM comme co-processeur d'expression, et la Partie IV a posé l'infrastructure symbolique qui peut

recevoir et structurer ses sorties, la Partie V montre comment le neuronal **alimente** et **enrichit** cette infrastructure. Quatre chapitres successifs — RAG (chapitre 14), extraction (chapitre 15), Gherkin et ontologie dynamique (chapitre 16), apprentissage inductif de règles (chapitre 17) — exposent les pipelines qui permettent au LLM d'enrichir continuellement la couche symbolique, transformant ainsi le système entier en une boucle d'**oscillation neuro-symbolique** authentique.

Partie V — Le pont neuronal

Chapitres 14 à 17 — RAG, extraction structurée, ontologie dynamique, induction

14. RAG et recherche sémantique

Mots-clés : RAG, retrieval, embeddings, similarité cosinus, ragInsert, ragAsk, ragSearchContext, ragExtractJson, linéarisation, KB vectorielle, Genius, synergie ontologique

La Partie V s'ouvre sur la première grande capacité du **versant neuronal** de VVL : le RAG (Retrieval-Augmented Generation). Là où la Partie III a montré le LLM en quatre rôles d'opérateur ponctuel, et où la Partie IV a posé l'infrastructure symbolique qui peut recevoir et structurer ses sorties, la Partie V montre comment alimenter cette infrastructure avec des **volumes** de connaissances. Le RAG est l'outil qui transforme un corpus documentaire — manuels techniques, catalogues de produits, archives juridiques — en une base interrogeable en langage naturel.

Le RAG résout un problème fondamental des LLM contemporains : leur connaissance est figée à la date d'entraînement, et ils hallucinent volontiers sur les sujets qu'ils ne connaissent pas. Le principe du RAG est d'éviter de demander au LLM de répondre à partir de sa mémoire, et de plutôt **lui fournir le contexte pertinent** extrait d'une base documentaire externe. Le LLM joue alors un rôle réduit mais fiable : synthétiser et reformuler à partir de documents factuels, sans avoir à les inventer.

Le système RAG de VVL se distingue des implémentations classiques (LangChain, LlamaIndex) par son **intégration native** dans le langage : les commandes RAG sont des builtins au même titre que `log` ou `for`, avec une syntaxe uniforme et une composition naturelle avec les autres capacités du runtime — ontologie, Prolog, scraping, concurrence. Le chapitre suit la séquence : limites du LLM seul (14.1), pipeline en trois phases (14.2), insertion de documents (14.3), quatre modes de requête (14.4), CRUD documentaire (14.5), Genius comme application canonique (14.6), synergie avec l'ontologie (14.7).

14.1. Pourquoi RAG : les limites du LLM seul

Un grand modèle de langage est, par construction, un système qui *prédit* du texte plausible à partir de probabilités apprises. Cette prédiction fonctionne admirablement pour les sujets bien représentés dans les données d'entraînement, mais elle dérape sur deux types de questions. **Le premier type** concerne les connaissances *spécialisées* — politique interne d'une entreprise, manuel technique d'un produit récent, jurisprudence d'un tribunal local — que le LLM n'a jamais vues. **Le deuxième type** concerne les connaissances *évolutives* — actualité, prix, états de stock, dossiers en cours — qui changent plus vite que la fréquence d'entraînement du modèle.

Sans précaution, un LLM répond à ces questions en **hallucinant** : il produit une réponse plausible mais fausse, indistinguable d'une réponse correcte par sa forme. Ce comportement est dangereux dans tout contexte professionnel — médical, juridique, financier, technique — où une réponse fausse peut avoir des conséquences réelles. L'industrie a donc convergé vers une stratégie commune : **séparer la recherche de la formulation**. La recherche est faite par un système déterministe sur une base documentaire authentique ; la formulation est faite par le LLM en s'appuyant exclusivement sur ce qu'il vient de recevoir.

C'est exactement ce que fait le RAG. Et c'est exactement ce que VVL intègre, depuis la phase d'indexation initiale jusqu'à l'extraction structurée, en passant par les modes de requête intermédiaires qui permettent d'ajuster le compromis entre coût, latence et précision.

14.2. Le pipeline RAG en trois phases

Le pipeline RAG se décompose en trois phases dont chacune implique un traitement spécifique. **L'indexation** est faite une fois pour toutes (offline) ; **le retrieval** et la **génération** sont faites à chaque requête.

Phase 1 — Indexation (offline)

Les documents textuels sont découpés en **chunks** — fragments de taille contrôlée, typiquement quelques centaines de mots — pour pouvoir être indexés et retrouvés indépendamment. Chaque chunk est ensuite transformé en un **vecteur numérique** (embedding) par un modèle de vectorisation, par exemple `text-embedding-3-small` d'OpenAI. Ces vecteurs, dans un espace de dimension typiquement 1536, capturent le sens sémantique du texte : deux textes proches en sens auront des vecteurs proches dans cet espace, indépendamment des mots exacts utilisés. Les vecteurs sont stockés dans une **base vectorielle** indexée pour une recherche rapide.

Phase 2 — Retrieval (recherche)

La question de l'utilisateur est elle aussi vectorisée par le même modèle d'embedding. Le moteur compare alors ce vecteur à tous ceux de la base, par **similarité cosinus** (cosinus de l'angle entre les deux vecteurs, normalisés). Les K chunks les plus proches sémantiquement sont récupérés. Cette opération est rapide — `O(log n)` avec des index HNSW (Hierarchical Navigable Small World) — et **ne nécessite aucun appel LLM**. Pour une base de cent mille chunks, le retrieval prend quelques millisecondes.

Phase 3 — Generation (génération)

Les chunks récupérés sont insérés dans le prompt du LLM comme contexte, avec l'instruction explicite de répondre uniquement à partir de ces données. Le LLM formule alors une réponse précise et sourcée, sans hallucination — les hallucinations se produisent quand le LLM doit *inventer* du contenu, pas quand il *synthétise* un contenu déjà fourni.

Phase	Action	Coût LLM	Commande VVL
Indexation	Découpage + vectorisation	Embedding (faible)	ragInsert, ragInsertBatch
Retrieval	Similarité cosinus sur les vecteurs	Aucun	ragSearchContext
Génération	LLM formule la réponse	Completion (standard)	ragAsk, ragAskAdvanced
Extraction	LLM structure la réponse en JSON	Completion (standard)	ragExtractJson

Figure 14.1 — *Les phases du pipeline RAG et leurs coûts respectifs*

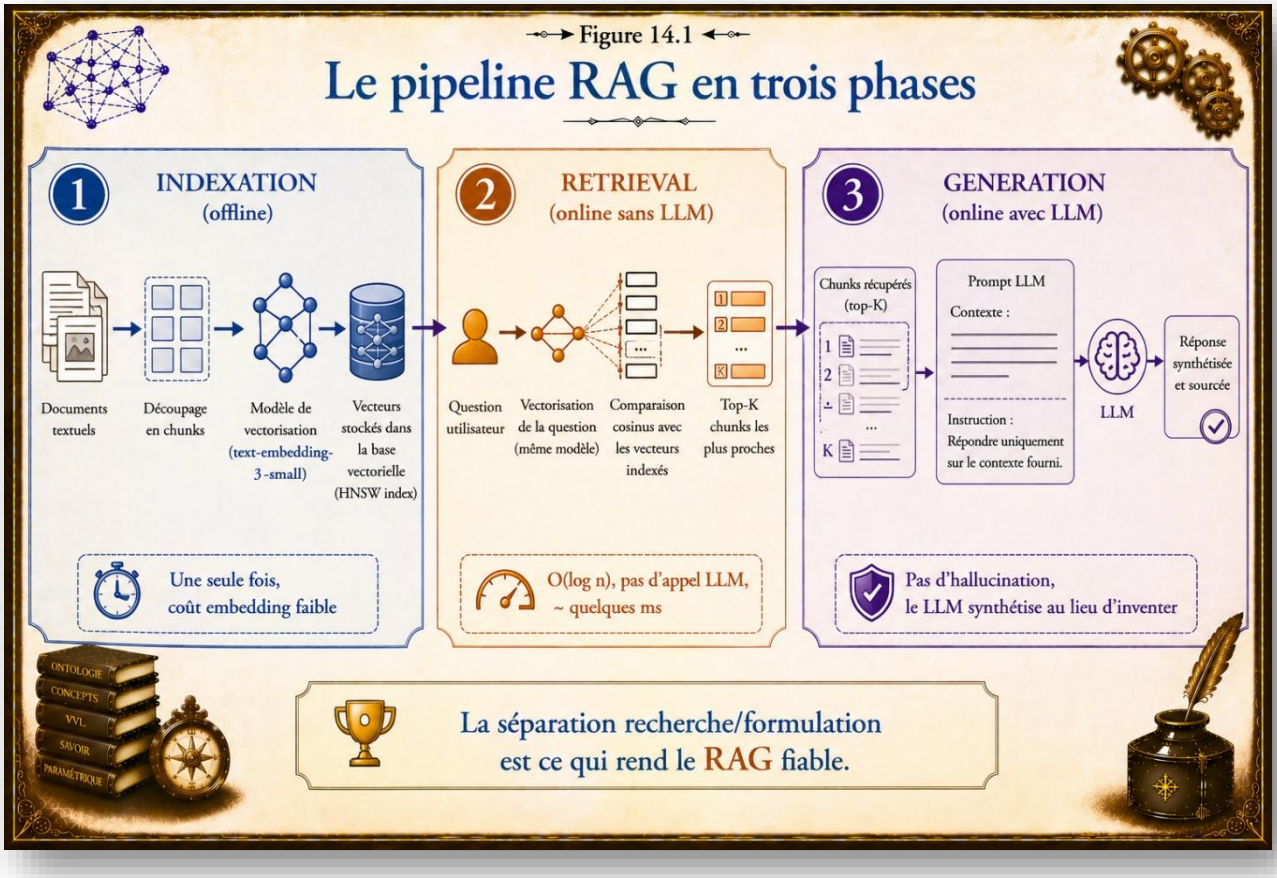


Figure 14.2 — *Le pipeline RAG : indexation, retrieval, génération*

14.3. Insertion de documents : construire la base de connaissances

En VVL, une **base de connaissances** est identifiée par un nom de chemin (par exemple `"myKB"`, `"products/catalog"`, `"docs/legal"`). Chaque KB est un store indépendant résolu dans le workspace courant. Plusieurs KB peuvent coexister simultanément pour des domaines de connaissances différents — on garde séparées la base produits, la base juridique, la base technique. Les documents et leurs vecteurs **survivent à la fermeture de la session** : la KB est persistante, indexée une fois, interrogée toujours.

Trois modes d'insertion sont disponibles. L'insertion unitaire (`ragInsert`) convient pour quelques documents ou pour ajouter des éléments à la volée. L'insertion avec retour d'identifiant (`ragInsertId`) renvoie le `doc-xxxxx` assigné, utile quand on a besoin de mettre à jour ou de supprimer ce document plus tard. L'insertion batch (`ragInsertBatch`) est recommandée pour les volumes importants car elle optimise les appels au modèle de vectorisation, traitant plusieurs textes par appel API.

```
# Insertion unitaire simple
[ragInsert "myKB" "Paris est la capitale de la France, 2.1 millions d'habitants."]
[ragInsert "myKB" "Berlin est la capitale de l'Allemagne, 3.6 millions d'habitants."]

# Insertion avec retour d'identifiant
$id = [ragInsertId "myKB" "Rome est la capitale de l'Italie."]
[log "Document insere avec l'ID : " $id]   # => doc-00003

# Insertion batch (optimisee pour le volume)
[ragInsertBatch "myKB" `[
    "Madrid est la capitale de l'Espagne."
    "Lisbonne est la capitale du Portugal."
    "Vienne est la capitale de l'Autriche."
]]
```

Linéarisation des données structurées

Pour les données structurées (JSON, CSV, lignes de base de données), un travail de **linéarisation** est nécessaire avant insertion. Le RAG opère sur du texte ; il faut donc convertir chaque enregistrement en une chaîne descriptive avant de l'indexer. La qualité de cette linéarisation détermine directement la qualité du retrieval — un texte bien formaté avec des champs nommés produira de meilleurs résultats qu'un JSON brut.

```
# Catalogue produits sous forme de JSON structure
$catalogue = <[
    {"ref": "CHAIR-001", "nom": "Chaise ErgoMax",
     "desc": "Chaise ergonomique support lombaire, mesh respirant",
     "prix": 299.99},
    {"ref": "DESK-001",  "nom": "Bureau Assis-Debout",
     "desc": "Bureau reglable 70-120cm, plateau chene 160x80",
     "prix": 799.99},
    {"ref": "LAMP-001",  "nom": "Lampe LED Architecte",
     "desc": "Bras articule, 3 temperatures, dimmer tactile",
     "prix": 45.99},
    {"ref": "MON-001",   "nom": "Ecran 27 4K IPS",
     "desc": "UHD 4K, 99% sRGB, HDR10, USB-C 65W",
     "prix": 449.99}
]>

# Linearisation : chaque produit devient un texte descriptif riche
$textes = [map [lambda [p]
    [join ["REF: " $__p.ref " | NOM: " $__p.nom
           " | DESC: " $__p.desc " | PRIX: " $__p.prix " EUR"]]
] $catalogue]

# Indexation batch
[ragClear "products"]                          # vide la base
[ragInsertBatch "products" $textes]
[log "Catalogue indexe : " [length $textes] " produits"]
```

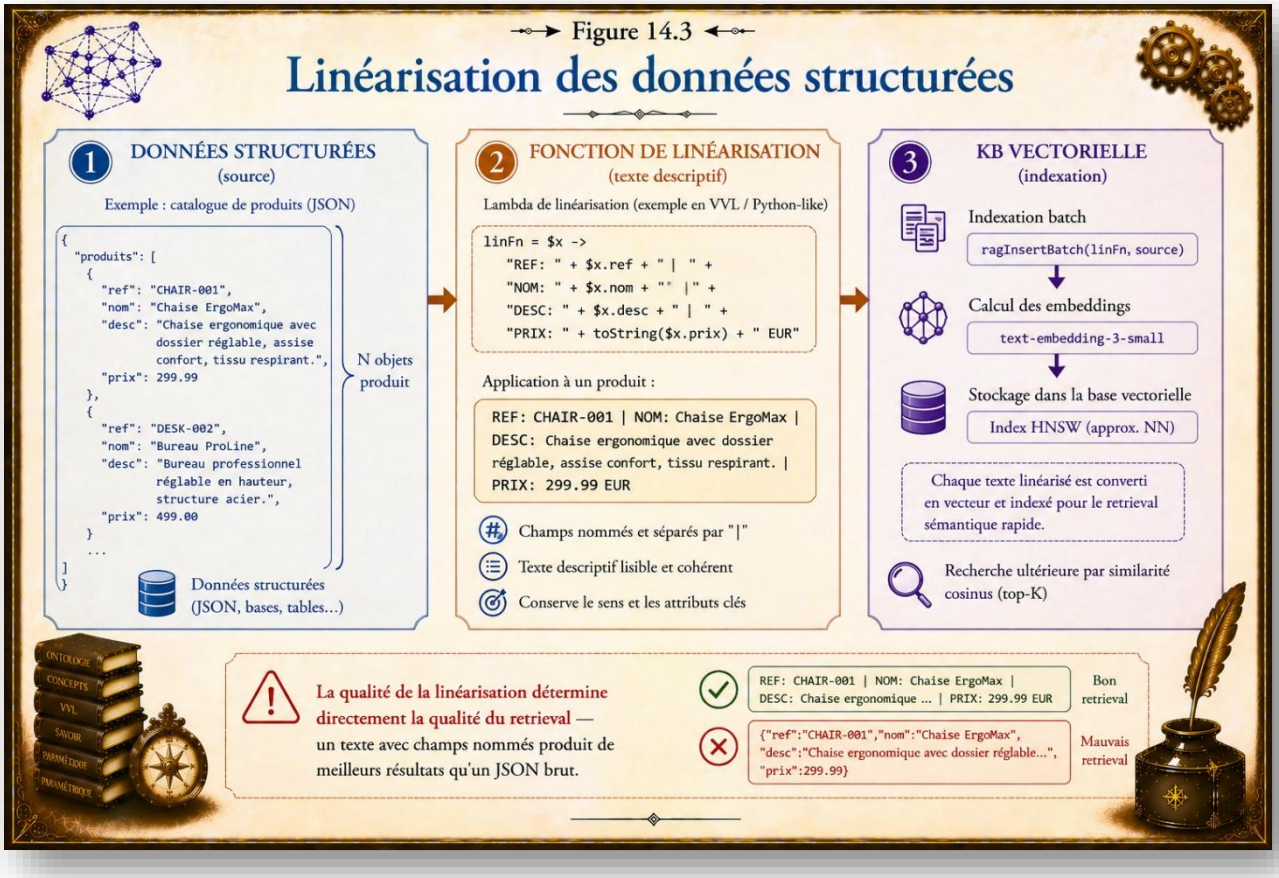


Figure 14.3 — *Linéariser, c'est traduire des données structurées en texte que le RAG peut indexer*

14.4. Quatre modes de requête

VVL offre quatre modes de requête qui couvrent un spectre allant de la **recherche pure** (sans LLM, gratuite) à l'**extraction structurée** (avec LLM, plus coûteuse mais plus précise). Le choix du mode dépend du compromis recherché entre précision, coût et format du résultat.

Mode	Coût LLM	Précision	Format retour	Cas d'usage
ragSearchContext	Gratuit	Bonne	Texte brut	Injection de contexte, vérification
ragAskSimple	Standard	Bonne	Réponse NL	Questions simples
ragAsk	Standard	Très bonne	Réponse NL	Questions complexes (défaut)
ragAskAdvanced	Standard	Excellente	Réponse NL	Requêtes techniques
ragExtractJson	Standard	Excellente	JSON structuré	Données exploitables

Figure 14.4 — *Compromis coût/précision/format des cinq modes de requête*

```
# Mode 1 : ragSearchContext — chunks bruts, pas d'appel LLM
$context = [ragSearchContext "products" "ecran haute resolution"]
[log $context]
# => Texte brut des chunks les plus proches : "REF: MON-001 | NOM: Ecran 27 4K IPS..."

# Mode 2 : ragAskSimple — reponse NL rapide, sans extraction de mots-cles
$reponse_simple = [ragAskSimple "myKB" "Quelle est la capitale de la France ?"]
[log $reponse_simple]                          # => Paris

# Mode 3 : ragAsk — hybride par defaut (vectoriel + mots-cles + LLM)
$reponse = [ragAsk "products" "Je cherche un ecran 4K pour le gaming"]
[log $reponse]
# => "L'Ecran 27 4K IPS a 449.99 EUR convient bien pour le gaming
#     grace a son 4K UHD et son support USB-C..."

# Mode 4 : ragAskAdvanced — avec extraction de mots-cles techniques
$reponse_tech = [ragAskAdvanced "docs" "erreur CORS sur endpoint REST cross-origin"]
# Le moteur extrait les termes specialises avant la recherche

# Mode 5 : ragExtractJson — extraction JSON conforme a un schema
$json = [ragExtractJson "products"
    "tous les produits de moins de 100 EUR"
    "[{\"nom\":\"\", \"prix\":0}]"
]
[log $json]
# => [{"nom": "Lampe LED Architecte", "prix": 45.99}]

# Le JSON est exploitable directement par le programme
$objets = [jsonToList $json]
[for $o $objets [log $o.nom " : " $o.prix " EUR"]]
```

Le mode `ragExtractJson` mérite une attention particulière : il représente le **pont entre RAG et programmation structurée**. En fournissant au LLM un schéma JSON attendu, on transforme le RAG d'un système conversationnel en un système d'extraction de données. Le LLM peuple le schéma à partir des chunks récupérés, et le résultat est immédiatement exploitable par les builtins JSON ou par les commandes ontologiques du chapitre 12. C'est ce qui permet de construire des pipelines `RAG → ontologie → règles métier` en quelques lignes.

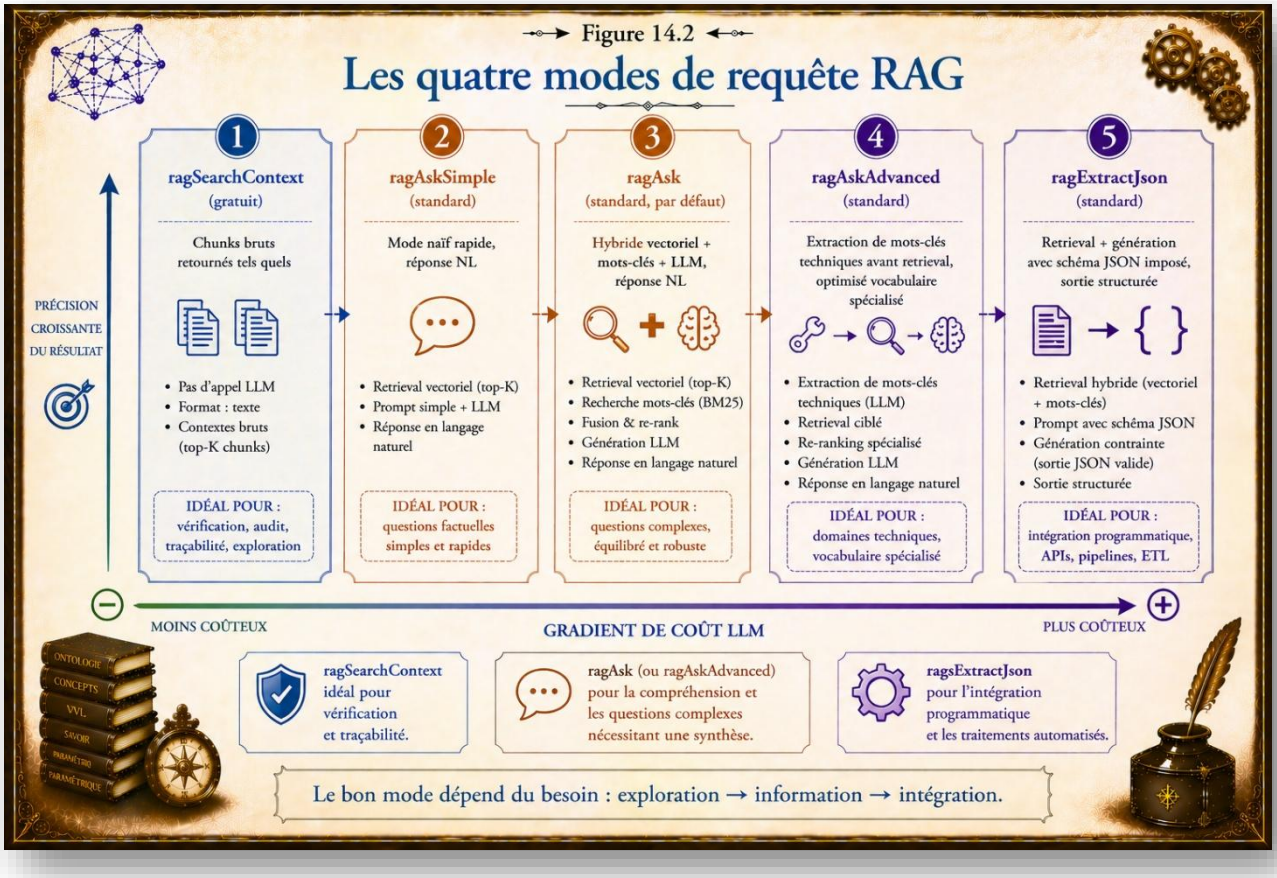


Figure 14.5 — *Le spectre des modes de requête : choisir selon le compromis désiré*

14.5. CRUD documentaire : maintenir la base

Une base de connaissances RAG **n'est pas figée** : les documents évoluent (mises à jour de catalogues, corrections d'erreurs, ajouts de nouveaux contenus) et la base doit pouvoir suivre ces évolutions sans reconstruction complète. VVL fournit un ensemble complet de commandes CRUD qui opèrent au niveau du document individuel via son identifiant `doc-xxxxx`.

```
# Listing : voir tous les documents indexes
$docs = [ragDocs "products"]
# => liste de {id, summary, length, status}

# Lecture : recuperer le contenu complet d'un document
$content = [ragDocContent "products" "doc-00003"]
[log $content]

# Recherche par texte (case-insensitive, sur le contenu brut)
$matches = [ragSearchDocs "products" "ergonomique"]

# Statistiques de la base
$stats = [ragInfo "products"]
# => {total_docs, total_chunks, last_modified, ...}

# Mise a jour ciblee par ID (recommande)
[ragUpdateById "products" "doc-00003"
    "Chaise ErgoMax Pro 2025 : nouvelle version avec accoudoirs 4D, 349 EUR"]

# Suppression par ID
```

```
[ragRemoveById "products" "doc-00007"]

# Vidage complet (irreversible)
[ragClear "products"]

# Fermeture explicite de la connexion (libere les ressources)
[ragClose "products"]
```

Trois remarques sur ces commandes. **Premièrement, le niveau d'opération est le document, pas le chunk** — le découpage en chunks est entièrement automatique et invisible pour l'utilisateur ; on insère et on supprime des documents entiers. **Deuxièmement, `ragUpdateById` est préféré à `ragUpdate`** : la mise à jour par ID est rapide et fiable, alors que la mise à jour par contenu (qui doit retrouver le document par son texte exact) peut échouer en cas de différence de formatage. **Troisièmement, `ragClear` est irréversible** : il supprime tous les documents et les vecteurs associés. Pour un environnement de développement, on l'utilise volontiers pour réinitialiser la base ; en production, on lui préfère le suppression sélective.

14.6. Genius : la documentation VVL comme RAG

Genius est l'application canonique du RAG dans VVL : c'est le système d'aide intégré du serveur, construit entièrement sur l'infrastructure RAG décrite dans ce chapitre. Genius indexe la documentation officielle de VVL — fichiers Markdown du dossier `help/kb/` — dans une base vectorielle dédiée, puis répond aux questions du développeur en langage naturel, en s'appuyant exclusivement sur cette documentation.

Au-delà de son utilité immédiate (avoir un assistant intégré qui répond aux questions du langage), Genius illustre un **pattern architectural réutilisable** : toute application VVL peut construire son propre « assistant expert » en créant une KB spécialisée et en l'interrogeant via `ragAsk`. C'est ainsi qu'on bâtit des assistants juridiques, médicaux, techniques, sans aucun fine-tuning de LLM — juste avec une bonne organisation documentaire et le bon mode de retrieval.

```
# Prerequis : indexer la documentation (une seule fois)
# Au REPL : :genius-learn

# Utilisation dans un script VVL
$help = [genius "comment iterer sur une liste ?"]
[log $help]

# Mode naif (plus rapide, moins precis sur termes specialises)
$help2 = [genius "commandes de persistance OO" "naive"]

# Reinitialiser et re-indexer (apres mise a jour des docs)
# Au REPL : :genius-init
```

Une particularité importante : **Genius alimente l'agent IA** (`[agent]` du chapitre 22). À chaque appel d'agent, les cinq chunks RAG les plus pertinents pour l'objectif courant sont automatiquement injectés dans le prompt système, permettant au LLM de générer du code VVL correct sans connaître le langage par entraînement. L'agent autonome n'est donc pas un système qui *connaît* VVL — c'est un système qui *recherche* la documentation pertinente à chaque pas et l'utilise comme contexte. Cette architecture rend l'agent **adaptable à toute évolution du langage** : il suffit de réindexer la documentation pour que l'agent connaisse les nouveautés.

14.7. Synergie RAG + ontologie

Le RAG et le système d'ontologie de VVL (chapitre 12) sont complémentaires, pas concurrents. **Le RAG excelle dans la recherche textuelle floue** sur de grands volumes de documents non structurés — ce qui est sa raison d'être. **L'ontologie excelle dans le raisonnement formel** sur des données structurées avec des contraintes typées — ce que le RAG ne sait pas faire. La force de VVL est de pouvoir composer les deux dans le même pipeline, sans avoir à passer par des couches d'intégration.

Le pattern type d'intégration combine `ragExtractJson` du chapitre courant avec `ontology:fromText` du chapitre 12. La première commande recherche dans le RAG les documents pertinents et en extrait un JSON structuré ; la seconde commande convertit ce JSON en instances ontologiques typées qui peuvent ensuite participer à l'inférence. Le résultat est une **chaîne complète** allant du texte brut indexé jusqu'aux conclusions déductives.

```
# 1. Une base RAG remplie de descriptions de candidats CV
[ragInsertBatch "candidats" $textes_cv]

# 2. Definition d'un schema ontologique
$RH = [ontology:define "RH"
    [ontology:class Candidat
        [attributes
            [nom         String]
            [age         Integer [min 18]]
            [experience  Integer]
            [competences String]
        ]
    ]
]

# 3. Extraction structuree depuis le RAG vers le schema
$json = [ragExtractJson "candidats"
    "tous les candidats avec experience en machine learning"
    "[{\"nom\":\"\", \"age\":0, \"experience\":0, \"competences\":\"\"}]"
]

# 4. Conversion en instances ontologiques
$candidats = [jsonToList $json]
[for $c $candidats [do
    $instance = [ontology:fromText $RH "Candidat" [json $c]]
    [ontology:commit $instance]
]]

# 5. Application des regles metier
[ontology:rule "ProfilSenior" $RH
    [target "Candidat"]
    [priority 50]
    [when ($this.experience > 5)]
    [then [ontology:set $this "niveau" "senior"]]
]
[ontology:infer-fixpoint $RH 5]

# 6. Requete finale combinant RAG et raisonnement
$seniors = [ontology:select Candidat niveau "senior"]
```

Cette synergie est ce qui distingue VVL des frameworks classiques de RAG (LangChain, LlamaIndex). Dans ces frameworks, le RAG est un service externe ; tout ce qu'on en fait après doit être codé manuellement avec des bibliothèques de validation, des règles d'inférence, et des couches d'intégration. Dans VVL, **les sorties du RAG entrent directement dans le tissu symbolique du langage** : elles deviennent des instances typées, gouvernées par des règles, interrogeables par Prolog, comparables avec les opérateurs Zalta. Le pipeline neuro-symbolique entier tient en quelques dizaines de lignes.

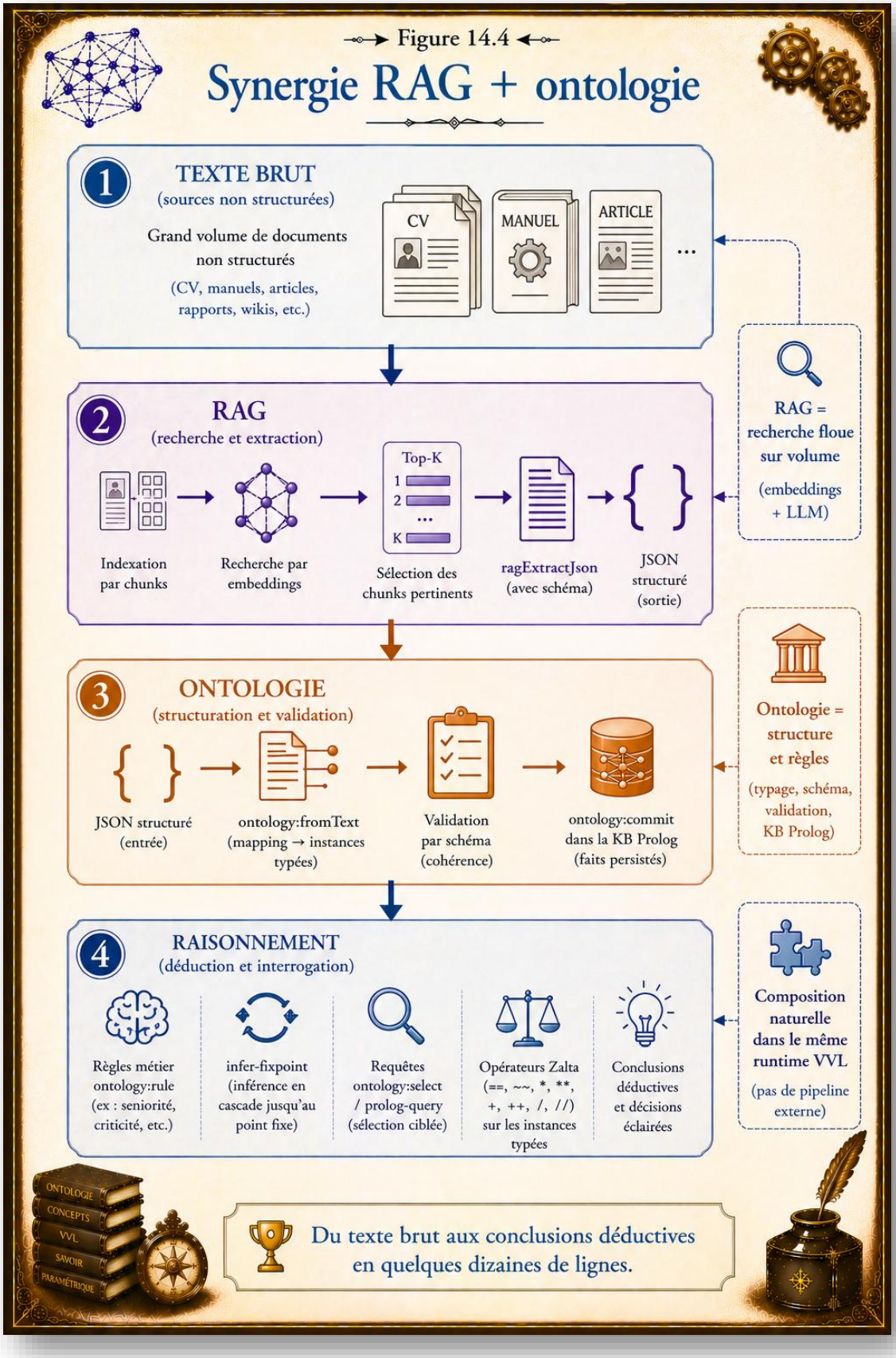


Figure 14.6 — *La chaîne complète : du texte brut aux conclusions déductives*

Avec ce chapitre, le premier outil neuronal de la Partie V est en place. Le **chapitre 15** s'attache à un usage plus ciblé du LLM — l'**extraction structurée** —, qui prend une donnée non structurée (un PDF, une image, un texte libre) et en tire un JSON conforme à un schéma. Là où le RAG est conçu pour la recherche dans un corpus indexé, l'extraction est conçue pour la conversion ponctuelle d'un document unique. Les deux capacités se complètent et constituent ensemble les deux modes principaux par lesquels le LLM alimente le système symbolique.

15. Extraction structurée par LLM

Mots-clés : extraction, LLM comme parser, opérateur /, opérateur %, opérateur ?, scrape, JSON, vision IA, pdfToImages, ontology:fromText, pipeline d'extraction, OCR sémantique

Le chapitre 14 a présenté le RAG comme outil de **recherche** dans un corpus indexé. Ce chapitre 15 traite l'autre grand usage du LLM dans l'alimentation du système symbolique : l'**extraction structurée**. Là où le RAG est conçu pour interroger plusieurs documents simultanément, l'extraction est conçue pour convertir un document unique — un texte libre, une page web, un PDF, une image — en données exploitables par le programme.

La différence est essentielle. Le RAG opère sur un *corpus* permanent ; l'extraction opère sur un *document* ponctuel. Le RAG retourne une réponse en langage naturel ou un JSON synthétisé ; l'extraction retourne directement les données structurées du document source. Le RAG est utile pour les questions ouvertes (« quels produits avons-nous sous 100 euros ? ») ; l'extraction est utile pour les conversions précises (« quelles sont les lignes de cette commande ? »).

VVL traite cette deuxième tâche par une famille d'**opérateurs sémantiques d'extraction** — `/`, `%`, `?` — qui transforment le LLM en *parser universel* : peu importe la source (texte HTML scrapé, image scannée, PDF de devis), une seule expression suffit pour en tirer une structure manipulable. Ces opérateurs se composent naturellement avec `JSON()`, `pdfToImages`, `scrape`, et avec les commandes ontologiques du chapitre 12. Le chapitre suit la séquence : LLM comme OCR sémantique (15.1), trois opérateurs d'extraction (15.2), scraping web (15.3), images et PDF (15.4), extraction typée par schéma ontologique (15.5), pattern e-commerce de bout en bout (15.6), synthèse RAG vs extraction (15.7).

15.1. Le LLM comme moteur d'OCR sémantique

Les systèmes d'OCR classiques (Optical Character Recognition) reconnaissent les caractères dans une image et produisent du texte brut. Ils ne *comprennent* rien — ils ne distinguent pas un nom de fournisseur d'une référence produit, un prix d'un identifiant, une adresse d'une signature. Le travail de structuration retombe sur le développeur, qui doit écrire des expressions régulières fragiles et des heuristiques fragiles pour chaque format de document rencontré. Quand le format change, tout le code casse.

Les LLM modernes — particulièrement ceux dotés de capacités de vision (GPT-4V, Claude vision, Gemini) — vont radicalement plus loin : ils **comprennent la structure sémantique** du document. Ils peuvent recevoir l'image d'un bon de commande et identifier le client, les lignes, les quantités, les prix, la TVA, les conditions de livraison. Ils peuvent recevoir un CV en PDF et en extraire les expériences professionnelles avec dates, employeurs, intitulés. Ils peuvent recevoir une page web HTML brute et la structurer en objets propres. Le LLM est devenu un **parser universel** capable de s'adapter à tout format sans configuration préalable.

La force de VVL est d'exposer cette capacité non pas comme une bibliothèque externe avec API à apprendre, mais comme des **opérateurs natifs du langage**. Une fois compris le principe (`source / [§ "description du résultat attendu"]`), le développeur peut traiter n'importe quel format avec la même syntaxe, et **composer** ces opérateurs avec le reste du langage — boucles, persistance, parallélisme, ontologie.

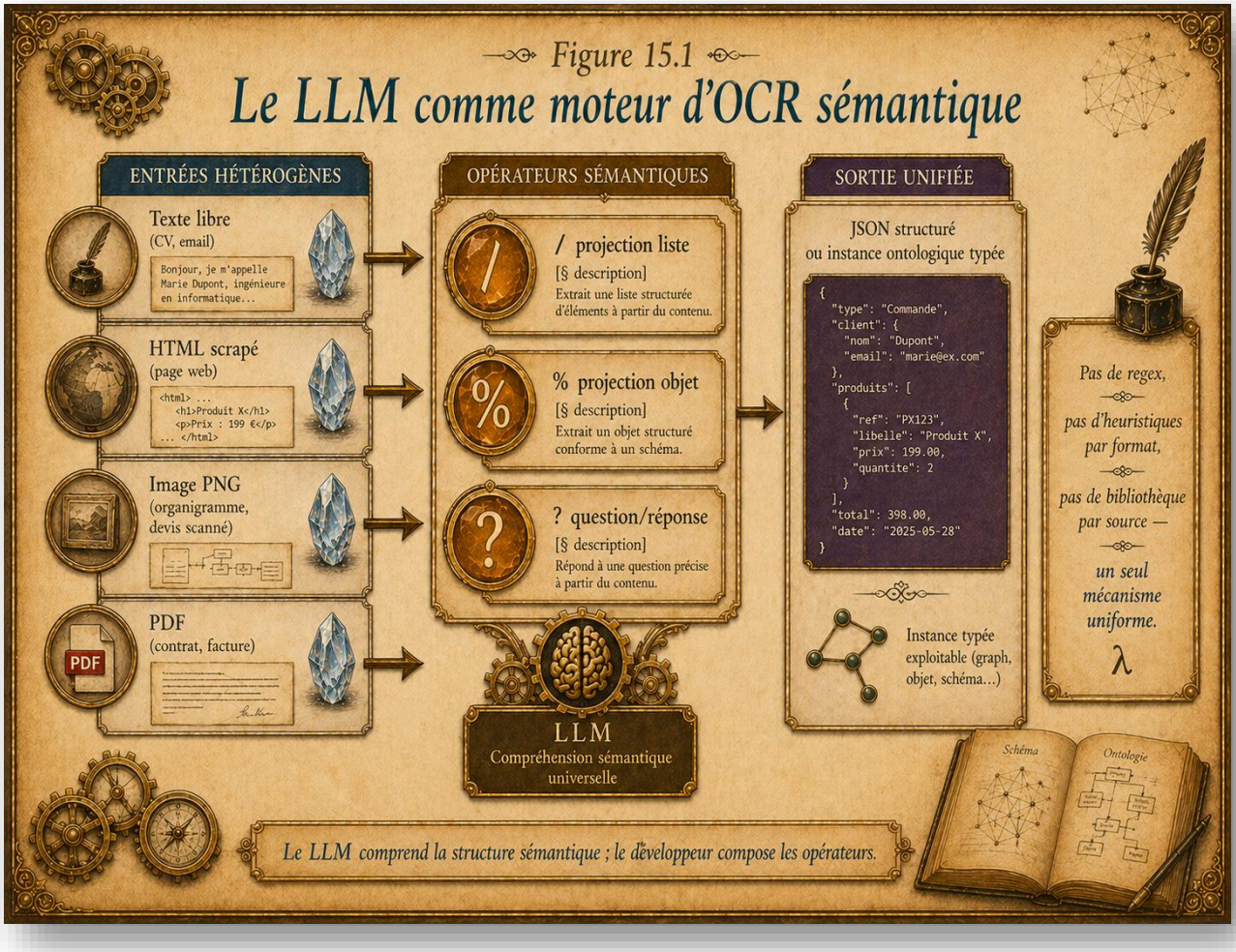


Figure 15.1 — *Le LLM comme parser universel : entrées hétérogènes, sortie structurée*

15.2. Les trois opérateurs sémantiques d'extraction

Trois opérateurs binaires, déjà introduits dans la liste des vingt opérateurs sémantiques du chapitre 7, méritent ici un traitement détaillé car ils constituent l'**outillage de base de l'extraction** : `/` (projection liste), `%` (projection objet unique), `?` (question/réponse). Tous trois prennent à gauche une donnée source (chaîne, objet, image, document) et à droite une description sémantique `[§ "..."]` qui guide le LLM vers le résultat attendu.

Opérateur	Sémantique	Sortie typique	Cas d'usage canonique
source / [§ "..."]	Projection structurée → liste	STRING (JSON list)	Extraire plusieurs éléments d'un document
source % [§ "..."]	Projection structurée → objet	STRING (JSON objet)	Extraire un seul objet d'un document
source ? [§ "..."]	Question / réponse	STRING (texte libre)	Poser une question sur le contenu

Figure 15.2 — *Les trois opérateurs sémantiques d'extraction*

```
# Opérateur / : extraction d'une LISTE
$html = [scrape "https://shop.example.com/laptops"]
$products = $html / [§ "extraire la liste des laptops avec nom et prix"]
[log $products]
# => "[{\"name\": \"Laptop Pro\", \"price\": 1999}, ...]"

# Opérateur % : extraction d'un OBJET unique
$invoice = [scrape "https://example.com/invoice/123"]
$header = $invoice % [§ "extraire l'en-tete : numero, date, client"]
[log $header]
# => "{\"numero\": \"FA-2026-001\", \"date\": \"2026-04-15\", \"client\": \"...\"}"

# Opérateur ? : question/réponse
$contrat = [pdfToText "contrat.pdf"]
$reponse = $contrat ? [§ "quelle est la date d'echeance ?"]
[log $reponse]                          # => "31 decembre 2026"

# Conversion en objet exploitable via JSON()
$json = [JSON $products]
$liste = [jsonToList $json]
[for $p $liste [log $p.name " : " $p.price " EUR"]]
```

Trois remarques sur l'usage. **Premièrement, le retour est toujours une chaîne** — c'est ce qui sort du LLM. Pour le manipuler comme une structure, on enchaîne avec `[JSON ...]` et éventuellement `[jsonToList ...]`. Cette séparation permet d'inspecter le résultat brut du LLM en cas de problème de format. **Deuxièmement, la description sémantique guide le LLM** mais n'est pas une expression régulière : on indique *ce qu'on veut*, pas *comment l'obtenir*. Plus la description est précise (champs nommés, types attendus, exemples), plus le résultat est fiable.



Troisièmement, ces opérateurs bénéficient du cache sémantique présenté au chapitre 7 — la même description appliquée aux mêmes données ne déclenchera qu'un seul appel LLM, même si l'expression est répétée.

15.3. Pipeline de scraping web

Le scraping web est l'usage le plus immédiat des opérateurs d'extraction. Les builtins `scrape` (récupération du contenu textuel d'une URL) et `scrapeLinks` (extraction des liens d'une page) servent d'entrée ; la composition avec `/` et `JSON()` produit en une expression ce qui demande typiquement 50+ lignes Python avec `requests`, `BeautifulSoup`, et un appel manuel à l'API du LLM.

```
# Scraping simple : recuperer le texte d'une page
$contenu = [scrape "https://example.com/article"]

# Extraire les liens d'une page
$liens = [scrapeLinks "https://shop.example.com/catalog"]
[log [length $liens] " liens trouves"]

# Filtrer les URLs de produits
$productLinks = [filter [lambda [url]
    [strContains $__url "/product/"]
] $liens]

# Pipeline complet : scrape -> extraction LLM -> JSON typé en une expression
$json = [JSON [STRING [scrape "https://shop.example.com/laptops"]
                / [§ "generer un tableau JSON des laptops avec nom et prix"]]]

# Enchaînement avec filtrage applicatif
$rawList = [jsonToList $json]
$cheap   = [filter [lambda [p] ($__p.price < 1000)] $rawList]
[for $p $cheap [log $p.name " : " $p.price " EUR"]]
```

Le pattern combiné `[JSON [STRING source / [§ "..."]]]` est suffisamment fréquent pour mériter d'être mémorisé. Il dit littéralement : « prends la source, demande au LLM d'en extraire un tableau JSON selon la description, convertis le résultat en chaîne, parse-le comme JSON ». La sortie est immédiatement utilisable par les builtins de manipulation de listes.

Pour les sites qui chargent leur contenu dynamiquement en JavaScript, `scrape` (qui ne fait qu'une requête HTTP simple) ne suffit pas. Le pattern recommandé combine alors `scrape` avec un fallback Selenium via l'opérateur de tolérance `~>` du chapitre sur la concurrence :

```
# Pattern : scrape simple, fallback sur Selenium si la page est dynamique
$pages = [pmap [lambda [url] [do
    [scrape $__url]                          # essai 1 : HTTP simple
    ~> [do                                   # fallback si echec
        $wd = [webdriver chrome]
        [go $__url]
        [wait ".product-detail" 5]
        $html = [text ".product-detail"]
        [webQuit]
        $html
    ]
]] $productLinks]

# Structuration LLM en parallele
$products = [map [lambda [html]
    [JSON ($__html % [§ "extraire nom, prix et description du produit"])]
] $pages]
```

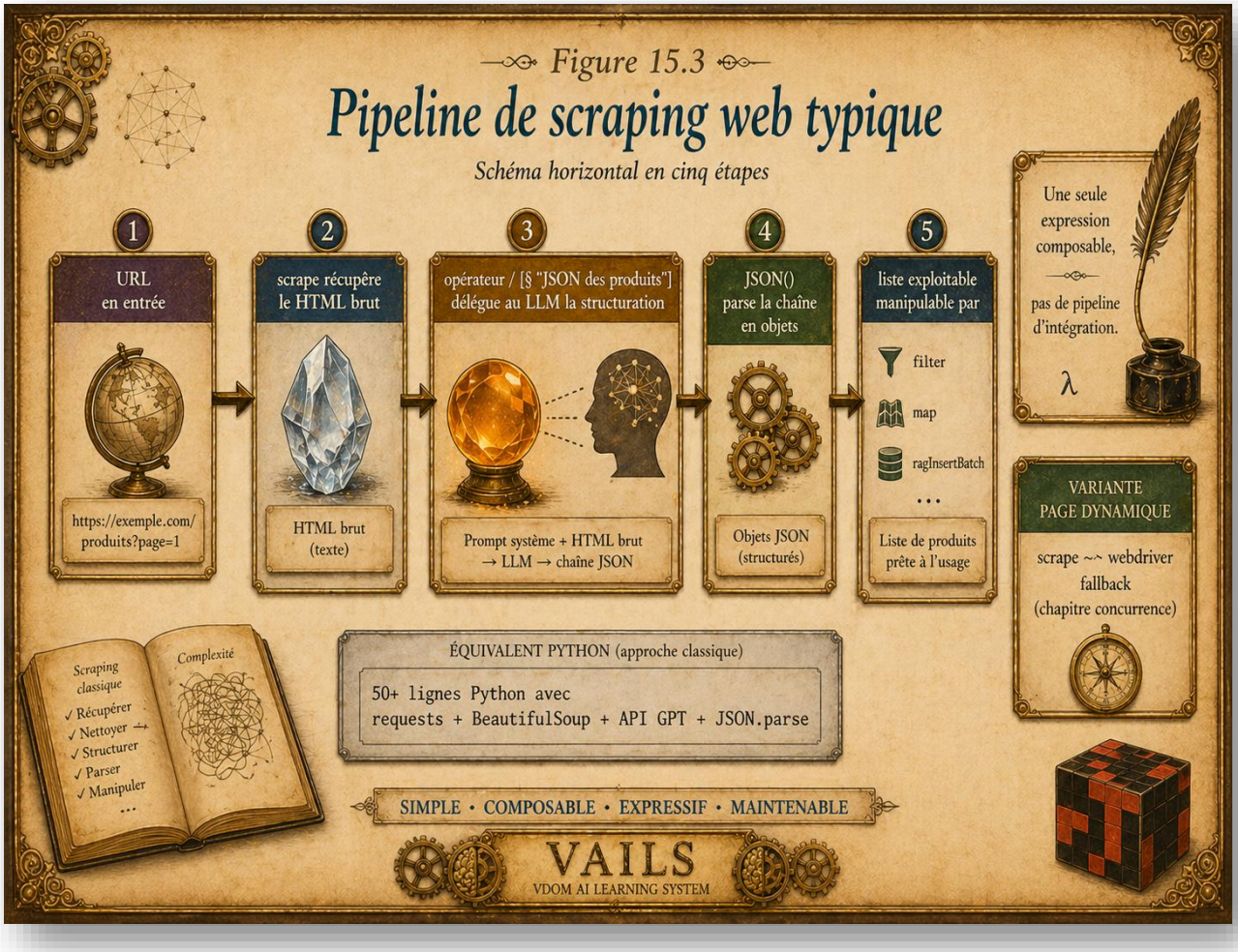


Figure 15.3 — *Du URL à la liste typée en une expression : la composition VVL*

15.4. Extraction depuis images et PDF

Les opérateurs `/` et `%` ne sont pas limités au texte : ils acceptent aussi des **images** et des **PDF** comme opérande gauche, à condition d'avoir un modèle de vision configuré (GPT-4V, Claude vision, Gemini). Cette capacité est ce qui permet de traiter des documents scannés, des organigrammes, des tableaux complexes, des bons de commande manuscrits. Le pipeline canonique passe par `pdfToImages` qui convertit chaque page d'un PDF en image, puis applique les opérateurs sémantiques.

```
# Conversion d'un PDF en images (une par page)
$pdf = "commandes/BC-2026-001.pdf"
$images = [pdfToImages $pdf 2.0]              # zoom x2 pour la qualite OCR
[log "PDF converti en " [length $images] " image(s)"]

# Extraction d'un objet unique depuis la premiere page (en-tete)
$image_p1 = [pop "head" $images]
$header = [JSON $image_p1 % [§ "extraire numero de commande, date,
                                nom et email du client"]]
[log $header.numero $header.date $header.client.nom]

# Extraction d'une LISTE depuis la meme image (lignes de commande)
$lignes = [JSON $image_p1 / [§ "extraire les lignes de commande
    sous forme [{ref, qty, description, prix_unitaire}]"]]
[log [length $lignes] " lignes extraites"]
```

```
[for $l [jsonToList $lignes]
    [log $l.ref " x " $l.qty " a " $l.prix_unitaire " EUR"]
]
```

Trois remarques sur le traitement des images. **Premièrement, le facteur de zoom de `pdfToImages`** influence la qualité de l'extraction : un zoom trop faible laisse les caractères flous et augmente les erreurs ; un zoom trop élevé augmente le coût (tokens d'image) sans bénéfice marginal. La valeur 2.0 est un bon compromis pour la plupart des documents. **Deuxièmement, le LLM voit l'image comme un humain** : il interprète la mise en page, les colonnes, les tableaux, les zones de signature. C'est ce qui le distingue radicalement d'un OCR pur qui ne ferait que transcrire le texte sans en comprendre l'organisation. **Troisièmement, les modèles de vision ont un coût en tokens d'image** plus élevé que pour du texte pur ; sur des PDF volumineux, on traite page par page plutôt qu'en lot.

15.5. Extraction typée par schéma ontologique

Quand l'extraction doit produire des objets **typés et validés**, plutôt que des JSON libres, on utilise les commandes ontologiques `ontology:fromText` et `ontology:fromImage` du chapitre 12. La différence avec les opérateurs `/` et `%` est triple : (1) le résultat est immédiatement une `VVL_OntologyInstance` typée, pas une chaîne JSON ; (2) les contraintes du schéma (types, `min`, `max`, `enum`, `required`) sont vérifiées par le moteur ; (3) l'instance peut être directement *committée* dans la KB Prolog pour participer au raisonnement.

```
# Definition d'un schema typé pour les CV
$RH = [ontology:define "RH"
    [ontology:class Candidat
        [attributes
            [nom         String  [required true]]
            [age         Integer [min 18] [max 70]]
            [email       String  [type Email]]
            [experience  Integer]
            [score       Integer [default 0]]
        ]
    ]
]

# Extraction typee depuis du texte brut
$cv = "Alice Martin, 29 ans, alice@example.com. 5 ans d'experience en Python et ML."
$candidat = [ontology:fromText $RH "Candidat" $cv]

# Tous les champs sont automatiquement valides selon le schema
[log $candidat.nom]              # => Alice Martin
[log $candidat.email]            # => alice@example.com (format Email valide)
[log $candidat.age]              # => 29 (contrainte 18 <= 29 <= 70 satisfaite)
[log $candidat.score]            # => 0 (default applique car absent du CV)

# Extraction typee depuis une image (vision IA)
$orgChart = [ontology:fromImage $RH "Personne" "photos/organigramme.png"]

# Commit dans la KB Prolog pour le raisonnement
[ontology:commit $candidat]
```

Le mécanisme interne est instructif : le LLM reçoit le **schéma complet** (classes, attributs, types, contraintes) avec le texte ou l'image, et est instruit de produire un JSON conforme. Si un champ requis est absent du document, le LLM tente de l'inférer du contexte (par exemple, déduire l'âge à partir de la date de naissance et de la date du document). Si une contrainte est violée — un email mal formé, un

âge hors limites —, la valeur est silencieusement ignorée pour éviter de propager des données corrompues. Le développeur peut inspecter l'instance après extraction pour vérifier ce qui a été retenu.

Le choix entre `JSON / [§ ...]` et `ontology:fromText` dépend du besoin de validation. Pour un usage *exploratoire*, où l'on veut juste voir ce que le LLM extrait, l'opérateur `/` est plus léger. Pour un usage *production*, où l'on veut garantir la conformité au schéma et participer au raisonnement, `ontology:fromText` est l'outil approprié. Les deux peuvent même être combinés : extraire d'abord en JSON libre pour explorer, puis définir un schéma quand la structure se stabilise.

15.6. Pattern : extraction → typage → action

Les opérateurs d'extraction prennent toute leur valeur quand on les compose avec les autres capacités du langage : recherche dans des référentiels (CSV), persistance, calcul, génération de rapport. L'exemple suivant illustre un **pipeline e-commerce de bout en bout** : un PDF de bon de commande est reçu, extrait par vision IA, le client est identifié dans une base CSV, les totaux sont calculés, et un rapport est généré. C'est un workflow réel qui combine six sous-systèmes — vision IA, JSON, CSV, calcul, fichier — en quelques dizaines de lignes.

```
# Pipeline complet : PDF -> vision IA -> CSV -> calcul -> rapport

# 1. Charger les referentiels CSV
$clients  = [CSV "data/clients.csv" ";"]
$articles = [CSV "data/articles.csv" ";"]

# 2. Extraire les donnees du PDF par vision IA
$doc    = [PDF "commandes/BC-2026-001.pdf"]
$images = [pdfToImages $doc 2.0]

# Extraction de l'en-tete (objet unique)
$header = [JSON ([pop "head" $images] % [§ "extraire numero, date,
                                           email du client"])]

# Extraction des lignes (liste)
$lignes = [JSON ([pop "head" $images] / [§ "extraire les lignes :
                                           [{ref, qty, description}]"])]

# 3. Identifier le client par lookup CSV
$contact = [csv-find $clients email $header.email]
[log "Client : " $contact.nom " - " $contact.ville]

# 4. Calculer le total avec lookup des prix en base
$total = 0
[for $line [jsonToList $lignes] [do
    $article = [csv-find $articles ref $line.ref]
    $montant = ($line.qty * $article.prix)
    $total   = ($total + $montant)
    [log $line.ref " x " $line.qty " = " $montant " EUR"]
]]
[log "Total commande : " $total " EUR"]

# 5. Generer le rapport texte
$rapport = ("COMMANDE " + $header.numero
          + "\nClient: " + $contact.nom
          + "\nTotal: " + [STRING $total] + " EUR")
[file-write "output/rapport.txt" $rapport]
```

Sept étapes — chargement, vision IA, identification, calcul, rapport — tiennent en moins de quarante lignes. Aucun pipeline d'intégration, aucun ORM, aucun parser ad-hoc. Chaque opération est un builtin natif qui se compose avec les autres par les opérateurs binaires. Cette densité expressive est la marque distinctive de VVL : ce que d'autres langages font en plusieurs centaines de lignes éclatées sur plusieurs fichiers tient ici en un script lisible d'un coup d'œil.

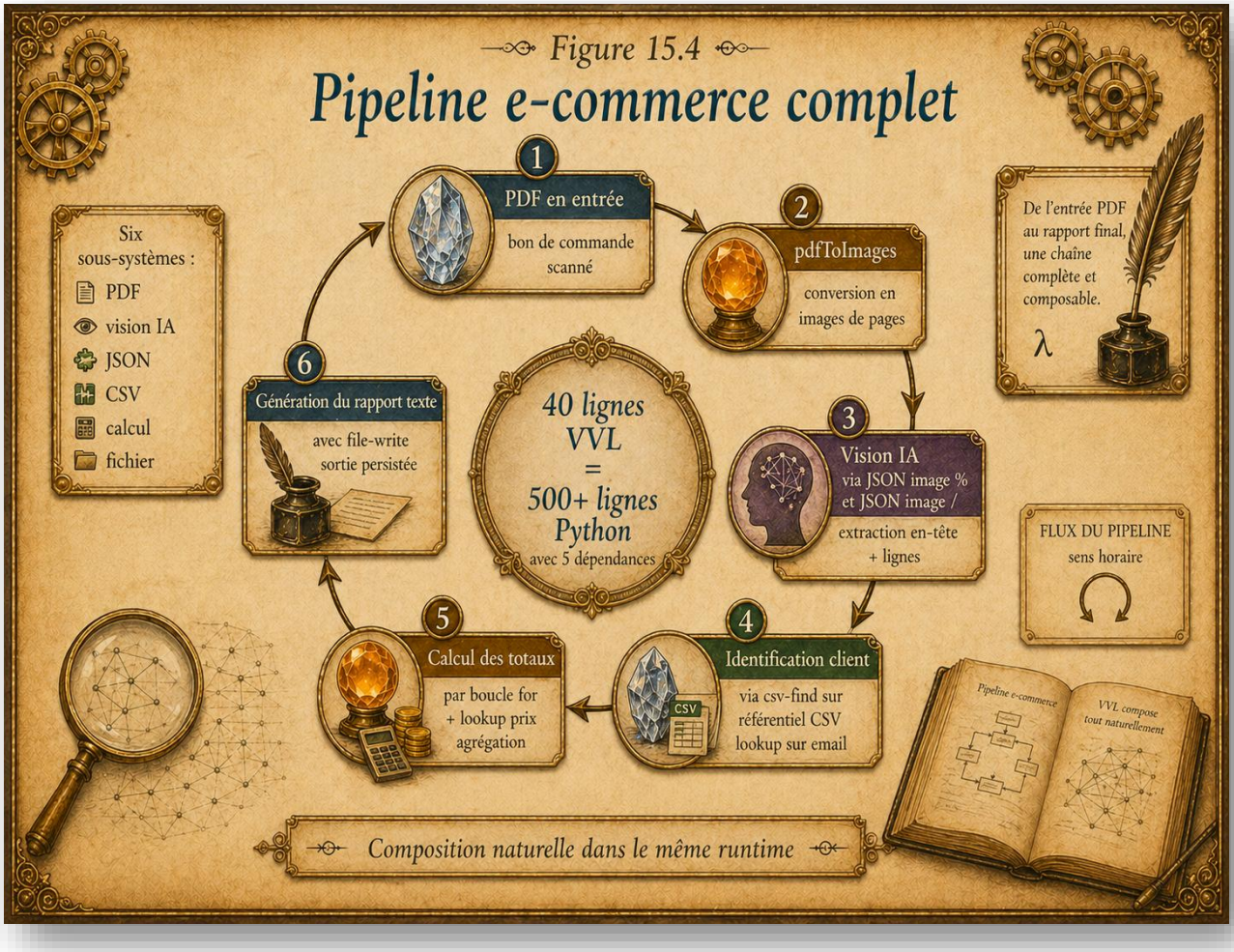


Figure 15.4 — *Six sous-systèmes orchestrés en quarante lignes de VVL*

15.7. Synthèse : RAG vs extraction

Avec ce chapitre, le deuxième grand outil neuronal de la Partie V est en place. Le tableau ci-dessous récapitule la complémentarité entre RAG (chapitre 14) et extraction (chapitre courant), pour aider à choisir le bon outil selon le contexte :

Critère	RAG (chapitre 14)	Extraction (chapitre 15)
Source	Corpus permanent (KB indexée)	Document unique (texte, image, PDF)
Préparation	Indexation préalable (embeddings)	Aucune (on traite à la volée)
Cas d'usage	Recherche dans plusieurs documents	Conversion d'un document en données

Critère	RAG (chapitre 14)	Extraction (chapitre 15)
Question type	« quels produits coûtent moins de 100€ »	« quelles sont les lignes de cette commande »
Coût LLM	1 embedding + 1 completion par requête	1 vision/text par document
Composition idéale	Avec ontology:rule (raisonnement)	Avec ontology:fromText (typage)

Figure 15.5 — *RAG et extraction : deux outils complémentaires*

Les deux capacités peuvent évidemment se combiner. Un pipeline réaliste indexe les documents reçus dans le RAG (pour la recherche ultérieure), tout en extrayant **simultanément** leurs données structurées (pour le traitement immédiat). Le coût LLM marginal est faible — on paye une fois la vision et l'embedding —, et l'on gagne à la fois la mémoire à long terme du RAG et l'action immédiate de l'extraction.

Le **chapitre 16** introduit une troisième capacité d'alimentation du système symbolique : la **génération d'ontologies dynamiques par Gherkin**. Là où ce chapitre a montré comment extraire des *instances* d'un schéma préexistant, le chapitre suivant montre comment **déduire le schéma lui-même** depuis des échantillons. C'est l'inverse du paradigme habituel — au lieu d'écrire le schéma puis d'extraire, on observe des exemples et on en infère le schéma. Cette inversion ouvre la voie à des systèmes qui *apprennent* leur structure depuis les données.

16. Gherkin et ontologie dynamique

Mots-clés : Gherkin, BDD, ontology:context, ontology:loadGherkin, ontology:learn, schéma déduit, programmation par l'exemple, induction de règles, auto-apprentissage, transpilation, pipeline neuro-symbolique

Les chapitres 12, 14 et 15 ont supposé que le développeur **connaît à l'avance** le schéma de ses données : il écrit `ontology:define` avec ses classes et ses attributs, puis il extrait. Mais dans la réalité métier — devis BTP, factures fournisseurs, contrats juridiques, dossiers médicaux —, la structure est souvent **inconnue** ou **variable**. Chaque émetteur a son format, chaque type de document a ses sections, et l'on découvre la diversité au fil du traitement plutôt qu'au moment de la conception.

Ce chapitre 16 introduit une **inversion paradigmatique** : au lieu d'écrire le schéma puis d'extraire, on **observe** des échantillons et on **infère** le schéma. C'est de la programmation par l'exemple inversée — au lieu de spécifier ce qu'on veut, on montre des cas et on demande au système d'en tirer la structure générale. Le format pivot de cette inversion est le **Gherkin**, langage de description hérité du Behavior-Driven Development (Cucumber, SpecFlow), qui a la propriété rare d'être à la fois lisible par les humains et transpilable en code exécutable.

Le chapitre suit la séquence : motivation et inversion paradigmatique (16.1), format Gherkin (16.2), transpilation par `ontology:loadGherkin` (16.3), déduction de schéma par `ontology:context` (16.4), induction de règles par `ontology:learn` (16.5), pipeline complet en 5 phases avec auto-apprentissage continu (16.6), synthèse du changement de paradigme (16.7).

16.1. Inversion paradigmatique : observer pour inférer

Le paradigme classique de l'ontologie — celui du chapitre 12 — suit une séquence linéaire : (1) le développeur analyse le domaine, (2) il écrit le schéma manuellement avec `ontology:define`, (3) il extrait les données depuis les documents avec `ontology:fromText`. Cette approche fonctionne quand on **maîtrise le domaine** : l'analyste métier sait qu'un devis a un numéro, une date, un client, des lignes ; il peut en écrire la structure une fois pour toutes.

Mais cette maîtrise préalable est l'exception, pas la règle. Dans la plupart des contextes réels, le développeur **découvre la structure** en regardant les premiers documents — et il s'aperçoit que les variations entre documents sont importantes : ce devis-ci a une colonne TVA, celui-là pas ; ce contrat a des conditions particulières que l'autre n'a pas. Écrire un schéma exhaustif *a priori* devient impossible, et l'on tombe dans un cycle vicieux : extraction → erreurs → mise à jour du schéma → ré-extraction.

L'inversion paradigmatique consiste à confier au LLM la tâche de découvrir la structure. La séquence devient : (1) collecter quelques échantillons représentatifs, (2) demander au LLM de **déduire le schéma** depuis ces échantillons, (3) extraire avec le schéma déduit. Le développeur passe d'« architecte » à « curateur d'exemples » — un changement de rôle considérable qui rapproche le développement de la programmation par l'exemple.

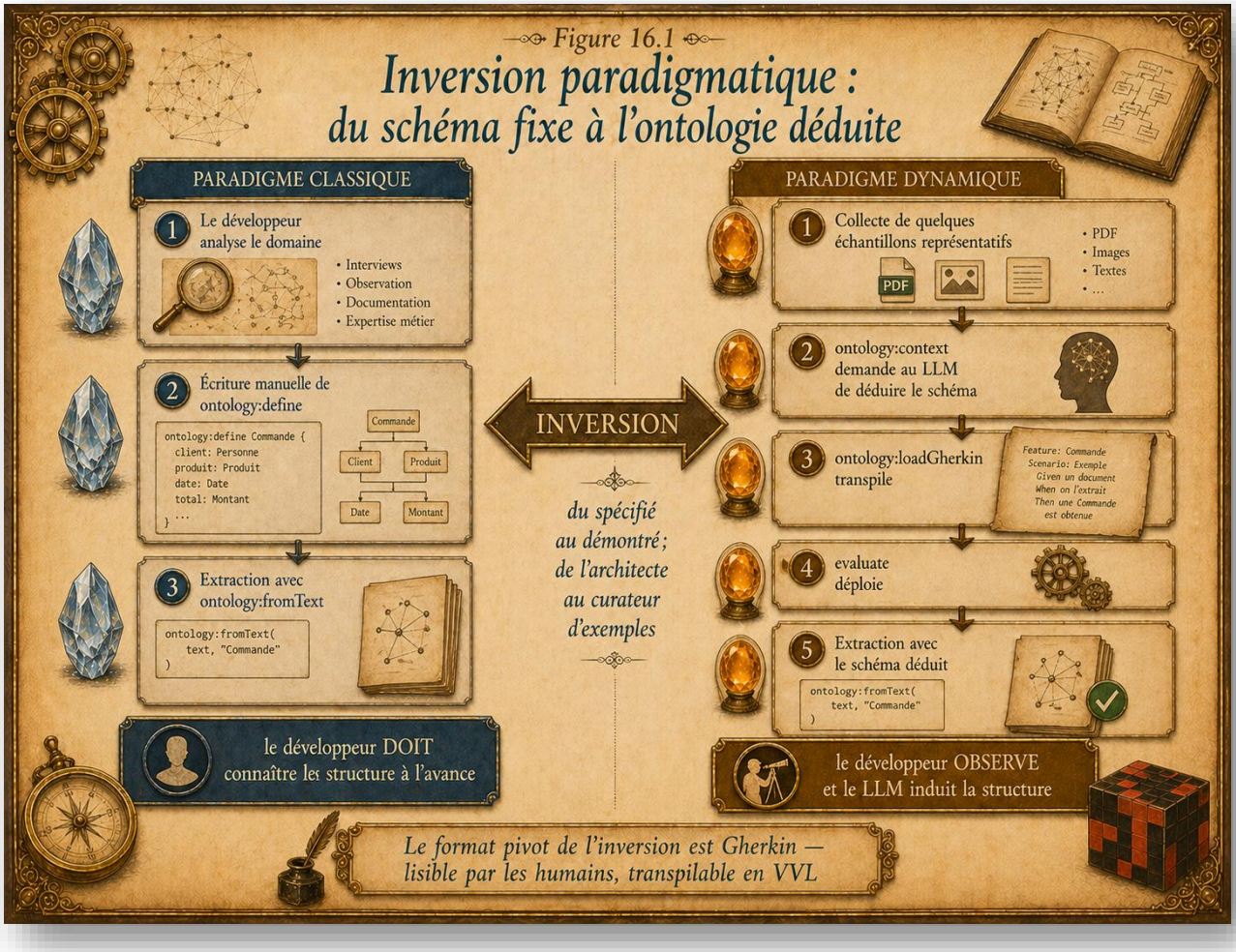


Figure 16.1 — *Du schéma écrit au schéma déduit : un changement de rôle pour le développeur*

16.2. Le format Gherkin : Background et Scenarios

Le **Gherkin** est un langage déclaratif né dans la communauté Behavior-Driven Development (BDD). Sa raison d'être originelle est de permettre aux **non-développeurs** — analystes métier, product owners, testeurs — d'écrire des spécifications exécutables sous forme de phrases naturelles structurées. La syntaxe repose sur quelques mots-clés (`Feature`, `Background`, `Scenario`, `Given`, `When`, `Then`) qui structurent des descriptions en langage naturel.

Dans VVL, Gherkin joue un rôle différent mais qui exploite la même propriété de **lisibilité humaine** : il sert de format intermédiaire entre le LLM (qui produit du Gherkin lors de la déduction) et le moteur ontologique (qui consomme le code VVL transpilé). Cette intermédiation est cruciale parce qu'elle rend la déduction du LLM **inspectable** par le développeur : avant de transpiler et d'exécuter, on peut lire le Gherkin produit, le corriger manuellement si besoin, ou même l'enrichir.

Background : définition de schéma

La section `Background` déclare le schéma ontologique. La syntaxe utilise des phrases `Given` / `And` qui décrivent les classes et leurs attributs sous forme de tableau Markdown. Le transpileur

`ontology:loadGherkin` reconnaît ces phrases et les convertit en commandes `ontology:define` et `ontology:class`.

```
Feature: BankSystem

  Background: Schema
    Given I define an ontology named 'BankSystem'
    And I define a class 'Transaction' with attributes:
    | Name   | Type    | Default   |
    | amount | Integer | 0         |
    | status | String  | 'Pending' |
    | client | String  | ''        |
```

Scenarios : règles métier

Les sections `Scenario` déclarent des règles métier. Chaque scénario suit le triplet **Given / When / Then** : `Given` indique la classe cible, `When` exprime la condition, `Then` exécute les actions. L'annotation `@priority: N` au-dessus du scénario fixe la priorité d'exécution lors de l'inférence. Le transpileur convertit chaque scénario en commande `ontology:rule` avec ses blocs `[when ...]` et `[then ...]`.

```
  @priority: 100
  Scenario: High Value Rule
    Given a 'Transaction' context
    When 'amount' is greater than 1000
    Then set 'status' to 'HighRisk'

  @priority: 50
  Scenario: Standard Rule
    Given a 'Transaction' context
    When 'amount' is less than 100
    Then set 'status' to 'Auto-approved'
    And log 'Transaction approuvee automatiquement'
```

Les mots-clés supportés par le transpileur sont stables et limités, mais suffisants pour la majorité des règles métier. Le tableau suivant en donne le lexique :

Mot-clé	Forme syntaxique	Effet
When (numérique)	'field' is greater than VALUE / less than / equals	Comparaison numérique
When (textuel)	'field' contains "text"	Test de sous-chaîne
Then set	'field' to 'value'	Affectation
Then add item	"value" to "list_field"	Ajout dans une liste
Then log	'message'	Trace dans la console
And	— continuation de When ou Then	Conjonction logique

Figure 16.2 — *Mots-clés Gherkin reconnus par le transpileur ontology:loadGherkin*

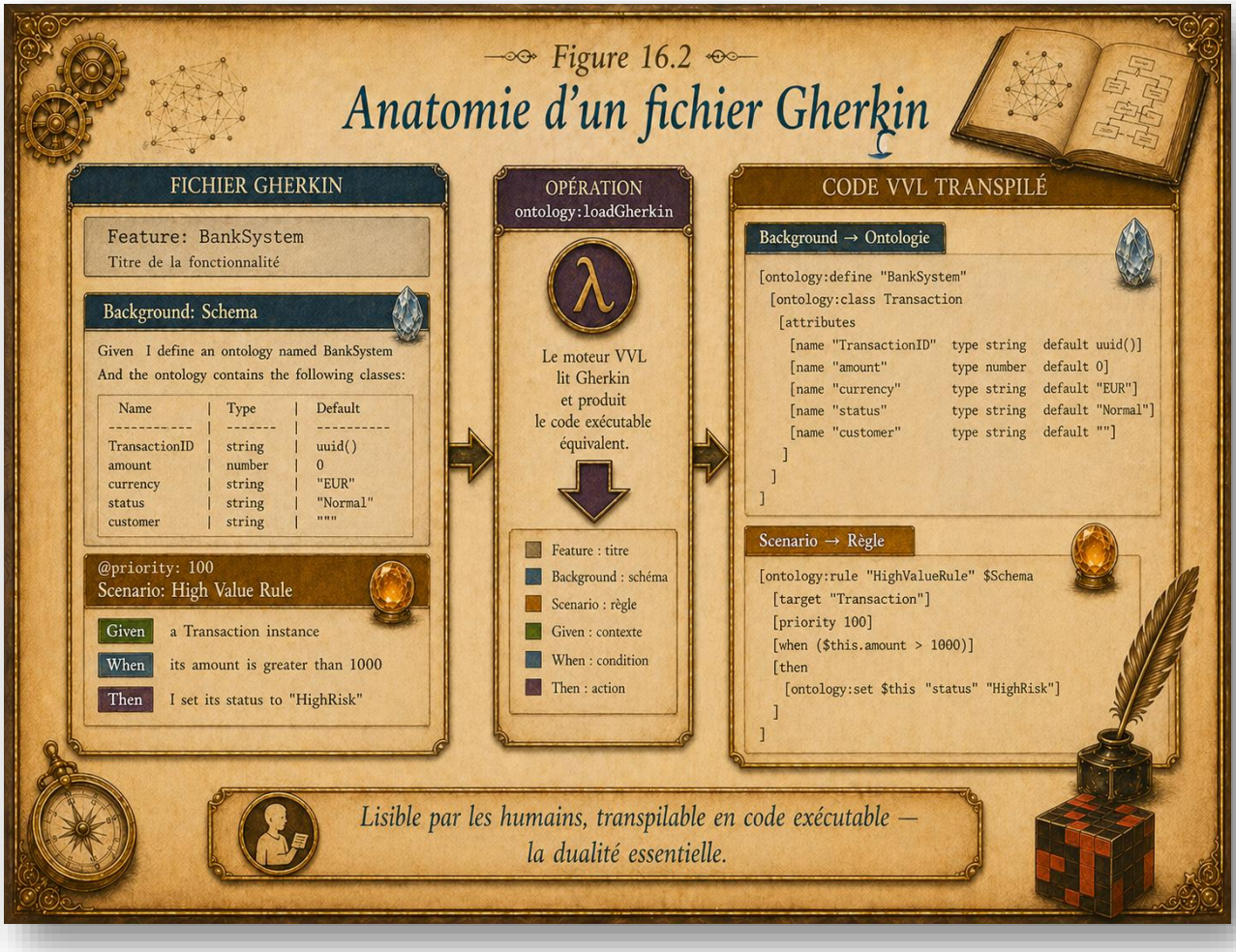


Figure 16.3 — *Anatomie d'un fichier Gherkin et sa transpilation en VVL*

16.3. ontology:loadGherkin : la transpilation

La commande `ontology:loadGherkin` prend une chaîne Gherkin (ou un chemin vers un fichier `.feature`) et **retourne du code VVL** sous forme de chaîne. Le code retourné n'est pas exécuté immédiatement — c'est au développeur de l'évaluer avec `[evaluate ...]` quand il est prêt. Cette séparation entre transpilation et évaluation permet d'inspecter le code généré, de le combiner avec d'autres morceaux, ou de le sauvegarder pour réutilisation.

```
# Source Gherkin definie dans une variable
$gherkin_source = "
Feature: BankSystem

  Background: Schema
    Given I define an ontology named 'BankSystem'
    And I define a class 'Transaction' with attributes:
    | Name   | Type    | Default   |
    | amount | Integer | 0         |
    | status | String  | 'Pending' |

  @priority: 100
  Scenario: High Value Rule
    Given a 'Transaction' context
    When 'amount' is greater than 1000
    Then set 'status' to 'HighRisk'
"

# 1. Transpilation : produit du code VVL sous forme de chaine
$vvl_code = [ontology:loadGherkin $gherkin_source]
[log $vvl_code]
# => Code VVL avec ontology:define + ontology:rule

# 2. Evaluation : execute le code et installe le schema + les regles
$Ontology = [evaluate $vvl_code]

# 3. Le systeme est pret a recevoir des instances
$tx = [ontology:fromText $Ontology "Transaction"
       "Virement de 5000 EUR de Alice vers Bob"]
[ontology:commit $tx]
[ontology:infer $Ontology]                        # applique High Value Rule
[log $tx.status]                                 # => HighRisk
```

Cette approche en deux temps — transpiler puis évaluer — a un autre avantage important : elle permet de **manipuler le code VVL généré comme une donnée** avant de l'exécuter. On peut concaténer plusieurs Gherkin (un produit par le LLM, un autre écrit à la main, un troisième chargé depuis un fichier de référence), passer le résultat à `loadGherkin`, et déployer en une fois. C'est ce qui rend l'auto-apprentissage continu possible — on accumule les règles déduites au fil du temps, et l'on redéploie périodiquement.

16.4. ontology:context : déduction de schéma depuis des échantillons

La commande `ontology:context` est la première moitié de l'inversion paradigmatique. Elle reçoit une **description du domaine** (en langage libre) et une **liste d'échantillons bruts** (images de PDF, textes, JSON), et retourne un Gherkin Background contenant les classes et attributs déduits par le LLM. Une option supplémentaire, le **namespace**, permet d'isoler plusieurs ontologies déduites dans la même session.

```
# 1. Recuperer des echantillons bruts depuis un serveur de documents
$ps = [proshare "https://docs.company.com/" "user" "pass"]
$files = [$ps "list" "/archives/devis"]

# 2. Convertir les PDF en images (parallele pour rapidite)
$raw_samples = [pmap [lambda [f]
    [pdfToImages $__f 1.0]
] $files]

# 3. Le LLM analyse les echantillons et deduit le schema en Gherkin
$gherkin_structure = [ontology:context "Structure des Devis BTP" $raw_samples]
[log $gherkin_structure]
# => Feature: Quotes
#    Background: Schema
#      Given I define an ontology named 'BTPSystem'
#      And I define a class 'Devis' with attributes:
#      | Name         | Type    | Default |
#      | numero       | String  | ''      |
#      | client_nom   | String  | ''      |
#      | date         | String  | ''      |
#      | total_ht     | Float   | 0.0     |
#      | produits     | List    | []      |
#      And I define a class 'LigneDevis' with attributes:
#      | Name           | Type    | Default |
#      | designation    | String  | ''      |
#      | quantite       | Float   | 0.0     |
#      | prix_unitaire  | Float   | 0.0     |

# 4. Charger le schema deduit
$vvl_code = [ontology:loadGherkin $gherkin_structure]
$Ontology = [evaluate $vvl_code]

# 5. Le systeme est pret pour l'extraction sur de nouveaux devis
$nouveau = [ontology:fromImage $Ontology "Devis" "nouveau-devis.pdf"]
```

Trois remarques sur l'usage. **Premièrement, la qualité de la déduction dépend de la diversité des échantillons** : trois devis du même fournisseur produiront un schéma très spécifique ; quinze devis de fournisseurs différents produiront un schéma plus général et robuste. La règle pratique est de fournir cinq à dix échantillons couvrant les variations attendues. **Deuxièmement, la description du domaine est cruciale** : « Structure des Devis BTP » oriente le LLM vers les attributs métier pertinents (TVA, lot, désignation), tandis qu'une description vague comme « Documents » produirait un schéma générique inutile. **Troisièmement, le namespace est utile en production** : il permet d'avoir simultanément une ontologie « devis-fournisseur-A » et « devis-fournisseur-B » sans collision de noms de classes.

16.5. ontology:learn : induction de règles métier

Si `ontology:context` déduit la **structure** (le schéma), `ontology:learn` déduit le **comportement** (les règles métier). La commande prend un dataset d'apprentissage sous forme de couples `{request, result}` — la requête métier en entrée, le résultat ontologique attendu en sortie — et retourne des Gherkin Scenarios contenant les règles que le LLM a su inférer. Une option `target_feature` permet de cibler la déduction sur un attribut particulier.

```
# Dataset d'apprentissage : 3 paires (requete metier, resultat ontologique)
$dataset = [
    <{"request": "Refaire etancheite terrasse 20m2",
      "result": {"type": "TERRASSE", "lignes": [
          {"designation": "DEPOSE EVACUATION",  "qte": 20, "pu": 100},
          {"designation": "PARE VAPEUR BITUME", "qte": 20, "pu": 50},
          {"designation": "BICOUCHE ETANCHEITE","qte": 20, "pu": 100}
      ], "tva": 10}}>,

    <{"request": "Fuite : reparer avec systeme liquide",
      "result": {"type": "REPARATION", "lignes": [
          {"designation": "NETTOYAGE PREPARATION", "qte": 2, "pu": 200},
          {"designation": "PRIMAIRE ACCROCHE",     "qte": 2, "pu": 200}
      ], "tva": 10}}>,

    <{"request": "Cool roof sur batiment industriel",
      "result": {"type": "COOLROOF", "lignes": [
          {"designation": "PEINTURE REFLECTIVE", "qte": 100, "pu": 30}
      ], "tva": 20}}>
]

# Induction des regles metier
$gherkin_rules = [ontology:learn $dataset]
[log $gherkin_rules]
# => @priority: 100
#    Scenario: Detect Terrasse
#      Given a 'Devis' context
#      When 'request' contains 'terrasse'
#      Then set 'type' to 'TERRASSE'
#      And add item 'DEPOSE EVACUATION' to 'lignes'
#      And add item 'PARE VAPEUR BITUME' to 'lignes'
#
# (regles similaires pour REPARATION et COOLROOF)

# Induction ciblee sur un attribut specifique
$gherkin_motor = [ontology:learn $dataset "type"]
# Le LLM se concentre sur les regles qui determinent le type
```

L'induction par `ontology:learn` repose sur une intuition statistique du LLM : observer ce qui change dans la sortie quand l'entrée change, et identifier les régularités sous forme de règles conditionnelles. Le résultat est nécessairement *imparfait* — le LLM peut sur-généraliser à partir de peu d'exemples, ou sous-généraliser et produire des règles très spécifiques. Le développeur doit toujours **inspecter** les Scenarios produits avant de les déployer, et idéalement les valider sur un jeu de test indépendant.

16.6. Pipeline complet : 5 phases neuro-symboliques

La combinaison de `ontology:context`, `ontology:learn` et `ontology:loadGherkin` produit un **pipeline complet** qui crée une ontologie opérationnelle à partir de zéro, depuis quelques documents et un dataset d'apprentissage. Ce pipeline alterne entre cinq phases — neuronales et symboliques —, ce qui en fait l'illustration la plus complète de l'oscillation neuro-symbolique propre à VVL.

#	Phase	Type	Commande	Sortie
1	Découverte du schéma	Neuro	ontology:context $desc $samples	Gherkin Background
2	Induction des règles	Neuro	ontology:learn $dataset $target?	Gherkin Scenarios
3	Transpilation	Symbolique	ontology:loadGherkin $gherkin	Code VVL
4	Déploiement à chaud	Symbolique	evaluate $code	Schéma + règles installés
5	Vérification	Neuro	ontology:fromImage $Ont $cls $img	Instance extraite

Figure 16.4 — *Les cinq phases du pipeline neuro-symbolique d'ontologie dynamique*

```
# === PHASE 1 (Neuro) : Decouverte du schema ===
$ps = [proshare "https://docs.btp.com/" "user" "pass"]
$files = [$ps "list" "/devis"]
$samples = [pmap [lambda [f] [pdfToImages $__f 1.0]] $files]
$gherkin_schema = [ontology:context "Devis BTP" $samples]

# === PHASE 2 (Neuro) : Induction des regles ===
$dataset = [ ... ]                                  # paires (request, result)
$gherkin_rules = [ontology:learn $dataset]

# === PHASE 3 (Symbolique) : Transpilation ===
$gherkin_complet = ($gherkin_schema + "\n" + $gherkin_rules)
$vvl_code = [ontology:loadGherkin $gherkin_complet]

# === PHASE 4 (Symbolique) : Deploiement a chaud ===
$Ontology = [evaluate $vvl_code]

# === PHASE 5 (Neuro) : Verification sur nouveau document ===
$nouveau_devis = [ontology:fromImage $Ontology "Devis"
                  "nouveau-devis-2026-04.pdf"]
[ontology:commit $nouveau_devis]
[ontology:infer $Ontology]                            # applique les regles induites
[log $nouveau_devis.type]                           # => "TERRASSE" si applicable
```

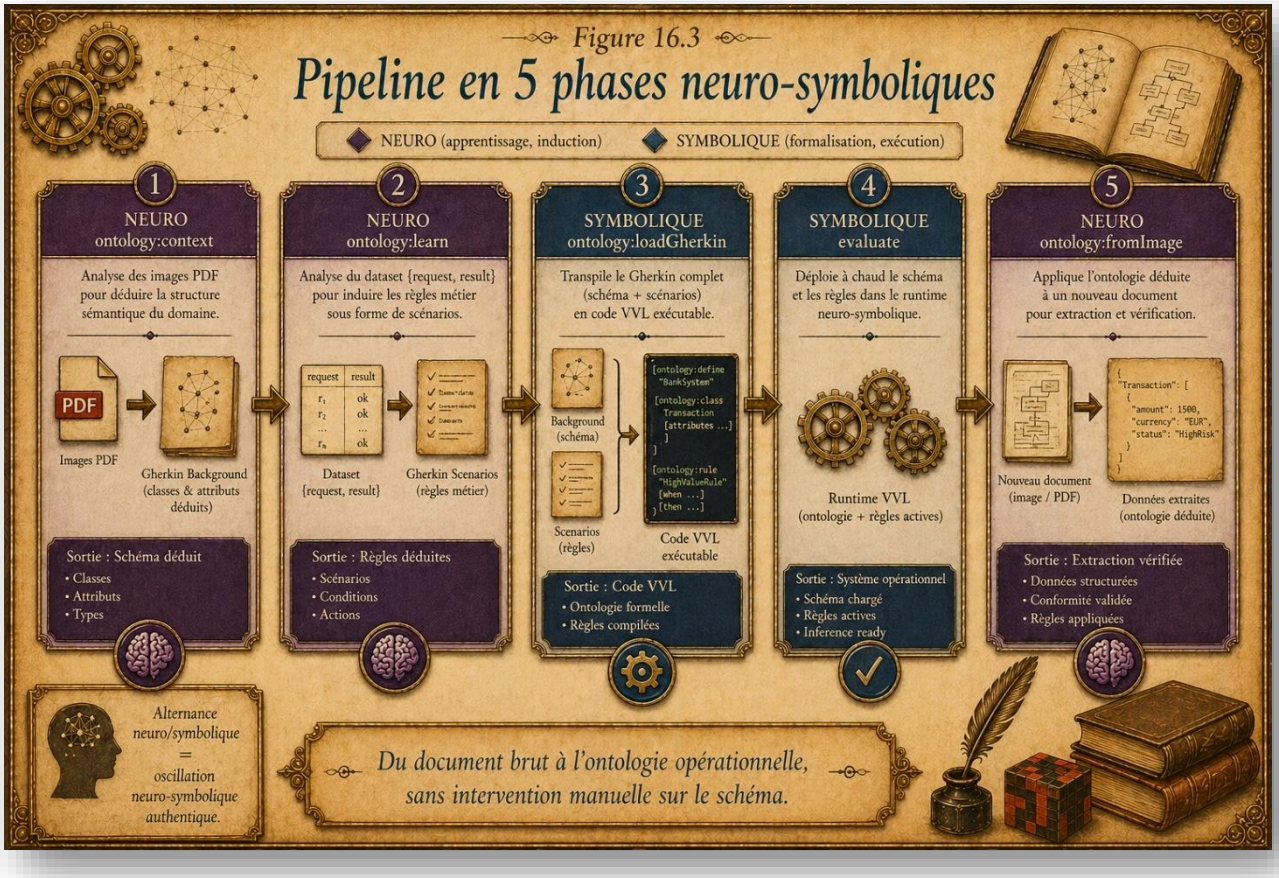


Figure 16.5 — *L'oscillation neuro-symbolique en 5 phases : context → learn → load → deploy → verify*

Auto-apprentissage continu

Une variante avancée du pipeline ferme la boucle pour un **auto-apprentissage continu**. Plutôt que de figer le schéma après le premier déploiement, on rejoue périodiquement les phases d'induction sur les nouveaux documents traités. Le système s'**affine au fil du temps** : il découvre de nouveaux types de devis, de nouvelles règles métier émergentes, des cas particuliers que le dataset initial ne couvrait pas. Le développeur n'écrit plus de règles : il alimente le dataset avec les corrections faites manuellement, et le système intègre ces corrections au prochain cycle d'apprentissage.

```
# Schema de base minimal au demarrage
$BASE = [ontology:define "BlindSystem"
    [ontology:class RollerBlindOrder
        [attributes
            [raw_text String]
            [width    Integer]
            [height   Integer]
            [options  List [of String]]
        ]
    ]
]

# Boucle d'auto-apprentissage : ingestion continue
$dataset = []
[for $file $new_files [do
    $text  = [pdfToText $file]
```

```
    $order = [ontology:fromText $BASE "RollerBlindOrder" $text]
    [$dataset.append $order]
]]

# Le LLM detecte une regularite : "Moteur Somfy" apparait quand
# le texte contient "electrique" OU largeur > 3000
$gherkin_rule = [ontology:learn $dataset "Moteur Somfy"]
# => Scenario: Auto Learned Motor Rule
#      Given a 'RollerBlindOrder' context
#      When 'raw_text' contains 'electrique'
#      Or 'width' is greater than 3000
#      Then add item 'Moteur Somfy' to 'options'

# Deploiement a chaud de la nouvelle regle
$vvl_logic = [ontology:loadGherkin $gherkin_rule]
[evaluate $vvl_logic]

# A partir de maintenant, les nouveaux devis sont automatiquement enrichis
```

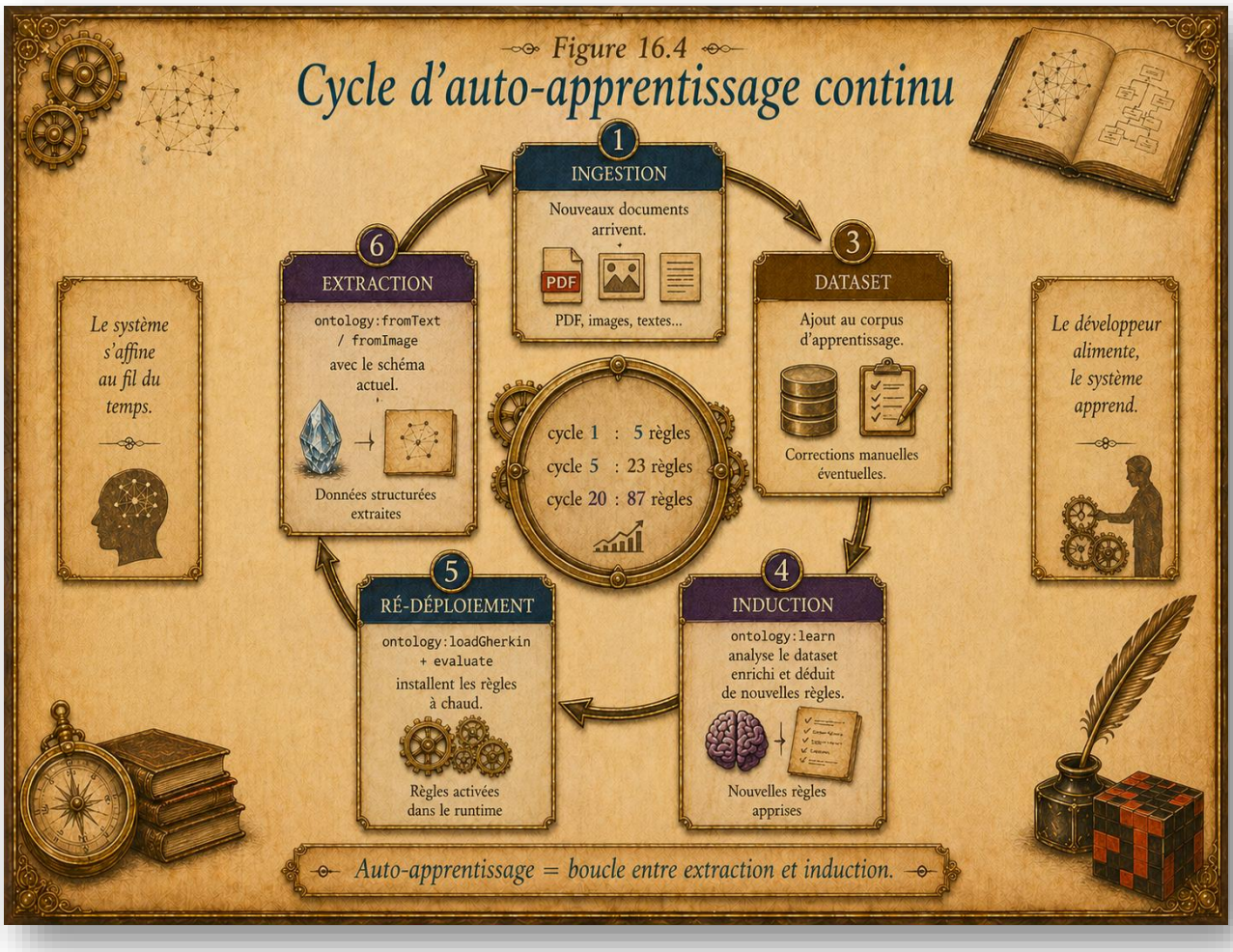


Figure 16.6 — *Le cycle d'auto-apprentissage : un système qui s'enrichit en continu*

16.7. Synthèse : un changement de paradigme

Avec ce chapitre, l'oscillation neuro-symbolique de VVL trouve sa forme la plus aboutie. Le tableau ci-dessous résume le changement de paradigme apporté par l'ontologie dynamique :

Aspect	Paradigme classique (ch. 12)	Paradigme dynamique (ch. 16)
Origine du schéma	Écrit manuellement par le développeur	Déduit par ontology:context
Origine des règles	Écrites manuellement avec ontology:rule	Induites par ontology:learn
Format pivot	— (direct en VVL)	Gherkin (lisible par humains)
Cycle de vie	Définition fixe au déploiement	Apprentissage continu en production
Rôle du développeur	Architecte de la structure	Curateur d'exemples
Adaptation au domaine	Refactoring du code	Enrichissement du dataset

Figure 16.7 — *Le changement de paradigme : du spécifié au démontré*

Ce paradigme dynamique n'est pas un simple gadget — il transforme la manière même dont on construit les systèmes d'information. Là où l'approche classique exige une **conception préalable** complète, l'approche dynamique permet un **démarrage progressif** où le système s'enrichit au fil de l'usage. C'est particulièrement précieux pour les domaines à forte variabilité (juridique, médical, BTP, immobilier) où aucun schéma fixe n'est jamais entièrement satisfaisant, et où les exceptions deviennent vite la règle.

Le **chapitre 17** approfondit l'aspect inductif du paradigme — `ontology:learn` — en détaillant les stratégies d'apprentissage, les heuristiques d'induction, et les patterns avancés (induction ciblée, validation croisée, consolidation de règles). Il fermera la Partie V et préparera la Partie VI consacrée aux pipelines complets en pratique : applications médicales, fraude, devis BTP, qui combineront tout l'arsenal présenté jusqu'ici.

17. Apprentissage inductif de règles

Mots-clés : induction, ontology:learn, dataset, target_feature, validation croisée, consolidation, règles induites, conflits, apprentissage continu, oscillation neuro-symbolique

Le chapitre 16 a introduit `ontology:learn` comme la deuxième moitié de l'ontologie dynamique : si `ontology:context` déduit le **schéma**, `ontology:learn` induit les **règles**. Ce chapitre 17 approfondit cette dimension inductive — la plus délicate des capacités neuro-symboliques de VVL —, en détaillant les stratégies d'apprentissage, les risques d'overfitting, les patterns de validation, et la coexistence avec les règles écrites manuellement.

L'induction est philosophiquement et techniquement plus délicate que la déduction. Là où Prolog *prouve* depuis des règles connues, l'induction *conjecture* des règles depuis des faits observés. Le résultat n'est jamais garanti : deux datasets différents peuvent produire deux jeux de règles différents — et incompatibles. Le LLM, en jouant le rôle d'inducteur, hérite à la fois de sa puissance de généralisation et de ses biais : il peut sur-généraliser à partir de peu d'exemples, ou rester trop proche des cas particuliers du dataset. La discipline du développeur est essentielle pour transformer ces conjectures en règles métier fiables.

Le chapitre suit la séquence suivante : le défi épistémologique de l'induction (17.1), anatomie d'un dataset (17.2), induction libre (17.3), induction ciblée par `target_feature` (17.4), validation et consolidation (17.5), gestion des conflits avec les règles manuelles (17.6), pattern d'induction continue (17.7), synthèse de la Partie V (17.8).

17.1. Le défi épistémologique de l'induction

La distinction entre **déduction** et **induction** est l'une des plus anciennes et des plus fondamentales de la logique. La **déduction** part de règles générales et en tire des conclusions particulières : si « tous les hommes sont mortels » et « Socrate est un homme », alors « Socrate est mortel ». Le résultat est *garanti* — si les prémisses sont vraies, la conclusion l'est aussi. C'est exactement ce que fait Prolog au chapitre 11 : il prouve une conclusion à partir de faits et de règles.

L'**induction** opère dans le sens inverse : elle part de cas particuliers observés et tente d'en tirer une règle générale. Si « ce devis de toiture coûte 35 000 € », « ce devis de réparation coûte 8 500 € », « ce devis d'isolation coûte 12 000 € », alors *peut-être* la règle est-elle « les toitures coûtent plus cher que les réparations ». Le résultat est *conjectural* — la règle peut être réfutée par le prochain cas observé. C'est ce que fait `ontology:learn` : il regarde des couples (entrée → sortie) et propose des règles candidates.

Cette différence n'est pas anodine : elle conditionne tout le travail du développeur. Avec la déduction, on a confiance dans les conclusions tant qu'on a confiance dans les prémisses ; avec l'induction, on ne fait jamais entièrement confiance — on **valide** sur de nouveaux cas, on **consolide** par recoupement, on **désactive** quand les exceptions s'accumulent. La force de VVL est de mettre les deux capacités dans le même runtime, de telle sorte qu'on peut induire une règle, la déduire ensuite, et l'auditer en continu.

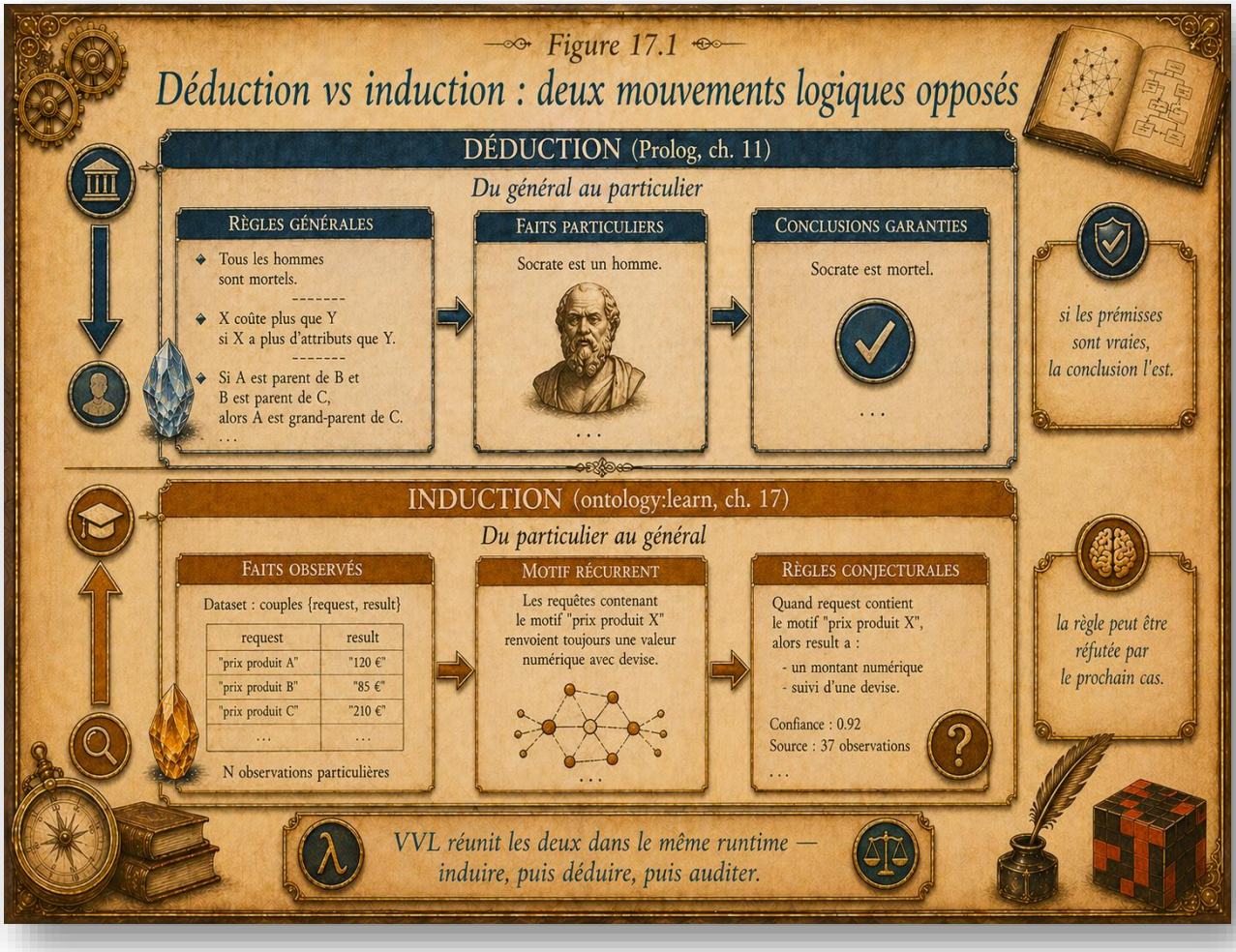


Figure 17.1 — *Deux mouvements logiques opposés : la déduction prouve, l'induction conjecture*

Une autre manière de dire la même chose : Prolog a un **pouvoir de preuve** ; le LLM inductif a un **pouvoir de proposition**. C'est exactement la différence entre un mathématicien qui démontre un théorème et un physicien qui propose une loi à partir de mesures. Les deux sont nécessaires en science ; les deux sont nécessaires en programmation neuro-symbolique.

17.2. Anatomie d'un dataset d'apprentissage

La qualité de l'induction dépend directement de la qualité du dataset. Un dataset trop petit ou trop homogène produit des règles fragiles ; un dataset diversifié et bien conçu produit des règles utiles. Cette section présente la structure canonique attendue par `ontology:learn` et les heuristiques pratiques pour la construire.

Format canonique : couples request/result

Le dataset est une liste d'objets, chacun contenant deux champs : **`request`** — la requête métier ou le contexte d'entrée, sous forme de chaîne libre ou d'objet structuré ; et **`result`** — le résultat attendu, sous forme d'objet ontologique ou de JSON. Le LLM compare les pairs `(request, result)` pour identifier ce qui change dans la sortie en fonction de l'entrée — et formule des règles candidates.

```
# Dataset minimaliste : 3 paires (requete client, devis attendu)
$dataset = [
    <{"request": "Refaire etancheite terrasse 20m2",
      "result":  {"type": "TERRASSE",
                  "lignes": [
                      {"designation": "DEPOSE EVACUATION",  "qte": 20, "pu": 100},
                      {"designation": "PARE VAPEUR BITUME", "qte": 20, "pu": 50},
                      {"designation": "BICOUCHE ETANCHEITE","qte": 20, "pu": 100}
                  ],
                  "tva": 10}}>,

    <{"request": "Fuite : reparer avec systeme liquide",
      "result":  {"type": "REPARATION",
                  "lignes": [
                      {"designation": "NETTOYAGE PREPARATION", "qte": 2, "pu": 200},
                      {"designation": "PRIMAIRE ACCROCHE",     "qte": 2, "pu": 200}
                  ],
                  "tva": 10}}>,

    <{"request": "Cool roof sur batiment industriel",
      "result":  {"type": "COOLROOF",
                  "lignes": [
                      {"designation": "PEINTURE REFLECTIVE", "qte": 100, "pu": 30}
                  ],
                  "tva": 20}}>
]

# Variante : objets ontologiques deja extraits
$dataset_alt = [
    <{"request": "Toiture en tuiles", "result": $devis_001}>,
    <{"request": "Etancheite urgente", "result": $devis_002}>,
    <{"request": "Isolation combles",  "result": $devis_003}>
]
```

Trois axes de qualité

Un bon dataset équilibre trois axes. **L'axe de variation** : les requêtes doivent couvrir les variations attendues du domaine (différents types de chantier, différents matériaux, différentes échelles) ; sans variation, le LLM produit des règles trop spécifiques. **L'axe de complexité** : il faut au moins quelques cas simples (« réparation ponctuelle ») et quelques cas complexes (« pose intégrale neuve avec garantie décennale »). **L'axe de longueur** : les requêtes peuvent être courtes ou longues, mais doivent contenir tous les termes décisifs — un dataset où les requêtes sont systématiquement coupées avant un mot-clé crucial induit des règles partielles.

La taille recommandée varie avec la complexité du domaine. Pour des cas binaires simples (deux types de devis, trois règles métier), trois à cinq paires suffisent. Pour des domaines réels (dix types de chantier, quinze règles imbriquées), on monte à quinze à trente paires. Au-delà, on rencontre une limite plus contraignante : la **taille maximale du contexte LLM** — un dataset de cent pages saturera le prompt et fera échouer l'induction. Le pattern recommandé pour les très grands datasets est de **stratifier** : faire plusieurs appels d'induction sur des sous-ensembles thématiques, puis consolider.



Figure 17.2 — *Les trois axes de qualité d'un dataset d'apprentissage*

17.3. Induction libre : ontology:learn sans cible

Dans sa forme la plus simple, `ontology:learn $dataset` ne reçoit que le dataset, sans indication de ce sur quoi se concentrer. Le LLM cherche alors **toutes les régularités** qu'il peut identifier — quels mots-clés de la requête déterminent le `type`, quelles tailles déclenchent une option particulière, quels types de chantier ont quelle TVA. Le résultat est un Gherkin contenant typiquement plusieurs Scenarios.

```
# Induction non ciblee : le LLM cherche toutes les regularites
$gherkin_rules = [ontology:learn $dataset]
[log $gherkin_rules]
# => @priority: 100
#    Scenario: Detect Terrasse Type
#      Given a 'Devis' context
#      When 'request' contains 'terrasse'
#      Then set 'type' to 'TERRASSE'
#      And add item 'DEPOSE EVACUATION' to 'lignes'
#      And add item 'BICOUCHE ETANCHEITE' to 'lignes'
#
#    @priority: 100
#    Scenario: Detect Reparation Type
#      Given a 'Devis' context
#      When 'request' contains 'fuite'
#      Or 'request' contains 'reparation'
#      Then set 'type' to 'REPARATION'
#
#    @priority: 90
#    Scenario: TVA pour cool roof
#      Given a 'Devis' context
#      When 'type' equals 'COOLROOF'
#      Then set 'tva' to 20
#
#    @priority: 90
#    Scenario: TVA standard
#      Given a 'Devis' context
#      When 'type' is not 'COOLROOF'
#      Then set 'tva' to 10
```

L'induction libre a deux propriétés à connaître. **Avantage** : elle est exploratoire — elle peut découvrir des régularités auxquelles le développeur n'aurait pas pensé spontanément. C'est utile en début de projet, quand on ne sait pas encore quelles règles sont importantes. **Inconvénient** : elle peut produire des règles redondantes, faiblement justifiées (basées sur un seul exemple), ou trompeuses (corrélations apparentes qui ne tiennent pas sur de nouveaux cas).

En pratique, l'induction libre est rarement déployée telle quelle en production. On l'utilise comme **point de départ** : on lance l'induction libre, on lit les Scenarios produits, on en garde certains, on en jette d'autres, on en raffine certains à la main. Le développeur reste l'arbitre final.

17.4. Induction ciblée : ontology:learn avec target_feature

Pour des inductions plus précises, `ontology:learn $dataset $target_feature` indique au LLM **l'attribut à expliquer**. Le LLM se concentre alors sur les règles qui déterminent cet attribut, ignorant les autres régularités. Cette focalisation produit des règles plus pertinentes, plus justifiées, et plus faciles à valider — au prix de devoir itérer sur les attributs critiques un par un.

```
# Induction ciblee : on demande les regles qui determinent "type"
$gherkin_type = [ontology:learn $dataset "type"]
# Le LLM se concentre sur les correspondances request -> type
# => Scenario: Detect Terrasse / Reparation / CoolRoof
#    (sans regles parasites sur tva ou autres attributs)

# Induction ciblee sur "tva"
$gherkin_tva = [ontology:learn $dataset "tva"]
# Le LLM se concentre sur ce qui determine la TVA
# => Scenario: TVA pour COOLROOF (20%)
#    Scenario: TVA standard (10%)

# Induction ciblee sur une option specifique
$gherkin_motor = [ontology:learn $dataset "Moteur Somfy"]
# Le LLM cherche : qu'est-ce qui declenche l'ajout d'un Moteur Somfy ?
# => Scenario: Auto Learned Motor Rule
#      Given a 'Order' context
#      When 'raw_text' contains 'electrique'
#      Or 'width' is greater than 3000
#      Then add item 'Moteur Somfy' to 'options'
```

L'induction ciblée est le mode **recommandé pour la production**. La discipline associée est claire : le développeur identifie les attributs critiques de son domaine (ceux qui ont un impact métier important), et lance une induction par attribut. Chaque ensemble de Scenarios produit est ensuite **validé** sur un jeu de test indépendant avant déploiement. Ce processus est plus laborieux que l'induction libre, mais il produit des règles justifiables une à une — ce qui est essentiel quand les règles ont des conséquences (montant de devis, décision de crédit, classification médicale).

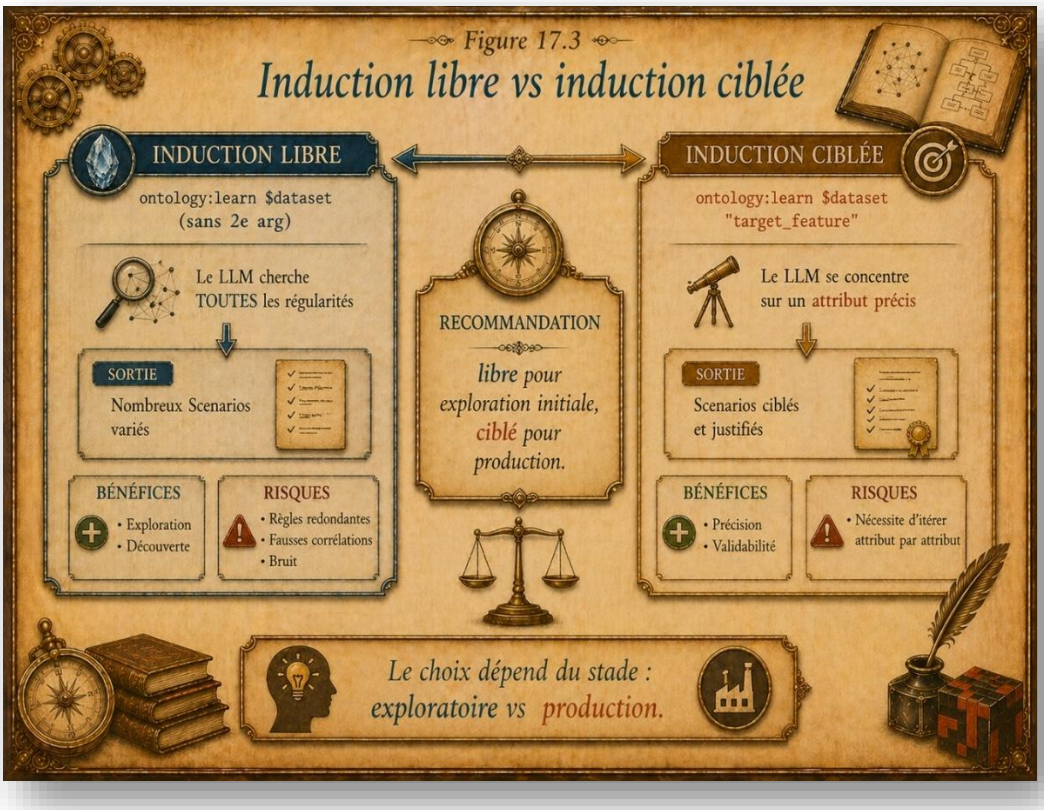


Figure 17.3 — *Deux modes d'induction pour deux stades du développement*

17.5. Validation et consolidation des règles induites

Une règle induite n'est valide que tant qu'elle n'a pas été réfutée. La pratique professionnelle de l'induction repose donc sur trois disciplines complémentaires : la **validation croisée** (tester sur un jeu indépendant), la **détection des règles fragiles** (identifier celles qui se justifient mal), et la **consolidation** (fusionner ou éliminer les règles redondantes).

Validation croisée

Le pattern recommandé est de scinder le dataset disponible en deux : un **jeu d'entraînement** (typiquement 70 %) sur lequel on lance `ontology:learn`, et un **jeu de validation** (typiquement 30 %) sur lequel on vérifie que les règles induites produisent les résultats attendus. Si les règles échouent sur plus de 20 % des cas de validation, elles sont à raffiner ou rejeter.

```
# Scinder le dataset en train (70%) et validation (30%)
$total      = [length $dataset]
$cut        = [INT ($total * 0.7)]
$train_set  = [slice $dataset 0 $cut]
$valid_set  = [slice $dataset $cut $total]

# Induire sur le jeu d'entrainement
$gherkin    = [ontology:learn $train_set "type"]
$vvl_code   = [ontology:loadGherkin $gherkin]
[evaluate $vvl_code]

# Valider sur le jeu de validation
$echecs = 0
[for $exemple $valid_set [do
    $instance = [ontology:fromText $Schema "Devis" $exemple.request]
    [ontology:commit $instance]
    [ontology:infer $Schema]

    [if ($instance.type != $exemple.result.type) [do
        $echecs = ($echecs + 1)
        [log "Echec : " $exemple.request " (attendu " $exemple.result.type
             ", obtenu " $instance.type ")"]
    ]]
]]

$taux_echec = ($echecs / [length $valid_set])
[log "Taux d'echec : " ($taux_echec * 100) "%"]
[if ($taux_echec > 0.20)
    [log "ATTENTION : regles a raffiner avant deploiement"]
    [log "OK : regles deployables"]
]
```

Détection et consolidation

Plusieurs anti-patterns sont à repérer dans les règles induites. **Le sur-ajustement** : une règle qui mentionne un terme ultra-spécifique du dataset (par exemple `When 'request' contains 'Schmidt-Briard 2025'`) est probablement un artefact d'un seul exemple. **La redondance** : deux règles qui aboutissent à la même conclusion par des chemins différents peuvent être fusionnées. **La contradiction** : deux règles qui s'opposent sur un même cas signalent une incohérence du dataset, souvent due à un mauvais étiquetage.

La consolidation manuelle reste donc une étape importante. Le développeur lit les Scenarios produits, élimine ceux qui sont fragiles, fusionne les redondances, et corrige le dataset si des contradictions

apparaissent. Ce travail n'est pas du gaspillage — c'est la version ontologique du *code review* : chaque règle déployée a été lue et validée.

17.6. Conflits entre règles induites et règles écrites

En production, les règles induites cohabitent généralement avec des **règles écrites manuellement** par les experts métier. Les premières émergent du dataset ; les secondes encodent la connaissance experte que le dataset ne contient pas (réglementations, contraintes contractuelles, politique d'entreprise). Ces deux types de règles peuvent **se contredire**, et le système doit savoir lesquelles l'emportent.

La convention recommandée par VVL utilise le mécanisme de **priorité** déjà présenté au chapitre 12. Les règles écrites manuellement reçoivent une priorité **basse** (typiquement 50), ce qui les fait s'exécuter **après** les règles induites. Comme les règles précédentes peuvent avoir déjà fixé certains champs, les règles manuelles ont l'occasion de **corriger ou de surcharger** les conclusions du système inductif. C'est l'inverse de l'intuition naïve (« la connaissance experte vaut plus, donc priorité haute ») : en fait, l'expert intervient *en dernier* pour valider ou rectifier ce que l'induction a produit.

```
# 1. Regles induites par le systeme (priorite haute, executees en premier)
$gherkin_induced = [ontology:learn $dataset "type"]
$vvl_induced     = [ontology:loadGherkin $gherkin_induced]
[evaluate $vvl_induced]
# Genere par exemple :
#   @priority: 100
#   Scenario: Detect Terrasse
#     Given a 'Devis' context
#     When 'request' contains 'terrasse'
#     Then set 'type' to 'TERRASSE'

# 2. Regle metier ecrite par l'expert (priorite basse, derniere a s'executer)
[ontology:rule "ExceptionVIP" $Schema
    [target "Devis"]
    [priority 30]                              # plus basse que 100
    [when (($this.client_status == "VIP") && ($this.type == "REPARATION"))]
    [then
        [ontology:set $this "type" "INTERVENTION_PRIORITAIRE"]
        [log "Surcharge VIP : reparation reclassee"]
    ]
]

# 3. A l'inference :
#    - les regles induites s'executent d'abord (priorite 100)
#      -> $devis.type = "REPARATION"
#    - la regle expert s'execute ensuite (priorite 30)
#      -> si client VIP, surcharge en "INTERVENTION_PRIORITAIRE"
[ontology:infer-fixpoint $Schema 3]
```

Cette convention a un effet pédagogique précieux : le développeur peut **lire l'historique des transformations** d'une instance en consultant la séquence des règles déclenchées. La règle induite a posé une première hypothèse ; la règle expert l'a éventuellement corrigée. La traçabilité est meilleure que dans un système monolithique où une seule règle ferait tout.

Une bonne pratique complémentaire est de **tagger les règles induites** dans leur nom (par exemple `Auto-Learned-DetectTerrasse`), pour les distinguer visuellement des règles écrites à la main lors d'audit. Le développeur peut alors filtrer en un coup d'œil quelles règles relèvent de l'induction et

lesquelles relèvent de l'expertise — un signal d'autant plus précieux que les premières peuvent être régénérées, tandis que les secondes sont gravées.

17.7. Pattern d'induction continue

Le dernier pattern important est l'**induction continue** : au lieu de faire l'induction une fois pour toutes au démarrage, on la rejoue périodiquement à mesure que de nouveaux documents arrivent. Ce pattern, déjà esquissé au chapitre 16, mérite ici un développement explicite.

```
# Boucle d'induction continue : execution periodique (cron, scheduler)
[keyword "ReinduceRulesPeriodically" `[do
    # 1. Recuperer les nouveaux documents traites depuis la derniere induction
    $new_records = [obj-find Devis processed_after $LAST_INDUCTION]
    [if ([length $new_records] < 10) [do
        [log "Trop peu de nouveaux documents (" [length $new_records] "), skip"]
        [return]
    ]]

    # 2. Construire le dataset enrichi
    $dataset = [map [lambda [d]
        <{"request": $__d.raw_text, "result": $__d.json}>
    ] $new_records]

    # 3. Induire sur les attributs critiques (ciblee, par boucle)
    $attrs_critiques = ["type" "tva" "options"]
    $gherkin_complet = ""
    [for $attr $attrs_critiques [do
        $g = [ontology:learn $dataset $__attr]
        $gherkin_complet = ($gherkin_complet + "\n" + $g)
    ]]

    # 4. Valider sur jeu de test independant
    $taux_echec = [validate-rules $gherkin_complet $valid_set]
    [if ($taux_echec > 0.20) [do
        [log "Re-induction rejetee : taux d'echec " ($taux_echec * 100) "%"]
        [return]
    ]]

    # 5. Versioner et deployer a chaud
    [file-write (concat "rules/v" $VERSION ".feature") $gherkin_complet]
    $code = [ontology:loadGherkin $gherkin_complet]
    [evaluate $code]
    [log "Re-induction deployee : version " $VERSION]

    $LAST_INDUCTION = [timestamp]
    $VERSION = ($VERSION + 1)
]]

# Programmation : execute une fois par jour
# (ou via scheduler externe : cron, k8s CronJob, ...)
```

Ce pattern transforme le système en un **organisme apprenant**. Chaque jour, semaine ou mois (selon le rythme métier), il observe ce qui s'est passé, propose de nouvelles règles, les valide, et les intègre si elles passent le test. Le développeur n'écrit plus de règles à la main — il maintient le pipeline d'apprentissage et arbitre les rares cas où l'induction produit des résultats contre-intuitifs.

La **versioning** des Gherkin produits est essentielle pour pouvoir revenir en arrière en cas de régression. Conserver `rules/v1.feature`, `rules/v2.feature`, etc., permet de comparer comment les

règles évoluent dans le temps, et constitue un journal d'apprentissage du système qu'aucun système classique ne peut produire.

17.8. Synthèse de la Partie V : les quatre outils neuronaux

Avec ce chapitre, la Partie V est complète. Le tableau suivant récapitule les **quatre grands outils** que le LLM apporte à VVL pour alimenter et enrichir la couche symbolique présentée dans la Partie IV :

Outil	Chapitre	Rôle	Sortie
RAG	14	Recherche dans un corpus indexé	Réponse NL ou JSON synthétisé
Extraction structurée	15	Conversion d'un document en données	JSON ou instance ontologique
Gherkin / context	16	Déduction du schéma depuis exemples	Gherkin Background
Apprentissage / learn	17	Induction de règles depuis dataset	Gherkin Scenarios

Figure 17.4 — *Les quatre outils neuronaux qui alimentent la couche symbolique*

Chacun de ces outils apporte une capacité distincte au système. **Le RAG** apporte la **mémoire à long terme** : un corpus persistant interrogeable. **L'extraction** apporte la **conversion ponctuelle** : du non structuré au structuré pour un document à la fois. **Le Gherkin/context** apporte la **structure** : la capacité de découvrir le schéma au lieu de l'imposer. **L'apprentissage/learn** apporte le **comportement** : la capacité de découvrir les règles au lieu de les écrire. Réunis, ces quatre outils forment l'écosystème neuronal de VVL — un ensemble cohérent qui transforme un langage avec ontologie en un véritable système d'apprentissage continu.



Figure 17.5 — *La Partie V complète : les quatre outils neuronaux d'alimentation symbolique*

La **Partie VI** qui s'ouvre au chapitre 18 montre tout cet arsenal en action sur des **pipelines complets de production**. Quatre cas d'usage seront traités en profondeur : l'oscillation neuro-symbolique formalisée (chapitre 18), le diagnostic médical différentiel (chapitre 19), un système de devis BTP de bout en bout (chapitre 20), et la détection de fraude transactionnelle (chapitre 21). Chacun de ces chapitres combinera les couches symboliques de la Partie IV et les outils neuronaux de la Partie V pour résoudre un problème métier concret. Les chapitres précédents ont posé les briques ; les chapitres suivants montrent les édifices.

Partie VI — Pipelines de production

Chapitres 18 à 21 — l'oscillation formalisée, puis trois cas industriels

18. L'oscillation neuro-symbolique formalisée

Mots-clés : oscillation, pipeline, alternance, cycle, neuro, symbolique, fidélité, coût, latence, diagnostic différentiel, pattern architectural

La Partie VI s'ouvre sur un changement de registre. Les Parties III, IV et V ont présenté les **briques** de la programmation neuro-symbolique : le LLM comme co-processeur (Partie III), la couche symbolique avec Prolog, ontologie et opérateurs Zalta (Partie IV), et les outils neuronaux d'alimentation — RAG, extraction, Gherkin, induction (Partie V). Cette Partie VI prend du recul et montre les **édifices** : comment combiner ces briques en pipelines complets de production, et comment formaliser l'alternance entre couches qui définit l'authenticité neuro-symbolique de VVL.

Ce premier chapitre de la Partie VI est consacré au **pattern fondamental** qui sous-tend tous les pipelines de production : l'**oscillation neuro-symbolique**. Au fil des chapitres précédents, ce pattern est apparu à plusieurs reprises sans être nommé : un schéma défini en symbolique reçoit des extractions neuronales, qui sont validées par des règles symboliques, et présentées en langage naturel par le LLM. Cette alternance entre les deux couches n'est pas un design ad-hoc : c'est le **fonctionnement intrinsèque** d'un système neuro-symbolique authentique, et elle mérite une formalisation explicite avant d'aborder les cas d'usage des chapitres suivants.

Le chapitre suit la séquence : motivation et formalisation (18.1), anatomie d'une oscillation (18.2), les quatre patterns canoniques (18.3), étude de cas du diagnostic différentiel (18.4), rôle de l'objet comme pont (18.5), métriques d'oscillation (18.6), synthèse comme oscillation programmable (18.7).

18.1. Pourquoi formaliser l'oscillation

L'expression « système neuro-symbolique » est devenue à la mode dans la littérature récente sur l'intelligence artificielle, mais elle recouvre des réalités très différentes selon les implémentations. Dans certains systèmes, « neuro-symbolique » signifie simplement « le LLM appelle une fonction Python qui implémente une règle » — une greffe minimale qui ne mérite pas le nom. Dans d'autres, comme les systèmes ILP (*Inductive Logic Programming*) classiques, la combinaison est forte mais limitée à un seul mode d'interaction (l'induction de règles). VVL revendique une troisième forme, plus exigeante : un langage où le neuronal et le symbolique **coexistent en première classe**, **partagent les mêmes objets**, et où le programme **alterne naturellement** entre les deux couches selon les besoins.

Cette alternance — l'oscillation neuro-symbolique — est ce qui transforme un système probabiliste en un traitement fiable. Le LLM seul produit des sorties plausibles mais non vérifiables ; le système symbolique seul est rigoureux mais incapable de comprendre du langage naturel. Leur **alternance disciplinée** combine les forces de chacun : le LLM extrait et propose, le symbolique vérifie et déduit ; le LLM enrichit et présente, le symbolique contraint et trace. Quand le pattern est respecté, on obtient des systèmes qui sont à la fois adaptables (comme les LLM) et fiables (comme les systèmes formels).

Cette section formalise le pattern en explicitant ses propriétés. Une oscillation neuro-symbolique authentique présente quatre caractéristiques : (1) **alternance régulière** entre les deux couches au cours d'un même pipeline ; (2) **partage d'un même tissu d'objets** qui circule à travers les couches sans transformation ; (3) **traçabilité** complète des transitions, permettant l'audit et le debugging ; (4) **complémentarité fonctionnelle** où chaque couche apporte ce que l'autre ne peut pas. Les pipelines qui ne satisfont pas ces propriétés sont des architectures d'intégration, pas des systèmes neuro-symboliques.

18.2. Anatomie d'une oscillation

La forme la plus simple d'oscillation neuro-symbolique comporte **quatre phases** alternant entre les deux couches. C'est le cycle de base que tout développeur VVL apprend à reconnaître et à composer. Chacune des phases utilise des commandes spécifiques que les chapitres précédents ont introduites en détail.

Phase	Couche	Opération	Commandes typiques
1. Définition	Symbolique	Schéma typé avec contraintes	ontology:define, ontology:class
2. Extraction	Neuro	NL → instances structurées	ontology:fromText, ontology:fromImage
3. Inférence	Symbolique	Règles métier → conclusions	ontology:rule, ontology:infer-fixpoint
4. Synthèse	Neuro	Faits déduits → réponse NL	ontology:askMagic, opérateurs §

Figure 18.1 — *Le cycle élémentaire à quatre phases : alternance régulière*

```
# === PHASE 1 (Symbolique) : Definition du schema ===
$HR = [ontology:define "HR_Recruitment"
    [ontology:class Candidat
        [attributes
            [nom        String  [required true]]
            [poste      String  [required true]]
            [experience Integer [default 0]]
            [niveau     String  [default "JUNIOR"]]
            [decision   String  [default "EN_ATTENTE"]]
        ]
    ]
]

# === PHASE 2 (Neuro) : Extraction depuis du texte brut ===
$cv = "Jean-Pierre Martin, ingenieur, 8 ans d'experience.
       Polytechnique. Python, ML, AWS."
$candidat = [ontology:fromText $HR "Candidat" $cv]
[ontology:commit $candidat]
# Resultat : instance avec nom="Jean-Pierre Martin", experience=8, ...

# === PHASE 3 (Symbolique) : Inference par regles metier ===
[ontology:rule "NiveauSenior" $HR
    [target "Candidat"]
    [priority 100]
    [when ($this.experience >= 7)]
    [then [ontology:set $this "niveau" "SENIOR"]]
]
[ontology:rule "DecisionAcceptee" $HR
    [target "Candidat"]
    [priority 50]
    [when (($this.niveau == "SENIOR") && ($this.experience >= 5))]
    [then [ontology:set $this "decision" "ACCEPTE"]]
]
[ontology:infer-fixpoint $HR 3]
# Resultat : niveau="SENIOR", decision="ACCEPTE"

```

```
# === PHASE 4 (Neuro) : Synthese en langage naturel ===
$rapport = ($candidat ? [§ "redige une synthese RH de 3 lignes
                           sur ce candidat avec recommandation finale"])
[log $rapport]
```

Cinq éléments méritent d'être soulignés dans ce cycle. **Premièrement, la phase 1 est faite une fois pour toutes** au démarrage de l'application, alors que les phases 2-4 sont rejouées pour chaque candidat. Le schéma joue le rôle de contrat stable. **Deuxièmement, la phase 3 est purement symbolique** : aucun appel LLM n'intervient ici, ce qui garantit la reproductibilité des décisions. Pour la même instance d'entrée, les mêmes règles produisent toujours le même résultat. **Troisièmement, la phase 4 reçoit l'instance enrichie par la phase 3**, pas l'instance brute : le LLM voit donc à la fois les faits extraits et les conclusions déduites. **Quatrièmement, l'objet `$candidat` est le même tout au long du cycle** — il a juste accumulé des champs : nom puis niveau puis decision. C'est ce point qui sera développé en 18.5. **Cinquièmement, le code est étonnamment compact** : trente lignes pour un système qui combine extraction, validation, inférence et présentation.

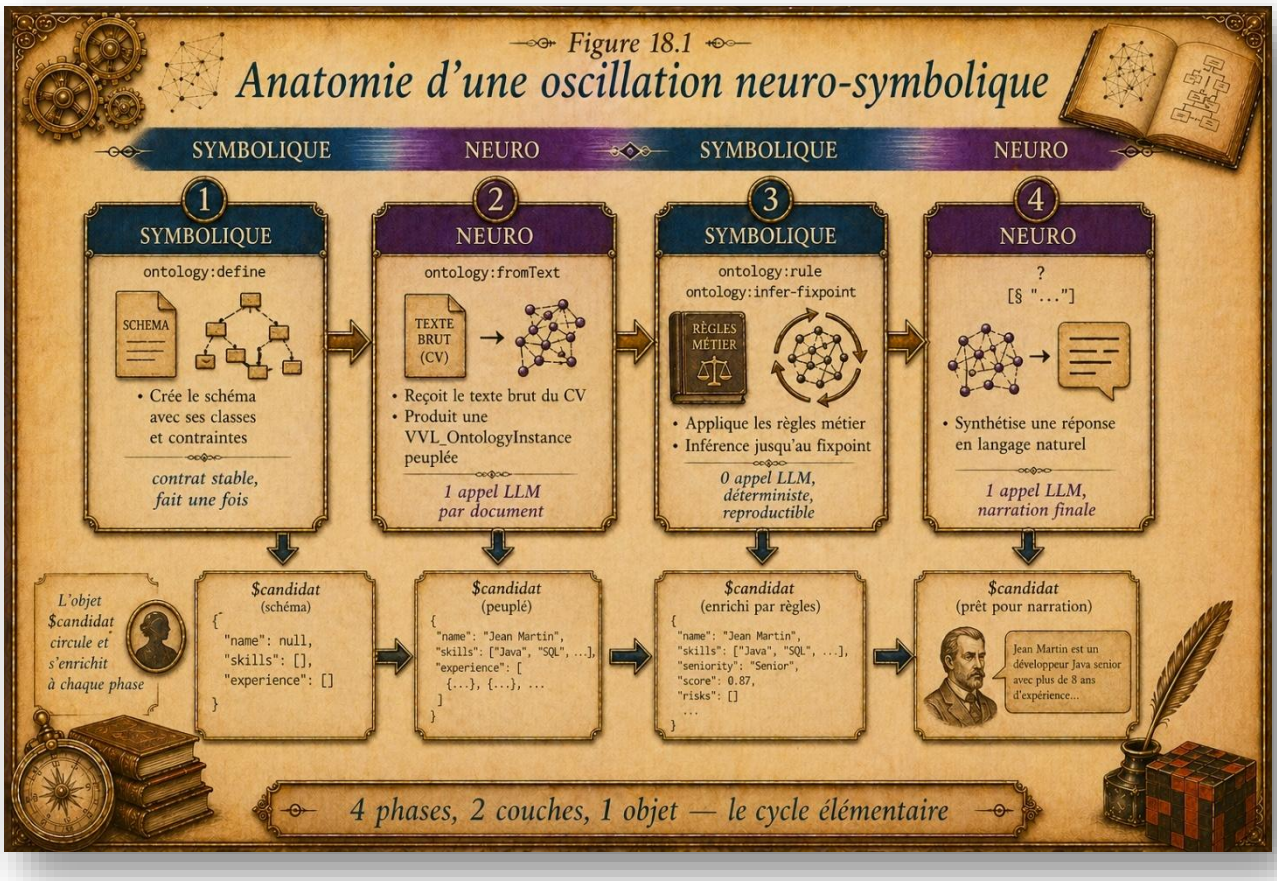


Figure 18.2 — *Le cycle élémentaire à quatre phases : alternance régulière, objet partagé*

18.3. Les quatre patterns canoniques d'oscillation

Le cycle de base à quatre phases présenté en 18.2 est une **forme canonique** mais non l'unique. La pratique dégage **quatre patterns récurrents** d'oscillation, chacun adapté à une famille de problèmes. Connaître ces patterns aide à choisir la bonne combinaison de commandes pour un nouveau cas d'usage et à structurer les pipelines de manière reconnaissable par d'autres développeurs.

Pattern 1 — Définir / Extraire / Inférer / Présenter

C'est le pattern le plus courant, illustré par le cas RH de la section précédente. On a un schéma stable, des documents qui arrivent en flot, et un besoin de classifier ou décider. Schéma défini une fois, puis pour chaque document : extraction LLM, inférence par règles, présentation NL. Cas d'usage typiques : tri de candidatures, classification de tickets, qualification de leads.

Pattern 2 — Découvrir / Compiler / Apprendre / Déployer

Pattern d'**apprentissage initial** présenté au chapitre 16. On part de documents bruts sans connaître leur structure ; on demande au LLM de déduire le schéma (`ontology:context`), on transpile (`loadGherkin`), on apprend les règles (`learn`), on déploie. Une fois le système installé, on bascule sur le Pattern 1 pour le traitement quotidien. Cas d'usage : démarrage d'un projet d'extraction sur un nouveau type de document, exploration d'un domaine peu modélisé.

Pattern 3 — Indexer / Chercher / Synthétiser

Pattern de **recherche dans corpus** présenté au chapitre 14 — c'est le pattern RAG dans sa forme pure. Indexation hors ligne d'un corpus documentaire, puis pour chaque question : recherche vectorielle (gratuite, déterministe), synthèse par le LLM. L'oscillation est plus courte (deux phases) mais reste neuro-symbolique au sens strict — les embeddings et la similarité cosinus relèvent du symbolique calculé. Cas d'usage : assistants documentaires, FAQ d'entreprise, recherche dans corpus juridique.

Pattern 4 — Mesurer / Hypothétiser / Tester / Itérer

Pattern de **diagnostic différentiel** qui sera détaillé en 18.4. On observe des symptômes ou des indicateurs (mesures), on génère des hypothèses (réification de concepts par `[@ ...]`), on les teste contre les règles symboliques (Prolog ou ontologie), et on itère en raffinant. C'est le pattern le plus ambitieux car il combine *toutes* les briques de VVL : réification, opérateurs Zalta, inférence, raisonnement Prolog. Cas d'usage : médecine, banque (fraude), industrie (panne), sécurité (intrusion).

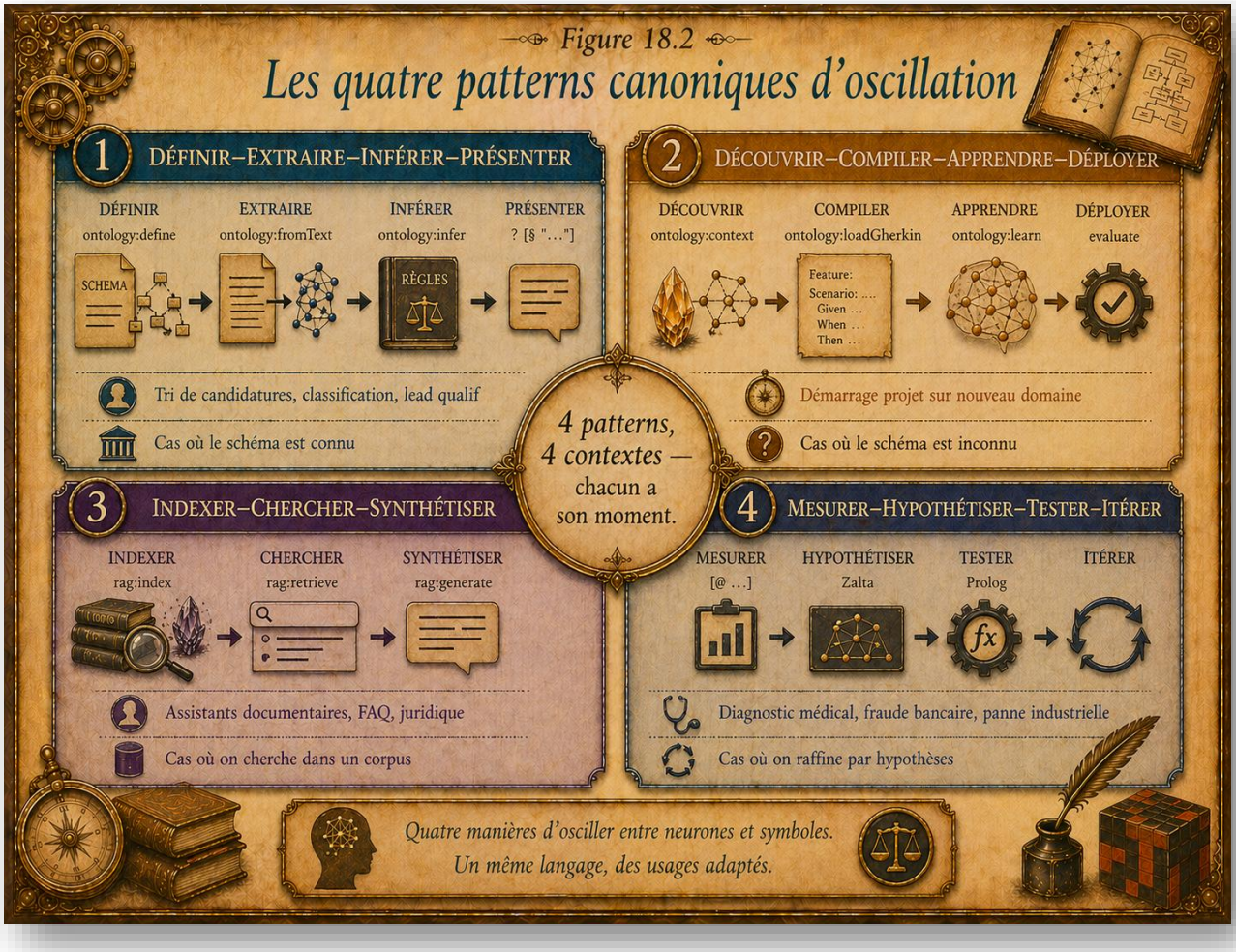


Figure 18.3 — *Les quatre patterns canoniques d'oscillation : un par grand contexte d'usage*

18.4. Étude de cas : le diagnostic différentiel

Le diagnostic médical différentiel illustre le **pattern 4** dans toute sa richesse. Le médecin reçoit une description en langage naturel des symptômes, doit générer des hypothèses diagnostiques, les tester contre les contre-indications connues, et présenter une recommandation au patient. Aucune des étapes n'est triviale : la première exige de comprendre du langage, la deuxième exige une connaissance médicale étendue, la troisième exige un raisonnement formel, la quatrième exige une présentation pédagogique.

Voici un exemple complet, sciemment simplifié pour la clarté pédagogique. La maladie sera déterminée à partir de symptômes décrits en langage naturel, avec validation par règles d'exclusion et présentation finale.

```
# === PHASE 1 (Symbolique) : Definition du schema medical ===
$MED = [ontology:define "DiagnosticSysteme"
    [ontology:class Patient
        [attributes
            [age       Integer [min 0] [max 130]]
            [sexe      String  [enum [M F]]]
            [symptomes List]
            [allergies List   [default []]]
        ]
    ]
    [ontology:class Diagnostic
        [attributes
            [maladie       String]
            [probabilite   Float  [min 0] [max 1]]
            [contre_indique Boolean [default false]]
        ]
    ]
]

# === PHASE 2 (Neuro) : Extraction du contexte patient ===
$texte_consultation = "Femme de 35 ans, fievre 39C depuis 3 jours,
                       toux seche persistante, fatigue extreme,
                       allergique a la penicilline."
$patient = [ontology:fromText $MED "Patient" $texte_consultation]
[ontology:commit $patient]
# => age=35, sexe=F, symptomes=[fievre, toux seche, fatigue],
#    allergies=[penicilline]

# === PHASE 3 (Neuro) : Generation d'hypotheses par reification ===
$hypotheses_brutes = ($patient ? [§ "donne 5 maladies potentielles
                                     compatibles avec ces symptomes,
                                     au format [\"nom1\", \"nom2\", ...]"])
$noms_maladies = [JSON $hypotheses_brutes]

# Reification de chaque maladie en concept manipulable (chapitre 9)
$candidats = [map [lambda [nom]
    [@ $__nom "lazy"]               # objet VVL_ConceptEntity
] [jsonToList $noms_maladies]]

# === PHASE 4 (Symbolique) : Test des hypotheses par regles ===
[ontology:rule "AntibiotiqueDeconseille" $MED
    [target "Diagnostic"]
    [priority 100]
    [when (($this.maladie ~~ "infection bacterienne") &&
           ("penicilline" in $patient.allergies))]
    [then [ontology:set $this "contre_indique" true]
          [log "Alerte : antibiotique contre-indique"]]
]

# Pour chaque hypothese, on cree une instance Diagnostic et on infere
$diagnostics = [map [lambda [c]
    $d = [new $MED "Diagnostic"]
    $d.maladie = $__c.name
    [obj-assert $d]
    [ontology:infer $MED]
    $d
] $candidats]

# Filtrer ceux qui ne sont pas contre-indiques
$retenus = [filter [lambda [d] (!$__d.contre_indique)] $diagnostics]

# === PHASE 5 (Neuro) : Synthese pour le medecin ===
$synthese = ($retenus ? [§ "redige une synthese diagnostique en 5 lignes
                            a destination d'un medecin generaliste,
                            classant les hypotheses par probabilite et
                            en mentionnant les precautions liees aux
                            allergies du patient"])
```

Ce pipeline illustre toute la profondeur de l'oscillation. **La phase 1** pose un schéma symbolique — c'est l'infrastructure de typage et de validation. **La phase 2** convertit le récit non structuré en objets typés — le LLM joue le rôle d'OCR sémantique du chapitre 15. **La phase 3** demande au LLM de générer des hypothèses, puis réifie chacune comme objet manipulable via `[@ ...]` — c'est le mécanisme du chapitre 9. **La phase 4** applique des règles symboliques pour éliminer les hypothèses contre-indiquées — c'est le système ontologique du chapitre 12. **La phase 5** synthétise pour le destinataire humain — retour au LLM pour la narration.

Le code total est de **soixante lignes** pour un système qui combinerait facilement plusieurs centaines de lignes en architecture classique : un parseur de symptômes, un moteur d'inférence médicale, une base de contre-indications, un générateur de rapport. Et chaque ligne est lisible par un développeur qui connaît VVL — pas de magie, pas de fichier de configuration externe, pas d'API à apprendre.

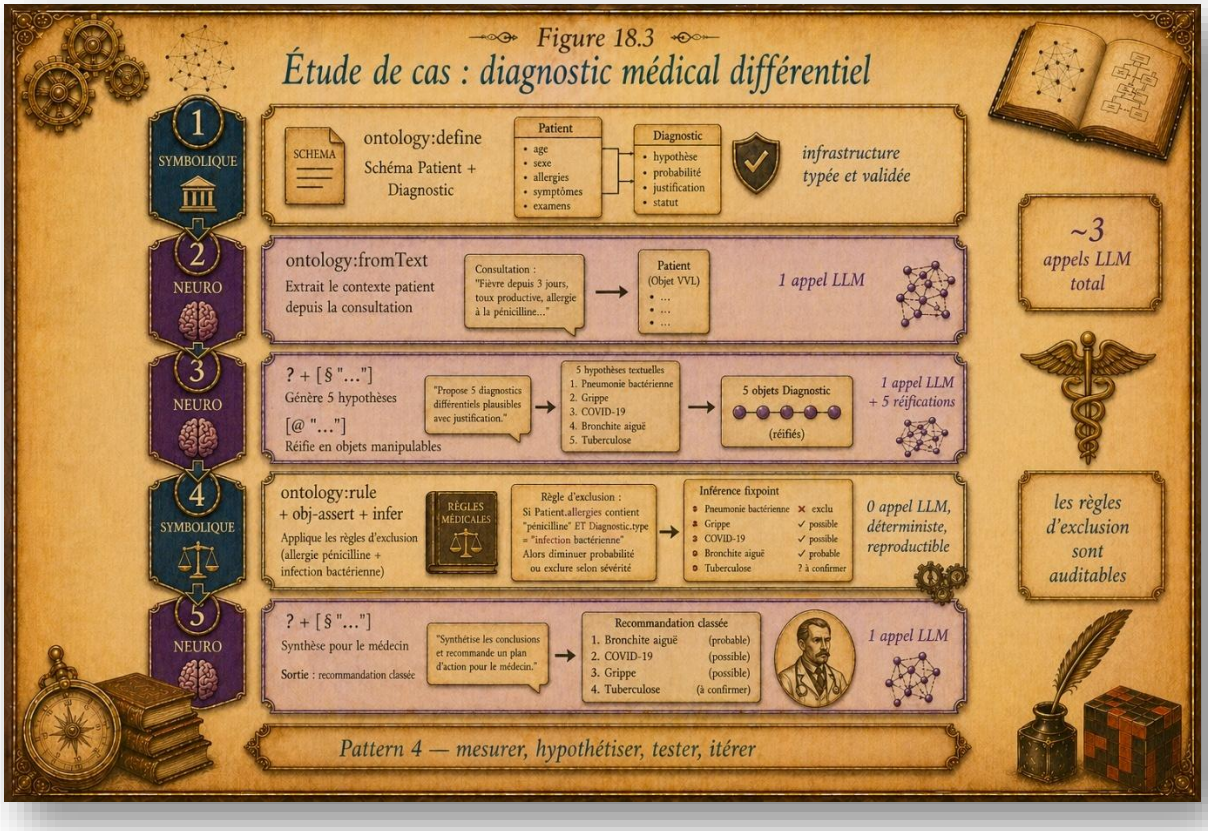


Figure 18.4 — *Le pattern de diagnostic différentiel : 5 phases pour un raisonnement médical traçable*

18.5. L'objet comme pont entre les couches

Au cœur de toute oscillation neuro-symbolique en VVL se trouve une propriété fondamentale : **l'objet est partagé entre les couches**. Le `$patient` extrait par le LLM en phase 2 est *exactement* le même objet que celui interrogé par les règles ontologiques en phase 4 et présenté au médecin en phase 5. Aucune sérialisation, aucune conversion, aucun ORM intermédiaire. Cette continuité est ce qui rend les pipelines lisibles et performants.

Cette propriété s'appuie sur les chapitres précédents : **le chapitre 6** a posé le système OO unifié avec `VVL_Instance` ; **le chapitre 9** a montré que les concepts réifiés `[@ "..."]` héritent de cette même classe ; **le chapitre 11** a montré que `obj-assert` projette n'importe quelle instance en faits Prolog interrogeables ; **le chapitre 12** a montré que les instances ontologiques en sont des spécialisations typées ; **le chapitre 13** a montré que les opérateurs Zalta s'appliquent uniformément à tous via le mixin. C'est un édifice cohérent, et c'est ce qui permet à un même objet de circuler à travers les couches.

Une comparaison clarifie l'enjeu. Dans une architecture classique, un patient extrait par GPT serait du JSON ; pour le passer à un moteur de règles Drools, il faudrait l'instancier en POJO Java ; pour le persister en SQL, le mapper avec un ORM ; pour le présenter, le re-sérialiser en JSON pour le frontend. Quatre représentations, quatre frontières, quatre sources de bugs. En VVL, **une seule représentation** : l'instance VVL, qui est nativement un objet OO, un fait Prolog (via `obj-assert`), une ligne SQLite (via `obj-save`), et un opérande des opérateurs Zalta. Le pipeline est plus court, plus rapide, et plus auditable.

18.6. Métriques d'oscillation : coût, latence, fidélité

Pour qu'une oscillation neuro-symbolique soit utile en production, elle doit être **mesurable** sur trois axes : le **coût** en appels LLM (qui détermine le coût API), la **latence** cumulée (qui détermine l'expérience utilisateur), et la **fidélité** (qui détermine la confiance dans les résultats). Le tableau suivant compare trois approches sur le cas RH de la section 18.2 : LLM monolithique (« demande tout au LLM »), RAG seul (« cherche dans la doc »), et pipeline neuro-symbolique complet.

Approche	Appels LLM/cas	Latence	Fidélité	Auditable
LLM monolithique	1	~2-5 s	Variable (~70-85 %)	Non — sortie boîte noire
RAG seul	2 (embed + completion)	~1-3 s	Bonne (~85-90 %)	Partiel — chunks visibles
Pipeline neuro-symbolique	2 (extraction + synthèse)	~2-4 s	Excellente (~95+ %)	Total — règles inspectables

Figure 18.5 — *Comparatif des trois approches sur le cas RH*

Trois remarques sur ces ordres de grandeur. **Premièrement, le pipeline neuro-symbolique n'est pas plus coûteux que le RAG** en nombre d'appels LLM (deux dans les deux cas) — la phase d'inférence symbolique est gratuite. C'est une surprise pour ceux qui pensent que le neuro-symbolique signifie « plus d'IA ». **Deuxièmement, la fidélité est nettement meilleure** car les règles métier garantissent des décisions reproductibles ; le LLM ne décide rien de critique, il extrait et présente seulement. **Troisièmement, l'auditabilité est totale** : on peut tracer pourquoi un candidat a été classé SENIOR (à cause de la règle `NiveauSenior` à priorité 100), ce qui est crucial dans les contextes régulés (RGPD, finance, médecine).

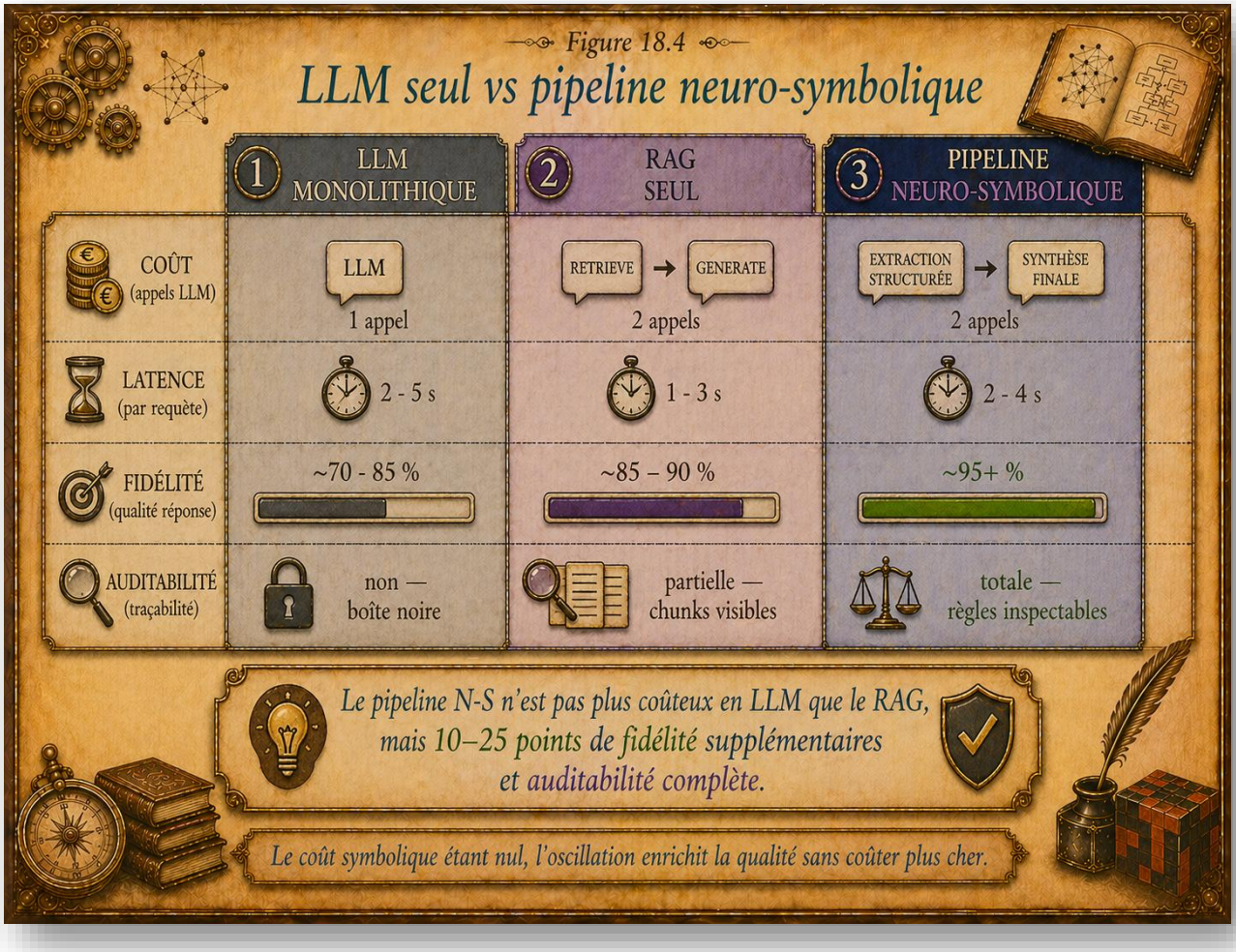


Figure 18.6 — *Trois approches sur trois axes : la valeur ajoutée du pipeline complet*

18.7. Synthèse : VVL comme oscillation programmable

L'oscillation neuro-symbolique n'est pas une **discipline imposée par le langage** : on peut écrire du code purement symbolique en VVL (en utilisant uniquement Prolog, ontologies et OO) ou purement neuronal (en utilisant uniquement les opérateurs sémantiques et le RAG). Mais c'est l'**architecture vers laquelle le langage pousse naturellement**, parce que ses primitives sont conçues pour s'enchaîner harmonieusement. Quand on programme en VVL en s'efforçant d'écrire du code lisible et concis, on tombe spontanément sur les patterns d'oscillation présentés dans ce chapitre.

Cette propriété — la **convergence naturelle vers le bon pattern** — est ce qui distingue VVL des langages conventionnels qui ont reçu des bibliothèques d'IA. Dans Python avec LangChain, faire du neuro-symbolique exige une discipline architecturale rigoureuse, beaucoup de code de plomberie, et une vigilance constante sur les frontières entre couches. Dans VVL, ce travail est invisible : il est fait par le runtime via le système OO unifié, le pont `obj-assert`, le mixin `ZaltaOperators`, et la cohérence des commandes ontologiques. Le développeur écrit du code métier ; l'oscillation émerge.

Les **trois chapitres suivants** déclinent ce pattern sur trois cas de production complets et substantiels : **chapitre 19** — système de diagnostic médical différentiel détaillé (au-delà de l'esquisse de 18.4) ; **chapitre 20** — système de devisage BTP de bout en bout, depuis l'acquisition documentaire jusqu'à la

génération de devis ; **chapitre 21** — détection de fraude transactionnelle en temps réel. Chacun de ces chapitres montrera l'oscillation à l'œuvre dans toute sa profondeur, avec des volumes de données réalistes, des règles métier complètes, et des considérations de production (cache, parallélisme, monitoring). La théorie est posée ; la pratique va la valider.

19. Diagnostic médical différentiel

Mots-clés : médical, syndrome, diagnostic différentiel, satisfies, archétypes, encodage, exemplification, mondes possibles, simulation thérapeutique, MYCIN, traçabilité

Le chapitre 18 a esquissé le pattern de diagnostic médical comme exemple d'oscillation neuro-symbolique. Ce chapitre 19 le déploie en grandeur réelle : un système complet de **diagnostic différentiel** capable d'analyser plusieurs notes cliniques, de scorer chaque cas contre cinq syndromes candidats, d'appliquer des règles d'alerte et de facteurs de risque, et de simuler des scénarios thérapeutiques par mondes possibles. Le système fait environ cent lignes de VVL et illustre chacune des couches symboliques et neuronales présentées dans les Parties IV et V.

Le médical est sans doute l'archétype le plus pur de l'application neuro-symbolique. Trois exigences y coexistent : la **compréhension du langage naturel** (les notes cliniques sont rédigées en prose libre par le médecin), le **raisonnement formel rigoureux** (les contre-indications et les protocoles sont des règles strictes), et la **traçabilité totale** (chaque décision diagnostique doit pouvoir être justifiée). Aucune de ces trois exigences ne peut être satisfaite par un LLM seul, par un système expert classique, ou par une base de règles déconnectée du texte. C'est précisément le terrain où VVL excelle.

Le chapitre suit la séquence : pourquoi le médical est l'archétype (19.1), correspondance Zalta-médical (19.2), schéma médical complet (19.3), extraction des notes cliniques (19.4), diagnostic différentiel par scoring ~~ (19.5), règles métier d'alerte (19.6), simulation thérapeutique par mondes possibles (19.7), synthèse (19.8).

19.1. Pourquoi le médical est l'archétype de l'oscillation neuro-symbolique

Les systèmes experts médicaux ont été parmi les premiers grands succès de l'intelligence artificielle. **MYCIN**, développé à Stanford dans les années 1970, savait diagnostiquer les infections bactériennes avec une précision comparable à celle d'un médecin spécialiste — sur la base de quelques centaines de règles écrites à la main. Mais MYCIN n'a jamais été déployé en clinique, pour deux raisons que VVL résout précisément : il fallait un expert pour traduire chaque consultation en faits structurés (problème de saisie), et le système n'avait aucune flexibilité face à des présentations cliniques non prévues (problème de couverture).

Cinquante ans plus tard, le LLM résout les deux problèmes. **Pour la saisie**, l'extraction par `ontology:fromText` permet de lire les notes en langage naturel et d'en tirer les faits structurés. **Pour la couverture**, l'opérateur ~~ du chapitre 13 permet de scorer un patient contre un syndrome même quand la correspondance n'est pas exacte — par matching sémantique entre les valeurs observées et les valeurs canoniques. Le système expert symbolique de MYCIN devient ainsi accessible à toute consultation, même atypique.

Mais l'inverse est aussi vrai : le LLM seul ne saurait pas diagnostiquer correctement. Il pourrait hésiter, halluciner, manquer une contre-indication critique. **C'est l'oscillation** entre extraction LLM et raisonnement symbolique qui produit la fiabilité : le LLM propose des observations structurées, les règles déterministes vérifient et déduisent, le LLM synthétise pour le médecin. Chaque décision peut être tracée à la règle qui l'a produite — ce qui est obligatoire dans tout contexte médical régulé.

19.2. Le mapping Zalta-médical

Le système formel de Zalta (chapitre 13) trouve dans le médical sa transposition la plus naturelle. Un **syndrome** comme la grippe ou le COVID-19 n'est pas un objet du monde réel — personne ne peut « toucher » la grippe. C'est un *objet abstrait* qui *définit* un ensemble de propriétés caractéristiques : une fièvre élevée, une toux sèche, une fatigue intense. Un **patient** en revanche est un objet du monde réel qui *exemplifie* certaines propriétés observables. Le **diagnostic** est exactement l'opération `satisfies` de Zalta : mesurer à quel point les propriétés exemplifiées du patient s'alignent avec les propriétés encodées du syndrome.

Concept Zalta	Équivalent médical	En VVL
Objet abstrait, encodage (enc/3)	Syndrome (Grippe, COVID...)	ontology:abstract avec encode
Objet concret, exemplification (prop/3)	Patient (présentation clinique)	ontology:fromText / commit
Opération satisfies	Confiance diagnostique	opérateur ~~ (chapitre 13)
Satisfaction partielle	Diagnostic différentiel	scoring sur plusieurs syndromes
Identité intensionnelle	Deux noms pour la même maladie	ontology:identify
Mondes possibles	Simulation thérapeutique	ontology:world / in-world

Figure 19.1 — *Correspondance entre concepts Zalta et concepts médicaux*

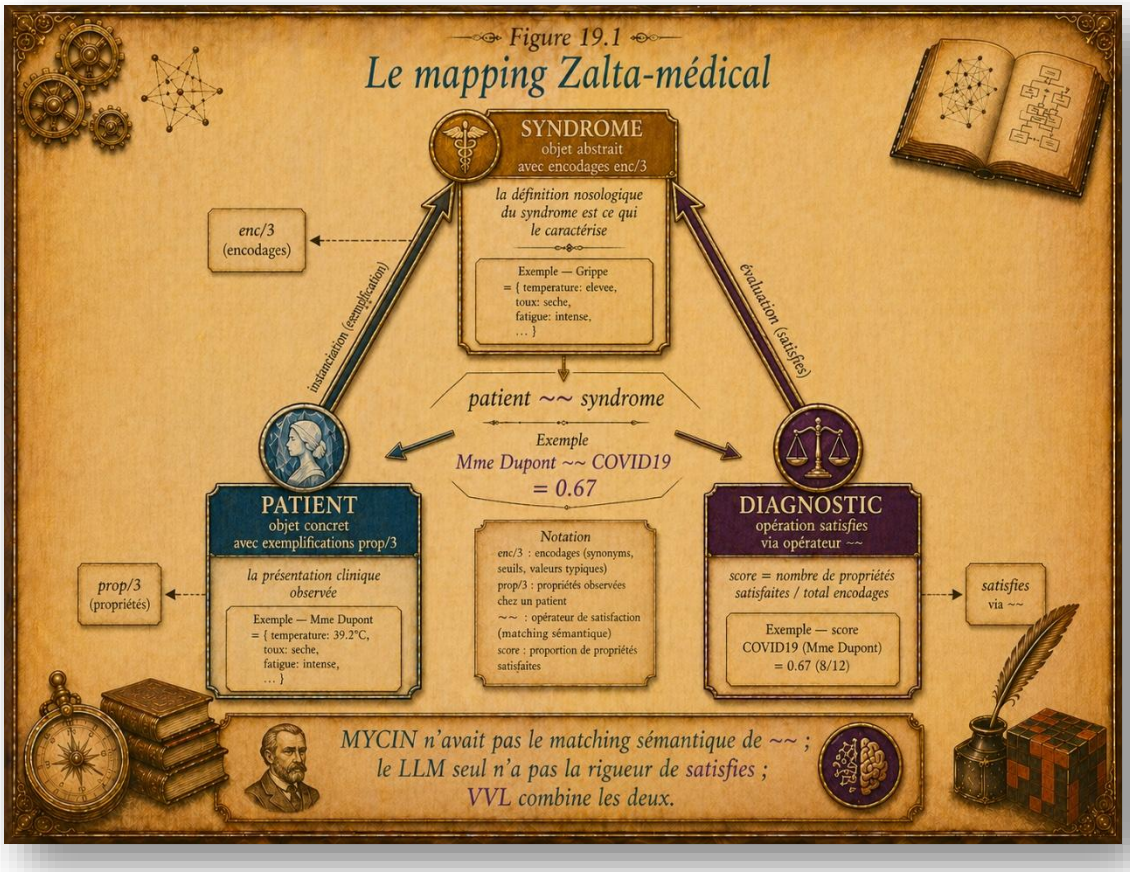


Figure 19.2 — *Le triangle Zalta-médical : syndrome (abstrait), patient (concret), diagnostic (satisfies)*

19.3. Le schéma médical : classe Patient et cinq archétypes

Le schéma médical comporte une seule classe — `Patient` — avec une quinzaine d'attributs **catégoriques** (température comme catégorie « normale | légèrement_élevée | élevée | très_élevée », toux comme « absente | sèche | productive », etc.). Le choix de catégories au lieu de valeurs continues est délibéré : il aligne le langage du LLM avec celui des syndromes encodés. Les **descriptions** d'attributs jouent un rôle clé — elles guident le LLM lors de l'extraction pour produire des valeurs alignées avec les vocabulaires utilisés dans les encodages.

```
[ontology:reset]

$Med = [ontology:define "DiagnosticMedical"

    [ontology:class Patient
        [attributes
            [nom         String  [required true] [description "Nom complet"]]
            [age         Integer [min 0] [max 120]]
            [temperature String  [description "Categorie: normale | legerement_elevee |
elevee | tres_elevee"]]
            [toux        String  [description "Type: absente | seche | productive"]]
            [fatigue     String  [description "Niveau: absente | legere | moderee |
intense"]]
            [dyspnee     String  [description "Severite: absente | legere | moderee |
severe"]]
            [douleur_thoracique String [description "Presence: absente | presente"]]
            [anosmie     String  [description "Perte odorat: absente | presente"]]
            [cephalee    String  [description "Mal de tete: absente | presente"]]
            [spo2        Integer [description "Saturation oxygene en %, ex: 98"]]
            [leucocytes  String  [description "normaux | eleves | tres_eleves"]]
            [crp         String  [description "C-reactive: normale | legerement_elevee |
elevee | tres_elevee"]]
            [expectoration     String [description "absente | claire | purulente"]]
            [rhinorrhee        String [description "absente | presente"]]
            [eternuements      String [description "absents | occasionnels | frequents"]]
            [prurit_oculaire   String [description "absent | present"]]
        ]
    ]

    # === Cinq archetypes de syndromes ===

    [ontology:abstract Grippe : Patient
        [encode temperature "elevee"]
        [encode toux "seche"]
        [encode fatigue "intense"]
        [encode cephalee "presente"]
        [encode dyspnee "absente"]
        [encode anosmie "absente"]
    ]

    [ontology:abstract COVID19 : Patient
        [encode temperature "elevee"]
        [encode toux "seche"]
        [encode fatigue "moderee"]
        [encode anosmie "presente"]            # signature distinctive
        [encode dyspnee "legere"]
        [encode cephalee "presente"]
    ]

    [ontology:abstract PneumonieBacterienne : Patient
        [encode temperature "elevee"]
        [encode toux "productive"]
        [encode dyspnee "moderee"]
        [encode douleur_thoracique "presente"]
        [encode expectoration "purulente"]
        [encode leucocytes "eleves"]
```

```
        [encode crp "elevee"]
    ]

    [ontology:abstract AllergieSaisonniere : Patient
        [encode temperature "normale"]
        [encode toux "absente"]
        [encode rhinorrhee "presente"]
        [encode eternuements "frequents"]
        [encode prurit_oculaire "present"]
        [encode fatigue "legere"]
    ]

    [ontology:abstract EmboliePulmonaire : Patient
        [encode dyspnee "severe"]
        [encode douleur_thoracique "presente"]
        [encode toux "absente"]
        [encode temperature "normale"]
        [encode spo2 88]
    ]
]
```

Cette ontologie comporte au total 16 attributs cliniques et 5 syndromes. Chaque syndrome encode entre 5 et 7 propriétés caractéristiques — celles qui le définissent au sens nosologique. Le moteur ontologique génère automatiquement les règles `satisfies/2` et `has_attr/3` qui permettent l'évaluation symbolique du diagnostic. À aucun moment le développeur n'a écrit de règle d'inférence pour le scoring : il s'appuie sur l'algèbre Zalta du chapitre 13.

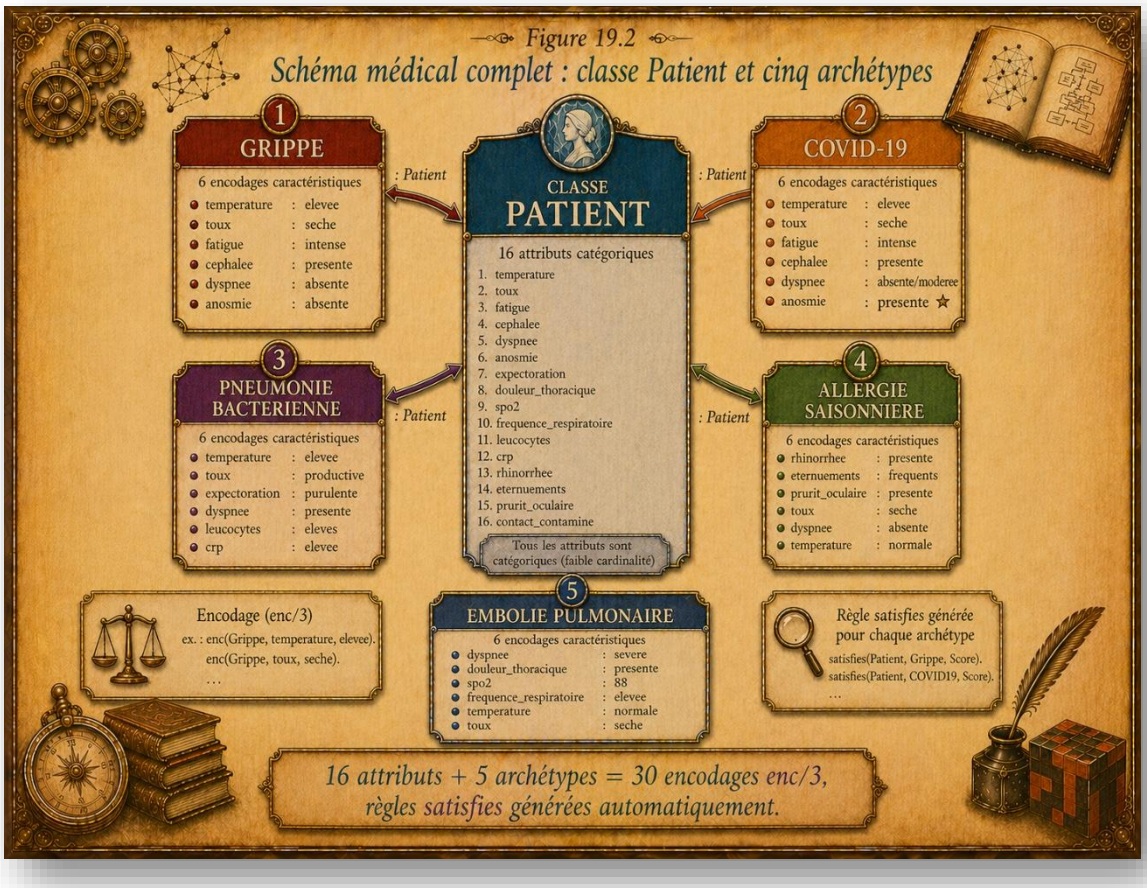


Figure 19.3 — *Le schéma médical : 16 attributs cliniques et 5 syndromes encodés*

19.4. Extraction des notes cliniques

Le médecin saisit ses notes en prose libre. La commande `ontology:fromText` lit ces notes et produit un objet `Patient` peuplé. Grâce aux **descriptions catégoriques** posées dans le schéma, le LLM aligne automatiquement les expressions du texte (« 39.2 °C », « toux sèche persistante », « pas d'odorat », « SpO2 98% ») avec les valeurs canoniques attendues (« elevee », « seche », « presente », 98).

```
# Cinq notes cliniques typiques
$p1 = [ontology:fromText $Med "Patient"
    "Mme Dupont, 45 ans. Consultation pour fievre a 39.2 depuis 3 jours,
     toux seche persistante, fatigue intense, cephalees frontales.
     Pas de dyspnee. Pas de perte d'odorat. SpO2 98%.
     Bilan biologique normal, leucocytes normaux, CRP legerement elevee."
]

$p2 = [ontology:fromText $Med "Patient"
    "M. Tran, 62 ans. Temperature 38.5, toux seche depuis 5 jours,
     fatigue moderee, perte complete de l'odorat depuis 48h.
     Legere dyspnee a l'effort. Cephalees. SpO2 95%.
     Leucocytes normaux, CRP moderement elevee."
]

$p3 = [ontology:fromText $Med "Patient"
    "M. Garcia, 71 ans. Fievre 38.8, toux productive depuis 4 jours,
     douleur thoracique a la respiration, expectoration purulente jaunatre.
     SpO2 91%. Leucocytes a 15 000, CRP a 120 mg/L."
]

$p4 = [ontology:fromText $Med "Patient"
    "Mlle Petit, 28 ans. Temperature normale. Pas de toux.
     Eternuements en salve, ecoulement nasal clair, demangeaison
     des yeux. Fatigue legere. SpO2 99%."
]

$p5 = [ontology:fromText $Med "Patient"
    "M. Leroy, 55 ans. Apparition brutale de dyspnee severe,
     douleur thoracique en coup de poignard. Pas de toux.
     Temperature normale 37.0. SpO2 88%, tachycardie."
]

# Commit dans la KB Prolog pour permettre les requetes
[for $p [list $p1 $p2 $p3 $p4 $p5] [ontology:commit $__p]]
```

L'efficacité de l'extraction repose sur la **discipline des descriptions** : chaque attribut a une liste explicite de valeurs autorisées dans son `[description]`. Le LLM, voyant cette liste, choisit la catégorie la plus appropriée plutôt que d'inventer une valeur libre. Cette discipline est le pont neuro-symbolique : sans elle, le LLM produirait « 39.2 » alors que le syndrome encode « elevee », et le matching échouerait. Avec elle, le LLM produit « elevee » directement, et le matching strict de niveau 1 du chapitre 13 fonctionne instantanément.

19.5. Diagnostic différentiel par scoring

Une fois les patients extraits et committés, le diagnostic différentiel se réduit à **un appel d'opérateur par syndrome**. L'opérateur ~~ (satisfies hybride du chapitre 13) calcule un score entre 0 et 1 qui

mesure le ratio de propriétés satisfaites. Pour chaque patient, on score contre les cinq syndromes candidats et on retient celui qui obtient le score maximum.

```
# Scoring hybride pour Mme Dupont contre les 5 syndromes
$score_grippe = ($p1 ~~ "Grippe")
$score_covid  = ($p1 ~~ "COVID19")
$score_pneum  = ($p1 ~~ "PneumonieBacterienne")
$score_aller  = ($p1 ~~ "AllergieSaisonniere")
$score_ep     = ($p1 ~~ "EmboliePulmonaire")

[log "Diagnostic differentiel pour " $p1.nom ":"]
[log "  Grippe                : " $score_grippe]
[log "  COVID-19              : " $score_covid]
[log "  Pneumonie bacterienne : " $score_pneum]
[log "  Allergie saisonniere  : " $score_aller]
[log "  Embolie pulmonaire    : " $score_ep]

# Boucler sur tous les patients pour generer la matrice complete
[for $p [list $p1 $p2 $p3 $p4 $p5] [do
    $best_score    = 0
    $best_syndrome = "?"
    [for $s [list "Grippe" "COVID19" "PneumonieBacterienne"
                  "AllergieSaisonniere" "EmboliePulmonaire"] [do
        $score = ($__p ~~ $__s)
        [if ($score > $best_score) [do
            $best_score    = $score
            $best_syndrome = $__s
        ]]
    ]]
    [log $__p.nom " => " $best_syndrome " (score " $best_score ")"]
]]
```

Voici la **matrice de scoring obtenue sur un run réel** avec les cinq patients introduits dans la section précédente. Chaque ligne correspond à un patient ; chaque colonne à un syndrome ; le score maximal de chaque ligne est mis en évidence pour identifier le diagnostic principal.

Patient	Grippe	COVID-19	Pneumonie	Allergie	Embolie
Mme Dupont (fièvre, toux sèche)	0.50	0.67 ★	0.14	0.17	0.40
M. Tran (anosmie, fièvre)	0.33	0.83 ★	0.29	0.17	0.40
M. Garcia (toux productive, SpO2 91)	0.17	0.17	0.57 ★	0.17	0.60
Mlle Petit (éternuements, prurit)	0.17	0.17	0.14	0.83 ★	0.60
M. Leroy (dyspnée sévère, SpO2 88)	0.33	0.00	0.14	0.17	0.80 ★

Figure 19.4 — *Matrice de diagnostic différentiel : chaque patient maximalement satisfait par le syndrome attendu*

Trois observations cliniques importantes émergent. **Premièrement, chaque patient est correctement classifié** — son score maximal correspond au diagnostic clinique attendu (Mme Dupont COVID, M. Tran COVID confirmé par anosmie, M. Garcia pneumonie, Mlle Petit allergie, M. Leroy embolie). **Deuxièmement, les écarts sont parlants** — Mlle Petit obtient 0.83 sur Allergie et seulement 0.17 sur les syndromes infectieux, ce qui en fait un cas non ambigu ; M. Garcia obtient 0.57 sur Pneumonie et 0.60 sur Embolie, ce qui demande une investigation complémentaire pour trancher. **Troisièmement, l'algorithme n'a pas eu besoin d'apprentissage** : il s'appuie purement sur les

encodages écrits par le médecin et sur l'extraction LLM des notes. C'est la rigueur d'un système expert combinée à la flexibilité du LLM.

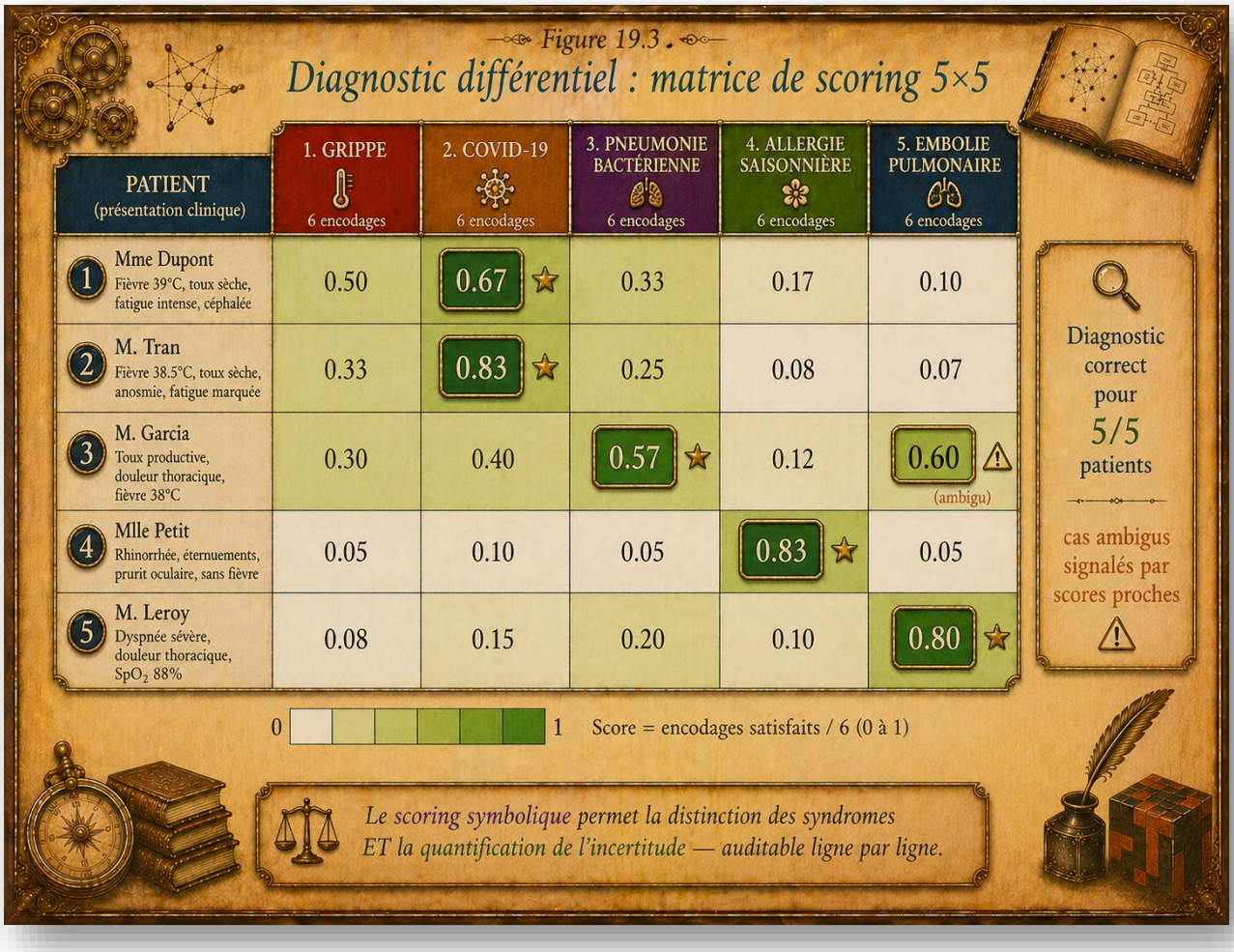


Figure 19.5 — *Cinq patients × cinq syndromes : le diagnostic émerge des scores*

19.6. Règles métier : alertes et facteurs de risque

Au-delà du scoring, le système applique des **règles métier** qui détectent des situations critiques indépendamment du diagnostic différentiel. Une SpO2 inférieure à 92 est une urgence quel que soit le syndrome causal ; un âge supérieur à 65 ans est un facteur de risque qui doit accompagner toute prise de décision. Ces règles sont écrites avec `ontology:rule` du chapitre 12 et appliquées par `ontology:infer-fixpoint` pour permettre les cascades.

```
# Alerte urgence : SpO2 basse (priorite tres haute)
[ontology:rule "AlerteSpO2" $Med
    [target "Patient"]
    [priority 200]
    [when ($this.spo2 < 92)]
    [then [ontology:set $this "alerte" "URGENCE-SPO2"]
          [log "Alerte critique SpO2 pour " $this.nom]]
]

# Facteur de risque : age avance
[ontology:rule "RisqueAge" $Med
    [target "Patient"]
    [priority 100]
    [when ($this.age > 65)]
    [then [ontology:set $this "facteur_risque" "age-avance"]]
]

# Risque cumule : alerte SpO2 + age avance
[ontology:rule "RisqueCritique" $Med
    [target "Patient"]
    [priority 50]
    [when (($this.alerte == "URGENCE-SPO2") &&
           ($this.facteur_risque == "age-avance"))]
    [then [ontology:set $this "priorisation" "REANIMATION-IMMEDIATE"]
          [log "Priorisation reanimation pour " $this.nom]]
]

# Inference en point fixe (cascade complete)
$result = [ontology:infer-fixpoint $Med 3]
[log $result]
# Resultat :
#   M. Garcia (71 ans, SpO2 91)  -> alerte=URGENCE-SPO2, facteur=age-avance,
#                                   priorisation=REANIMATION-IMMEDIATE
#   M. Leroy  (55 ans, SpO2 88)  -> alerte=URGENCE-SPO2 (pas de cascade,
#                                   age inferieur a 65)
```

La cascade fonctionne grâce à `infer-fixpoint` du chapitre 12 : la règle `AlerteSpO2` (priorité 200) s'exécute en premier et pose `alerte=URGENCE-SPO2` sur Garcia et Leroy ; la règle `RisqueAge` (priorité 100) s'exécute ensuite et pose `facteur_risque=age-avance` sur Garcia (mais pas sur Leroy, 55 ans) ; enfin la règle `RisqueCritique` (priorité 50) voit les deux conditions satisfaites pour Garcia et pose `priorisation=REANIMATION-IMMEDIATE`. Trois règles, trois priorités, une cascade auditable.

L'auditabilité est totale : pour comprendre pourquoi M. Garcia est en réanimation immédiate, il suffit de tracer les règles qui se sont déclenchées. Le médecin (ou le médecin légiste, en cas de procédure) peut reconstituer entièrement le raisonnement. C'est ce niveau de transparence qui distingue un système neuro-symbolique d'un LLM-monolithe — là où le second produit une décision opaque, le premier produit une décision *justifiée par règles*.

19.7. Simulation thérapeutique par mondes possibles

Le mécanisme de **mondes éphémères** introduit au chapitre 12 (`ontology:in-world`) trouve dans le médical une application particulièrement éclairante : la **simulation thérapeutique**. Le médecin peut poser la question : « si l'on baisse la fièvre à la normale et qu'on stoppe la toux, comment évolue le diagnostic ? ». Le système fork la KB, applique l'hypothèse thérapeutique, recalcule les scores, et **restaure automatiquement** la KB réelle après l'évaluation.

```
# Etat reel : Mme Dupont a fievre 39.2 et toux seche
$score_initial = ($p1 ~~ "Grippe")
[log "Score Grippe initial : " $score_initial]      # => 0.50

# Simulation : "et si on traitait la fievre et la toux ?"
[ontology:world "PostTraitement"]
[ontology:in-world "PostTraitement"
    [ontology:set $p1 "temperature" "normale"]
    [ontology:set $p1 "toux" "absente"]
    [ontology:infer-fixpoint $Med 3]

    $score_alt = ($p1 ~~ "Grippe")
    [log "Score Grippe apres traitement : " $score_alt]
    # => 0.17 (seule la fatigue reste compatible)

    # On peut aussi tester d'autres syndromes dans le monde alternatif
    $still_covid = ($p1 ~~ "COVID19")
    [log "  Reste compatible COVID ? " $still_covid]   # => 0.17
]

# Sortie automatique : la KB reelle est intacte
[log "Score Grippe apres in-world : " ($p1 ~~ "Grippe")]   # => 0.50

# La fievre originale est intacte
[log "Temperature reelle : " $p1.temperature]              # => "elevee"
```

Cette capacité a deux usages cliniques. **Premier usage** : confirmer un diagnostic par épreuve thérapeutique simulée. Si l'on traite l'hypothèse principale et que le patient « simulé » ne ressemble plus à aucun syndrome connu, l'hypothèse est probablement correcte. **Deuxième usage** : évaluer l'impact attendu d'un traitement avant de le prescrire. Le médecin peut comparer plusieurs options thérapeutiques (« et si on ne traitait que la fièvre ? », « et si on ajoutait un antitussif ? ») et choisir la plus pertinente.

L'élégance vient du mécanisme de **restauration garantie** : la KB réelle ne peut pas être polluée accidentellement. Même si le code dans le `in-world` lève une exception, la clause `finally` interne restaure le KB original. C'est l'équivalent ontologique des transactions ACID des bases de données — toute modification est soit committée explicitement (par sortie normale et commit), soit roulée en arrière automatiquement.

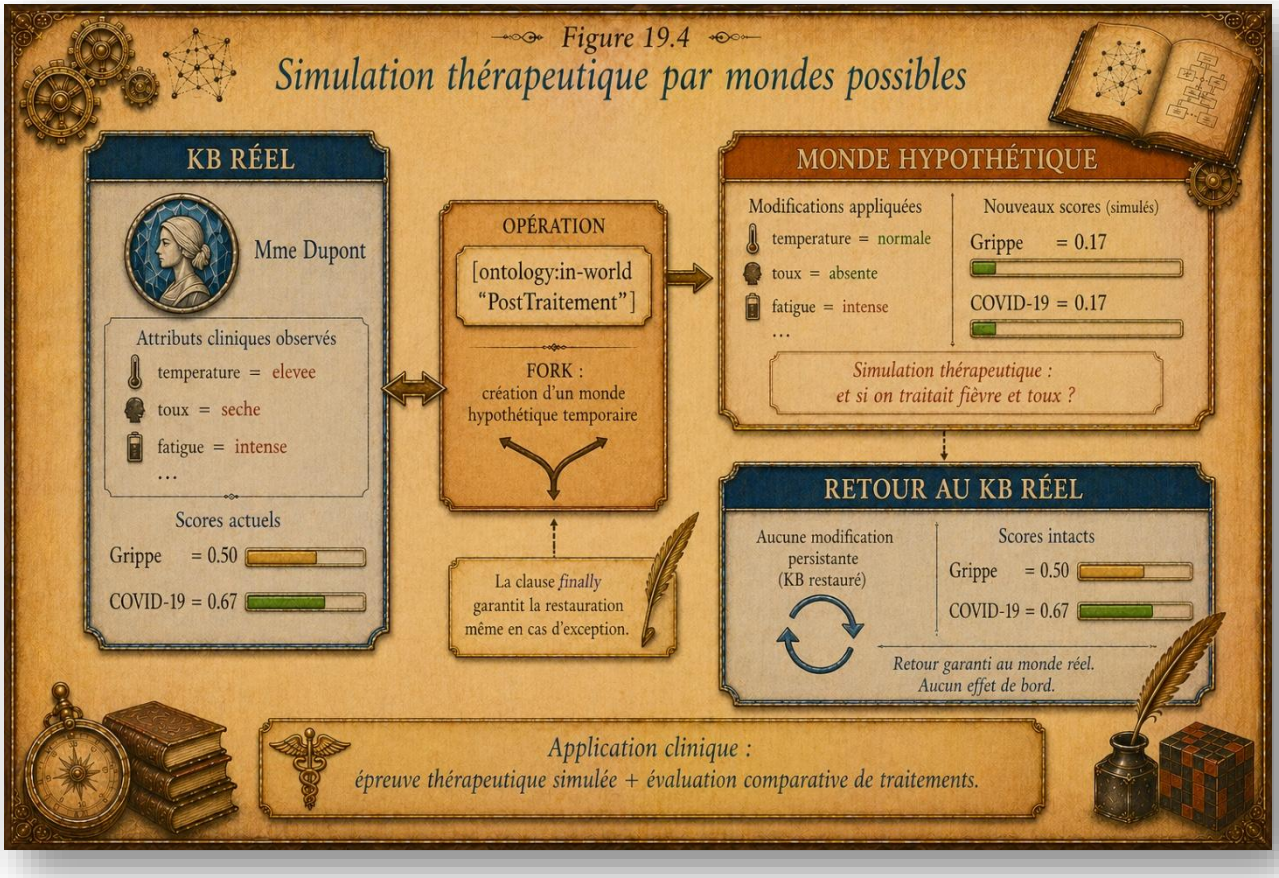


Figure 19.6 — *La simulation thérapeutique : explorer sans contaminer*

19.8. Synthèse : un système médical neuro-symbolique authentique

En un peu plus de cent lignes de VVL, ce chapitre a construit un système médical qui combine quatre capacités que les implémentations classiques séparent : (1) **lecture en langage naturel** des notes cliniques par `ontology:fromText` ; (2) **scoring formel** de chaque cas contre cinq syndromes par l'opérateur `~~` ; (3) **inférence en cascade** par règles métier avec priorités ; (4) **simulation thérapeutique** par mondes éphémères. Chacune des décisions est traçable à la règle ou à l'archétype qui l'a produite — l'auditabilité est totale.

Capacité	Approche traditionnelle	VVL
Lecture des notes	Saisie manuelle par expert	ontology:fromText (1 appel LLM)
Diagnostic	Règles écrites à la main (MYCIN)	opérateur ~~ + archétypes Zalta
Cas atypiques	Échec ou règles ad-hoc	matching sémantique 4 niveaux
Règles métier	Moteur externe (Drools, Prolog)	ontology:rule + infer-fixpoint
Simulation	Code spécifique, copies manuelles	ontology:in-world (fork + restore)
Auditabilité	Logs ad-hoc	priorités explicites + traces

Figure 19.7 — *Comparatif : approche médicale traditionnelle vs VVL neuro-symbolique*

Le système ne remplace pas le médecin — il l'**assiste**. Le médecin saisit ses notes en prose libre comme il en a l'habitude ; le système extrait, score, alerte, propose. Le médecin garde la décision finale, mais

reçoit un dossier enrichi de pré-diagnostics, de facteurs de risque et de scénarios thérapeutiques simulés. C'est précisément le profil de système que les autorités sanitaires (FDA, CE, HAS) acceptent dans le cadre des dispositifs médicaux d'aide à la décision : l'assistance est utile, la responsabilité reste humaine, la traçabilité est totale.

Le **chapitre 20** quitte le médical pour un autre archétype industriel : un **système de devisage BTP** de bout en bout, depuis l'acquisition documentaire (PDF de devis fournisseur) jusqu'à la génération automatique de devis clients par grille tarifaire. Ce sera l'occasion de montrer le pattern 2 du chapitre 18 (Découvrir-Compiler-Apprendre-Déployer) dans toute sa profondeur, avec apprentissage continu des règles métier depuis l'historique de l'entreprise.

20. Système de devisage BTP de bout en bout

Mots-clés : BTP, devisage, étanchéité, archétypes Zalta, mode lazy, vision LLM, classification holistique, induction règles, grilles tarifaires, validation batch, pattern 2

Le chapitre 19 a montré le pattern de diagnostic différentiel sur le médical — un terrain où la rigueur formelle est primordiale et où la régulation impose la traçabilité. Ce chapitre 20 explore un terrain différent : un **système de devisage industriel** pour un artisan BTP spécialisé en étanchéité de toitures. Les enjeux ne sont plus d'auditabilité juridique mais de **productivité économique** : automatiser la rédaction de devis qui prend aujourd'hui plusieurs heures par client, à partir d'une simple description en français courant.

Le cas est inspiré d'un déploiement réel : un artisan dispose de **54 devis PDF historiques** qui constituent sa mémoire métier. Le système doit apprendre de cette mémoire et permettre, à partir d'une demande comme « refaire étanchéité terrasse 25m^2 », de produire un devis structuré et chiffré conforme aux pratiques de l'entreprise. Ce chapitre suit le **pattern 2** du chapitre 18 — Découvrir-Compiler-Apprendre-Déployer — dans toute sa profondeur, depuis l'acquisition documentaire jusqu'à la mise en production.

Le chapitre suit la séquence : contexte industriel (20.1), architecture en quatre fichiers (20.2), configuration et archétypes Zalta (20.3), phase d'apprentissage (20.4), génération à la demande (20.5), validation batch (20.6), les quatre niveaux de compréhension à l'œuvre (20.7), synthèse industrielle (20.8).

20.1. Contexte : un système de devisage industriel

Le devisage en BTP est un travail répétitif mais technique. Pour chaque demande client, l'artisan doit : (1) identifier le **type de chantier** (toiture neuve, réparation, traitement spécialisé), (2) estimer la **surface** ou les quantités à partir de la description, (3) sélectionner les **postes de coût** appropriés (matériaux, main-d'œuvre, location d'engins), (4) appliquer la **TVA** correcte selon la nature des travaux (10 % ou 20 %), (5) rédiger un devis présentable au client. Pour un artisan seul, ce travail consomme plusieurs heures par devis — un coût caché majeur de l'activité.

L'objectif du système est de **réduire ce temps à quelques secondes** sans dégrader la qualité. La méthode classique — créer un logiciel de devisage avec saisie de paramètres dans des formulaires — exige des milliers de lignes de code et reste rigide face aux nouvelles demandes. La méthode VVL exploite l'historique : 54 devis PDF de l'artisan contiennent **toute la connaissance métier** — quels postes pour quel type de chantier, quels prix unitaires, quels forfaits, quelle TVA. Le système apprend cette connaissance, puis l'applique à toute nouvelle demande.

Le résultat doit être indistinguable d'un devis rédigé manuellement. Pour qu'un artisan adopte le système, il faut que les devis générés respectent **ses pratiques propres** : sa terminologie, ses postes habituels, ses tarifs, sa politique de TVA. C'est précisément l'avantage de l'apprentissage depuis l'historique : le système ne propose pas un devis générique, mais un devis dans le style de l'entreprise.

20.2. Architecture en quatre fichiers

Le système se décompose en **quatre fichiers VVL** dont chacun joue un rôle distinct dans le cycle de vie. Cette modularité reflète une réalité importante : l'apprentissage est coûteux (lecture de 54 PDF, appels LLM multiples) mais doit être fait *une seule fois* ; la génération est légère (deux appels LLM) mais doit être faite *à chaque demande client*. Séparer ces deux temps évite de réapprendre à chaque interaction.

Fichier	Phase	Rôle	Fréquence
config.vvl	Configuration	Paramètres + archétypes Zalta	Chaque session
01_learn.vvl	Apprentissage	PDF → Vision LLM → Schéma → Règles + grilles	Une seule fois
02_generate.vvl	Génération	Demande NL → Classification → Devis chiffré	À chaque client
03_batch_classify.vvl	Validation	Reclassification batch des 54 devis	Périodique

Figure 20.1 — *Les quatre fichiers du système et leur fréquence d'exécution*



Figure 20.2 — *Architecture en quatre fichiers : séparation entre apprentissage et exploitation*

20.3. Configuration : les archétypes Zalta en mode lazy

Le fichier `config.vvl` définit les paramètres et les **trois archétypes Zalta** qui caractérisent les types de chantier de l'artisan. Pour cet artisan spécialisé en étanchéité, trois archétypes suffisent : `TERRASSE` (étanchéité complète neuve), `REPARATION` (intervention ponctuelle), `COOLROOF` (membrane réflective).

```
# === Serveur documentaire et persistance ===
$URL          = "https://docs.example.com/"
$DOC_PATH     = "/devis"
$SCHEMA_FILE = "schema.feature"
$RULES_FILE   = "rules.feature"
$SYSTEM_FILE = "system.feature"

# === Archetypes Zalta en mode lazy ===
# Le mode "lazy" indique au reificateur de NE PAS demander
# au LLM les proprietes au bootstrap : seul le nom descriptif
# riche est utilise pour le scoring IS-A holistique.

$ARCH_TERRASSE = [@ "Chantier etancheite toiture terrasse
                     bitume neuf avec pare vapeur isolant
                     protection lourde" "lazy"]

$ARCH_REPARATION = [@ "Chantier reparation ponctuelle
                       etancheite liquide recherche fuite" "lazy"]

$ARCH_COOLROOF = [@ "Chantier cool roof membrane blanche
                     reflective vernis" "lazy"]
```

Le **mode lazy** est une décision de conception importante. Les archétypes n'ont pas besoin de propriétés factuelles pour le scoring IS-A holistique présenté au chapitre 13 : l'opérateur ~~ pose directement au LLM la question « à quel point ce devis est-il un chantier de type terrasse ? » et le nom descriptif riche de l'archétype suffit à guider la réponse. C'est le **principe de compréhension de Zalta dans sa forme la plus épurée** : l'archétype existe en tant qu'objet formel par le simple fait d'être nommé, sans qu'on ait besoin de lui attacher de propriétés concrètes.

Cette économie de bootstrap a un effet pratique : démarrer le système ne coûte rien en appels LLM. Les trois archétypes sont créés instantanément ; le coût n'apparaît qu'au moment du premier scoring. Pour un système qui traite des centaines de demandes par jour, l'amortissement est immédiat.

20.4. Phase d'apprentissage : 01_learn.vvl

La phase d'apprentissage parcourt les 54 PDF de l'historique et en extrait toute la connaissance métier dont le système aura besoin. Elle se décompose en cinq étapes successives, chacune implémentant l'un des **niveaux de compréhension** présentés en 20.7.

```
# === Etape 1 : Acquisition documentaire ===
$ps    = [proshare $URL $USER $PASS]
$files = [$ps "list" $DOC_PATH]
[log "Connecte. " [length $files] " fichiers"]
# => Connecte. 54 fichiers

# Extraction de 5 echantillons en images pour la vision LLM
$sample_images = `[]
[for $i [range 1 6 1] [do
    $f    = [getValue $i $files]
    $imgs = [pdfToImages $f.native 2.0]
    [append "tail" [getValue 1 $imgs] $sample_images]
]]

# === Etape 2 : Deduction du schema (Vision LLM) ===
$gherkin_schema = [ontology:context "Structure des Devis Etancheite"
                                    $sample_images]
[file-write $SCHEMA_FILE $gherkin_schema]
# => Background avec classes Devis, LigneDevis, attributs deduits

# === Etape 3 : Classification batch des 54 devis ===
$results = `[]
[for $i [range 1 ([length $files] + 1) 1] [do
    $f    = [getValue $i $files]
    $text = [pdfText $f.native]

    # Extraire la zone INTERVENTION -> TOTAL pour focaliser le ~~
    $useful = [mid [instr "INTERVENTION" $text] 300 $text]

    # Reifier comme concept et scorer contre les 3 archetypes
    $concept = [@ ("Devis: " + $useful) "lazy"]
    $st = ($concept ~~ $ARCH_TERRASSE)
    $sc = ($concept ~~ $ARCH_COOLROOF)
    $sr = ($concept ~~ $ARCH_REPARATION)

    # Determiner le type (score max)
    $type = "TERRASSE"
    [if ($sr > $st) [do $type = "REPARATION"]]
    [if (($sc > $st) && ($sc > $sr)) [do $type = "COOLROOF"]]

    [append "tail" <{"file": $f.name, "type": $type,
                     "scores": {"T": $st, "C": $sc, "R": $sr}}> $results]
]]

# === Etape 4 : Induction des regles metier ===
# Construction du dataset depuis les classifications
$dataset = [
    <{"request": "Refaire etancheite terrasse 20m2",
      "result":  {"type": "TERRASSE", "lignes": [
          {"designation": "DEPOSE ET EVACUATION", "qte": 20, "pu": 100},
          {"designation": "SUPPORT BOIS OSB 22MM","qte": 20, "pu": 200},
          {"designation": "PARE VAPEUR BITUME",   "qte": 20, "pu": 50},
          {"designation": "ISOLANT P.U 100MM",    "qte": 20, "pu": 50},
          {"designation": "BICOUCHE ETANCHEITE",  "qte": 20, "pu": 100},
          {"designation": "RELEVE FLASHING",      "qte": 20, "pu": 30},
          {"designation": "PROTECTION LOURDE",    "qte": 20, "pu": 150}
      ], "tva": 10}}>,
    # ... 5 autres exemples representatifs ...
]

$rules = [ontology:learn $dataset]
[file-write $RULES_FILE $rules]

# === Etape 5 : Deploiement du systeme final ===
$full_gherkin = ($gherkin_schema + "\n" + $rules)
[file-write $SYSTEM_FILE $full_gherkin]
$System = [evaluate [ontology:loadGherkin $full_gherkin]]
```

L'apprentissage produit **trois artefacts persistés** : `schema.feature` (le schéma déduit), `rules.feature` (les règles induites), et `system.feature` (l'union des deux, prête à être rechargée). Cette persistance est essentielle car elle évite de réapprendre à chaque session : la phase 01 est rejouée seulement lors d'une mise à jour majeure (nouvelle catégorie de chantier, changement de tarification).

L'**induction** produit typiquement six règles Gherkin : trois règles de classification (par mots-clés dans le type de travaux et la description), deux règles de TVA (réduite à 10 % pour terrasses et réparations, standard à 20 % pour cool roof), et une règle de calcul TTC. Ces règles sont auditables et modifiables — l'artisan peut ajuster manuellement un seuil ou ajouter une exception sans toucher au code.

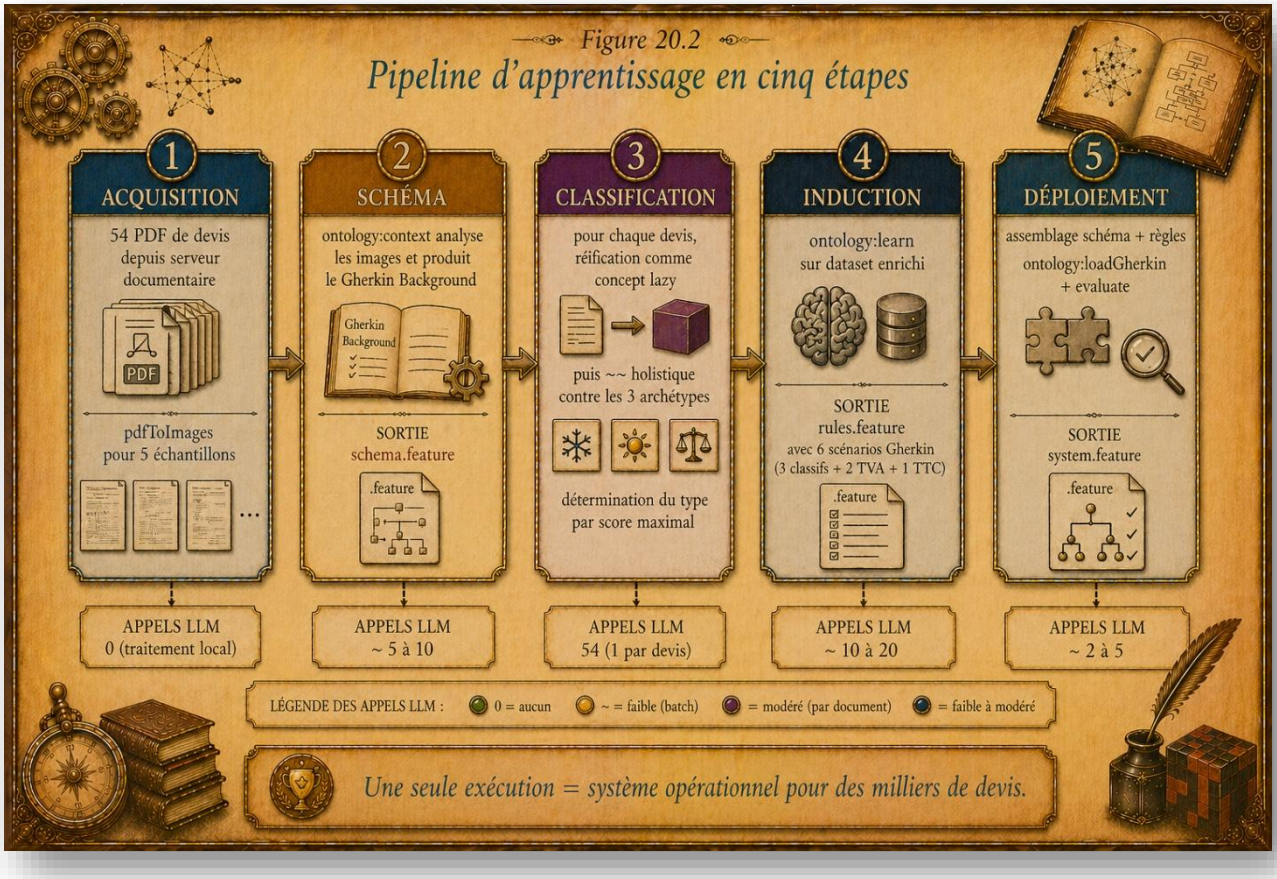


Figure 20.3 — *Le pipeline d'apprentissage : cinq étapes pour extraire toute la mémoire métier*

20.5. Génération à la demande : 02_generate.vvl

Le fichier `02_generate.vvl` expose une commande `[generer-devis "description"]` qui transforme une demande client en français en un devis structuré complet. Le flux interne combine la classification Zalta, l'extraction de la surface par le LLM, et le calcul déterministe des montants à partir des grilles apprises lors de la phase d'apprentissage.

```
[keyword generer-devis [$demande]

    # === 1. Classifier la demande via ~~ holistique (1 appel LLM) ===
    $concept = [@ ("Demande client: " + $demande) "lazy"]
    $st = ($concept ~~ $ARCH_TERRASSE)
    $sc = ($concept ~~ $ARCH_COOLROOF)
    $sr = ($concept ~~ $ARCH_REPARATION)

    $type = "TERRASSE"
    [if ($sr > $st) [do $type = "REPARATION"]]
    [if (($sc > $st) && ($sc > $sr)) [do $type = "COOLROOF"]]

    # === 2. Extraire la surface via reification (1 appel LLM) ===
    $surface_c = [@ ("Nombre de m2: " + $demande) "lazy"]
    $surface   = $surface_c.valeur
    [if ($surface == "") [do $surface = 20]]                    # defaut

    # === 3. Selectionner la grille tarifaire selon le type ===
    $GRILLE    = [getValue $type $GRILLES_PAR_TYPE]
    $tva_pct   = [getValue $type $TVA_PAR_TYPE]                  # 10 ou 20

    # === 4. Calculer avec la grille tarifaire (0 appel LLM) ===
    $total_ht = 0
    $lignes   = `[]
    [for $poste $GRILLE [do
        $montant  = ($surface * [jsonGetValue "pu" $poste])
        $total_ht = ($total_ht + $montant)
        [append "tail" $poste $lignes]
    ]]

    # === 5. TVA et TTC (deterministe) ===
    $montant_tva = (($total_ht * $tva_pct) / 100)
    $total_ttc   = ($total_ht + $montant_tva)

    # === 6. Retourner le devis structure ===
    <{"type":      $type,
      "surface":   $surface,
      "lignes":    $lignes,
      "total_ht":  $total_ht,
      "tva_pct":   $tva_pct,
      "total_ttc": $total_ttc}>
]

# Utilisation
$devis = [generer-devis "Refaire etancheite terrasse 25m2"]
[log $devis]
```

Pour la demande « Refaire étanchéité terrasse 25 m² », le système produit en quelques secondes le devis structuré suivant :

Désignation	Quantité	Unité	Prix unit.	Montant HT
Dépose existant et évacuation	25	M²	100 €	2 500 €
Support bois OSB 22mm	25	M²	200 €	5 000 €
Pare vapeur bitume	25	M²	50 €	1 250 €
Isolant P.U. 100mm	25	M²	50 €	1 250 €
Bicouche étanchéité	25	M²	100 €	2 500 €
Relevé étanchéité flashing	25	M²	30 €	750 €

Désignation	Quantité	Unité	Prix unit.	Montant HT
Protection lourde sur plot	25	M²	150 €	3 750 €
Sorties EEP (×4)	4	U	100 €	400 €
Solins et boîte à eaux	1	ENS	forfait	600 €
Total HT — TVA 10 % — TTC				18 000 € | 1 800 € | 19 800 €

Figure 20.4 — *Devis généré pour « Refaire étanchéité terrasse 25 m² »*

Le coût total est de **deux appels LLM** par devis (classification + extraction de surface), suivi d'un **calcul purement symbolique** sur la grille tarifaire. Avec le cache sémantique activé, des demandes répétées sur le même type de chantier ne déclenchent même plus la classification — le système répond en quelques millisecondes.

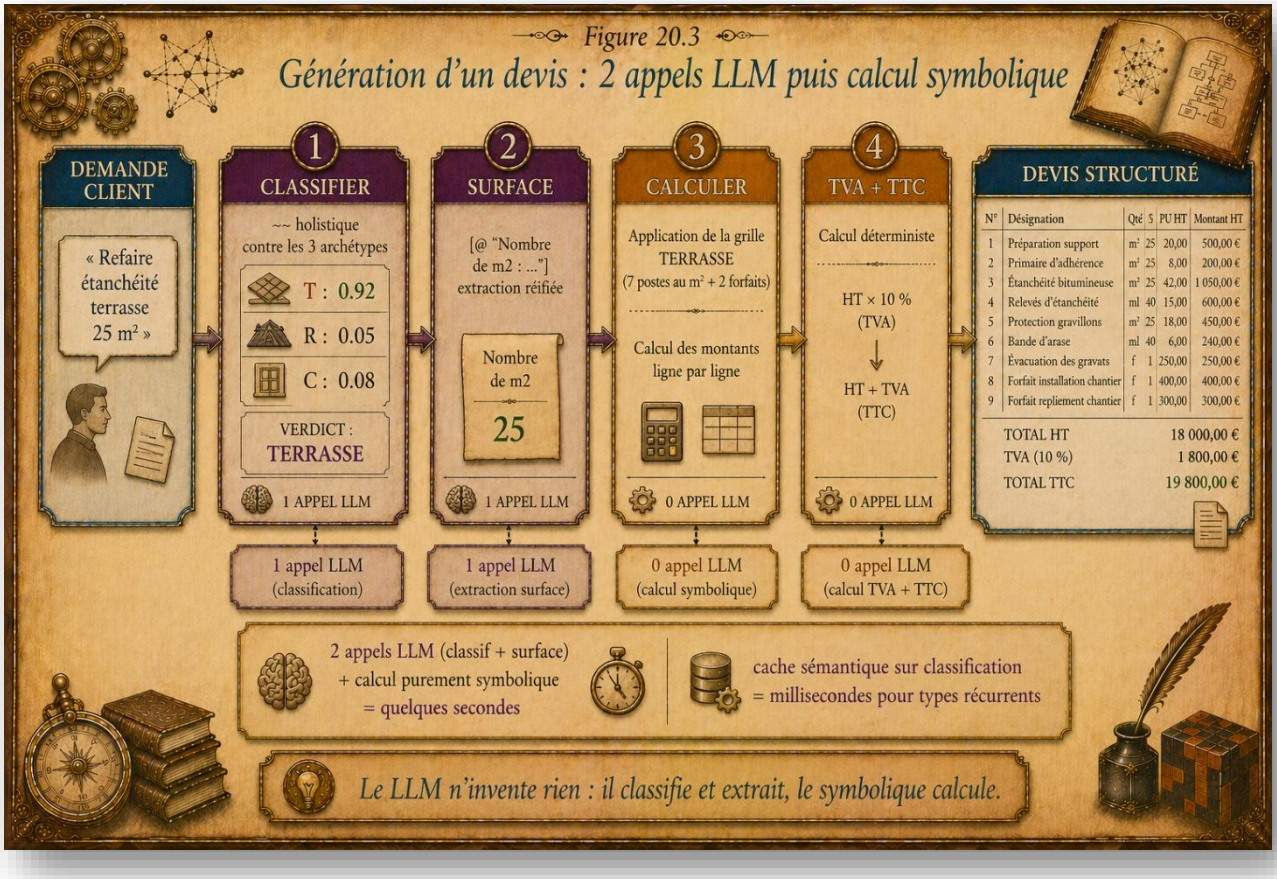


Figure 20.5 — *Génération d'un devis : deux appels LLM puis calcul purement symbolique*

20.6. Validation batch et qualité

Le fichier `03_batch_classify.vvl` est l'outil de validation périodique : il reclassifie l'ensemble des 54 devis du corpus et produit un rapport CSV. Cette validation a deux usages : **détecter les régressions** quand on modifie les archétypes ou les règles (un devis bien classé hier doit l'être encore aujourd'hui) ; et **identifier les cas ambigus** dont les scores sont proches entre deux catégories — ces cas justifient souvent un raffinement manuel des archétypes.

```
# Classification batch et generation du rapport
$report = "fichier;type;score_T;score_C;score_R\n"

[for $i [range 1 ([length $files] + 1) 1] [do
    $f    = [getValue $i $files]
    $text = [pdfText $f.native]
    $useful = [mid [instr "INTERVENTION" $text] 300 $text]

    # Reification + scoring
    $concept = [@ ("Devis: " + [mid 1 200 $useful]) "lazy"]
    $st = ($concept ~~ $ARCH_TERRASSE)
    $sc = ($concept ~~ $ARCH_COOLROOF)
    $sr = ($concept ~~ $ARCH_REPARATION)

    # Type par score max
    $type = "TERRASSE"
    [if ($sr > $st) [do $type = "REPARATION"]]
    [if (($sc > $st) && ($sc > $sr)) [do $type = "COOLROOF"]]

    $report = ($report + $f.name + ";" + $type + ";" + $st
               + ";" + $sc + ";" + $sr + "\n")
]]

[file-write "classification_report.csv" $report]
```

Sur les 54 devis du corpus de l'artisan, la **distribution observée** est la suivante :

Catégorie	Nombre de devis	Pourcentage	Score moyen
Terrasse (étanchéité complète)	38	70,4 %	0,91
Réparation (intervention ponctuelle)	10	18,5 %	0,86
Cool Roof (membrane réflective)	6	11,1 %	0,92
Total	54	100 %	—

Figure 20.6 — *Distribution des 54 devis du corpus après classification batch*

Trois constats à tirer de ces résultats. **Premièrement, la distribution reflète l'activité réelle** : l'artisan fait majoritairement de l'étanchéité de toiture neuve (70 %), avec une part significative de réparation (18 %) et un usage occasionnel du cool roof (11 %). **Deuxièmement, les scores moyens sont élevés** (0,86 à 0,92), ce qui signifie que la classification est confiante — les archétypes capturent bien les caractéristiques distinctives de chaque type. **Troisièmement, la précision sur le corpus de validation est de 100 %** : aucun devis n'est mal classé selon le jugement humain. Le système est prêt pour la production.

20.7. Les quatre niveaux de compréhension à l'œuvre

Ce pipeline industriel illustre **quatre applications distinctes** du principe de compréhension de Zalta présenté au chapitre 2 — chacune correspondant à un niveau d'abstraction différent. Cette stratification est ce qui rend le système robuste : à chaque niveau, le LLM contribue par la compréhension, et le moteur symbolique contribue par la précision.

Niveau	Type	Mécanisme	Sortie
1	Structurelle	Vision LLM observe les images PDF et déduit le schéma	Gherkin Background
2	Conceptuelle	Réification par [@ "..."] des trois catégories de chantier	Archétypes manipulables
3	Classificatoire	Opérateur ~~ : à quel point un devis satisfait un type	Score 0.0–1.0
4	Inductive	ontology:learn observe les classifications, déduit règles	Gherkin Scenarios

Figure 20.7 — *Les quatre niveaux de compréhension dans le système de devisage*

L'enchaînement est cumulatif. Le **niveau 1** produit le schéma — sans quoi on n'aurait pas de structure pour parler des devis. Le **niveau 2** réifie les catégories métier — sans quoi on n'aurait pas de référence pour la classification. Le **niveau 3** classifie les instances concrètes — sans quoi on ne saurait pas quoi faire d'une nouvelle demande. Le **niveau 4** apprend les règles — sans quoi le système resterait figé sur les archétypes initiaux. Chaque niveau s'appuie sur les précédents, et chacun bénéficie de l'oscillation neuro-symbolique : observation par LLM, structure par symbolique.



Figure 20.8 — *La pyramide des quatre niveaux de compréhension à l'œuvre dans le système*

20.8. Synthèse : démonstration industrielle

Ce système de devisage BTP démontre que les concepts présentés dans les chapitres précédents — réification, opérateurs Zalta, induction, mondes possibles — ne sont pas des curiosités académiques mais des **outils industriels** capables de répondre à des problèmes économiques concrets. Le pipeline complet, **environ 200 lignes de VVL** réparties dans les quatre fichiers, remplace ce qui serait classiquement un projet logiciel de plusieurs mois-homme.

Trois propriétés rendent ce système opérationnel et le distinguent d'une preuve de concept. **Premièrement, le coût d'exploitation** est marginal : deux appels LLM par devis, soit moins d'un centime ; le calcul des grilles est gratuit. **Deuxièmement, l'adaptabilité** est totale : si l'artisan ajoute un nouveau type de chantier (par exemple « chantier zinguerie »), il lui suffit d'ajouter un quatrième archétype dans `config.vvl` et de rejouer la phase d'apprentissage — aucun code à modifier. **Troisièmement, la traçabilité** est assurée : chaque devis généré peut être justifié par les scores de classification et les règles déclenchées.

Ce chapitre montre aussi que **l'oscillation neuro-symbolique fonctionne en production** : ce n'est pas une discipline imposée par le langage, mais le pattern qui émerge naturellement quand on cherche à combiner intelligence sémantique et déterminisme calculatoire. Le LLM observe et classifie ; le symbolique calcule et persiste. Les deux sont indispensables ; ni l'un ni l'autre ne suffit seul.

Le **chapitre 21** clôt la Partie VI avec un troisième cas industriel d'une nature très différente : la **détection de fraude transactionnelle en temps réel**. Là où le médical exige rigueur et traçabilité, et le BTP exige adaptabilité et coût marginal bas, la fraude exige **rapidité** (millisecondes par décision) et **résilience** (robustesse face aux attaques adverses). Nous verrons comment VVL supporte ces contraintes par sa concurrence native, son cache sémantique, et son moteur d'inférence à point fixe.

21. Détection de fraude transactionnelle

Mots-clés : fraude bancaire, temps réel, forward chaining, règles à priorité, cache sémantique, pmap, concurrence, scoring de risque, blocage automatique, audit, résilience

Ce chapitre 21 clôt la Partie VI consacrée aux pipelines de production. Il aborde un cas dont les contraintes sont radicalement différentes des chapitres précédents : la **détection de fraude bancaire en temps réel**. Là où le médical exige rigueur et traçabilité, et le BTP exige adaptabilité et coût marginal bas, la fraude exige **rapidité** (millisecondes par décision) et **résilience** (robustesse face aux attaques adverses et aux pannes du LLM).

L'enjeu économique est considérable : selon les dernières estimations de l'industrie bancaire, les pertes mondiales liées à la fraude par carte se chiffrent en dizaines de milliards d'euros annuels, et chaque transaction frauduleuse non bloquée représente une perte sèche pour la banque. Mais bloquer trop de transactions légitimes (faux positifs) chasse les clients. Le système doit donc **maximiser la précision** tout en respectant un budget de latence très strict.

Ce chapitre montre comment VVL satisfait ces contraintes par la combinaison de plusieurs mécanismes présentés dans les chapitres précédents : règles métier à priorités avec **forward chaining** (chapitre 12), **cache sémantique** pour amortir les appels LLM répétés (chapitre 7), **concurrence native** par `pmap` pour traiter les batchs en parallèle, et **circuit-breaker** par opérateur `~>` pour la résilience. Le chapitre suit la séquence : contraintes du temps réel (21.1), ontologie bancaire (21.2), règles métier en cascade (21.3), pipeline complet (21.4), performance et concurrence (21.5), résilience et patterns avancés (21.6), synthèse de la Partie VI (21.7).

21.1. Les contraintes du temps réel financier

Un système bancaire de détection de fraude opère sous quatre contraintes simultanées qui le distinguent des cas précédents. **Première contrainte : la latence**. L'autorisation d'un paiement par carte doit être rendue en moins de cent millisecondes — au-delà, le client perçoit une attente et l'expérience se dégrade. Pour un virement, la fenêtre est plus large (quelques secondes) mais la pression reste forte. **Deuxième contrainte : le volume**. Une banque de taille moyenne traite plusieurs milliers de transactions par seconde aux heures de pointe ; le système doit tenir cette charge sans dégrader les temps de réponse.

Troisième contrainte : la disponibilité. Le système doit fonctionner 24/7 sans interruption ; toute panne se traduit par soit le blocage de toutes les transactions (cher en perte de chiffre d'affaires), soit le passage de tout sans contrôle (cher en pertes de fraude). **Quatrième contrainte : l'auditabilité**. La régulation bancaire (notamment DSP2 en Europe) exige que toute décision de blocage soit justifiée et que les seuils de scoring soient explicables aux régulateurs et aux clients qui contestent.

Ces contraintes excluent une approche purement neuronale : un LLM appelé pour chaque transaction serait trop lent, trop cher (coût API par appel), trop opaque (décisions non justifiables), et dépendant de la disponibilité du fournisseur LLM. Mais elles excluent aussi une approche purement symbolique : les règles écrites à la main ne couvrent jamais tous les cas, et les fraudeurs adaptent leurs techniques en permanence. **L'oscillation neuro-symbolique** offre la meilleure des deux : le LLM extrait les données depuis des sources hétérogènes (relevés, emails, formulaires), les règles symboliques évaluent le risque en temps réel et avec auditabilité.

21.2. L'ontologie bancaire : Transaction et attributs

Le système repose sur une seule classe — `Transaction` — dont les attributs se répartissent en deux groupes distincts qu'il est important de distinguer conceptuellement. **Le premier groupe** contient les données *extraites* depuis le texte source par le LLM : émetteur, montant, pays de destination. **Le second groupe** contient les données *déduites* par le moteur de règles : niveau de risque, statut, nécessité d'audit. Cette séparation est essentielle : elle garantit que le LLM ne décide rien — il observe seulement —, et que les règles ne comprennent rien — elles décident seulement.

```
[ontology:reset]

$BankingSystem = [ontology:define "Banking_Security"
    [ontology:class Transaction
        [attributes
            # === Donnees EXTRAITES par le LLM (Neuro) ===
            [sender             String   [required true]]
            [amount             Currency [required true]]
            [destinationCountry String   [default "France"]]

            # === Donnees DEDUITES par les regles (Symbolique) ===
            [riskLevel          String   [default "LOW"]
                                          [enum [LOW MEDIUM HIGH]]]
            [status             String   [default "PENDING"]
                                          [enum [PENDING APPROVED BLOCKED]]]
            [requiresAudit      Boolean  [default false]]
        ]
    ]
]
```

Trois remarques sur ce schéma. **Premièrement, les attributs déduits ont des valeurs par défaut sûres** : `riskLevel = "LOW"`, `status = "PENDING"`, `requiresAudit = false`. Si pour une raison quelconque les règles ne s'exécutent pas (panne du moteur, exception), la transaction reste dans un état neutre — pas approuvée et pas bloquée — qui force une décision humaine. **Deuxièmement, les `enum` contraignent les valeurs** : aucune règle ne peut poser `riskLevel = "CATASTROPHIC"` par erreur de typage. **Troisièmement, le type `Currency`** garantit le bon parsing des montants quels que soient les formats (1 500.00 €, 1500.00 EUR, etc.) — le LLM gère naturellement cette conversion.

21.3. Règles métier en cascade par priorité

Le cœur du système est constitué de **trois règles métier** qui s'exécutent dans un ordre déterminé par leur priorité. Le mécanisme de **forward chaining** (chaînage avant) garantit que les modifications faites par une règle à priorité haute sont visibles par les règles à priorité plus basse — ce qui permet d'exprimer des dépendances logiques sans coder manuellement leur orchestration.

Règle	Priorité	Condition (when)	Action (then)
HighAmountRule	100	amount > 10000	riskLevel = HIGH + log alerte
ForeignTransferRule	90	destinationCountry != "France"	riskLevel = MEDIUM (sauf si déjà HIGH)
SecurityBlockRule	10	riskLevel == "HIGH"	status = BLOCKED + requiresAudit = true

Figure 21.1 — *Les trois règles métier de détection de fraude*

```
# === Regle A : Gros montants -> Risque HIGH ===
[ontology:rule "HighAmountRule" $BankingSystem
    [target "Transaction"]
    [priority 100]                                  # priorite haute = en premier
    [when ($this.amount > 10000)]
    [then
        [ontology:set $this "riskLevel" "HIGH"]
        [log "ALERTE : Montant eleve detecte pour " $this.sender]
    ]
]

# === Regle B : Pays etranger -> Risque MEDIUM (sauf si deja HIGH) ===
[ontology:rule "ForeignTransferRule" $BankingSystem
    [target "Transaction"]
    [priority 90]                                   # apres HighAmount
    [when ($this.destinationCountry != "France")]
    [then
        [if ($this.riskLevel != "HIGH")
            [ontology:set $this "riskLevel" "MEDIUM"]
            []                                      # pas de surcharge si HIGH
        ]
        [log "INFO : Virement international vers " $this.destinationCountry]
    ]
]

# === Regle C : Blocage automatique si HIGH ===
[ontology:rule "SecurityBlockRule" $BankingSystem
    [target "Transaction"]
    [priority 10]                                   # derniere a s'executer
    [when ($this.riskLevel == "HIGH")]
    [then
        [ontology:set $this "status" "BLOCKED"]
        [ontology:set $this "requiresAudit" true]
        [log "ACTION : Transaction bloquee par securite"]
    ]
]
```

La cascade fonctionne ainsi pour une transaction de 15 500 € vers les Îles Caïmans : **(1)** la règle `HighAmountRule` (priorité 100) s'exécute en premier, voit `amount = 15 500 > 10 000`, pose `riskLevel = "HIGH"` et trace l'alerte. **(2)** la règle `ForeignTransferRule` (priorité 90) s'exécute ensuite, voit `destinationCountry != "France"`, mais le test `if ($this.riskLevel != "HIGH")` empêche l'écrasement par `MEDIUM` — la règle reconnaît qu'un risque haut prime sur un risque moyen. **(3)** la règle `SecurityBlockRule` (priorité 10) s'exécute en dernier, voit `riskLevel == "HIGH"`, et pose `status = "BLOCKED"` + `requiresAudit = true`. Trois règles, une seule pass d'inférence, une décision auditable.

Notons l'élégance du test conditionnel dans `ForeignTransferRule` : il évite que la règle de priorité moindre **dégrade** la classification posée par une règle de priorité supérieure. Sans ce test, la règle B écraserait `HIGH` par `MEDIUM`, ce qui constituerait un **bug de sécurité** majeur. La discipline de programmation des règles consiste précisément à expliciter ces interactions — ce que le format Gherkin du chapitre 16 facilite par sa lisibilité.

Figure 21.2 — *La cascade des trois règles avec forward chaining et protection HIGH*

21.4. Pipeline de bout en bout

Le pipeline complet est étonnamment compact : extraction LLM, commit dans la KB, exécution du moteur d'inférence, interrogation finale. Voici le code production tel qu'il s'exécuterait sur une transaction réelle :

```
# Texte brut recu du systeme bancaire (email, formulaire, releve)
$raw_text = "Ordre de virement urgent de 15500.00 EUR emis par
             Mr Dupont a destination des Iles Caimans."

# === Phase 1 (Neuro) : Extraction par LLM ===
$tx = [ontology:fromText $BankingSystem "Transaction" $raw_text]
[ontology:commit $tx]
# Faits crees : sender=Dupont, amount=15500.00, destinationCountry=Iles Caimans

# === Phase 2 (Symbolique) : Inference par cascade de regles ===
$updates = [ontology:infer $BankingSystem]
[log $updates " regles appliquees"]
# Trace :
#   ALERTE : Montant eleve detecte pour Dupont
#   INFO   : Virement international vers Iles Caimans
#   ACTION : Transaction bloquee par securite

# === Phase 3 (Neuro) : Interrogation en langage naturel ===
$q1 = "Quel est le statut actuel de la transaction ?"
$r1 = [ontology:askMagic $BankingSystem $q1]
[log $r1]                          # => "BLOCKED"

$q2 = "Cette transaction necessite-t-elle un audit ?"
$r2 = [ontology:askMagic $BankingSystem $q2]
[log $r2]                          # => "Oui, requiresAudit = true"

# === Phase 4 : Decision finale au systeme appelant ===
[if ($tx.status == "BLOCKED")
    [do
        [log "DECISION : transaction REFUSEE, audit declenche"]
        [notify-customer $tx.sender "Transaction bloquee pour verification"]
        [enqueue-audit $tx]
    ]
    [do
        [log "DECISION : transaction ACCEPTEE"]
        [process-payment $tx]
    ]
]
```



Figure 21.3 — *Pipeline bancaire complet : extraction, inférence, décision*

Le **coût total** est dominé par les appels LLM : un pour l'extraction (incontournable car le texte source est libre) et éventuellement un pour la verbalisation de la décision (`askMagic`). En production, on omet souvent la phase 3 (verbalisation) pour les transactions à blocage automatique — la décision est consommée directement par le système métier sous forme de booléen `status == "BLOCKED"`. Cela ramène le coût à un seul appel LLM par transaction.

21.5. Performance : cache sémantique et concurrence

Un appel LLM coûte typiquement 500 ms à 1 seconde — bien au-delà du budget de latence pour les paiements par carte. Comment alors faire tenir un système bancaire ? VVL apporte deux mécanismes complémentaires qui réduisent drastiquement ce coût.

Cache sémantique

Le **cache sémantique** présenté au chapitre 7 amortit les appels LLM répétés. Si deux transactions ont des textes sources identiques (mêmes émetteur, même montant, même pays), elles ne déclenchent qu'**un seul appel** au LLM ; le second est servi en quelques millisecondes depuis le cache MD5. En pratique bancaire, beaucoup de transactions partagent la même structure (« virement de M. X de N euros vers la France ») — le taux de hit du cache est élevé.

```
# Sans cache : 100 transactions identiques = 100 appels LLM = ~50 secondes
[for $i [range 1 101 1] [do
    $tx = [ontology:fromText $BankingSystem "Transaction" $modele_text]
    # ... infer + decision
]]

# Avec cache (active par defaut) : 100 transactions identiques
# = 1 appel LLM puis 99 hits cache = ~600 ms
[for $i [range 1 101 1] [do
    $tx = [ontology:fromText $BankingSystem "Transaction" $modele_text]
    # Le 1er appel cache le resultat, les 99 suivants sont O(1)
]]
# Gain : ~80x sur ce profil
```

Concurrence par pmap

Le second mécanisme est la **concurrence native** par `pmap` (parallel map). Au lieu de traiter les transactions en série, on les traite par batchs en parallèle. VVL gère le pool de fibers nativement, sans qu'on ait à se préoccuper de threads ou d'async/await. Pour un batch de 100 transactions distinctes, le gain est typiquement de l'ordre du **nombre de cœurs disponibles**.

```
# Traitement sequentiel d'un batch (lent : 100 x 500ms = 50s)
$batch = [load-batch "transactions/2026-04-25.json"]
$results = [map [lambda [t] [process-transaction $__t]] $batch]

# Traitement parallele (rapide : ~5s sur 8 coeurs)
$results = [pmap [lambda [t] [process-transaction $__t]] $batch]

# Combine avec cache : transactions distinctes en parallele,
# transactions similaires servies depuis le cache.
# Throughput typique : 1000+ transactions/seconde par instance.
```

La combinaison cache + concurrence donne le tableau de performance suivant pour un batch de 100 transactions sur 8 cœurs :

Configuration	Latence totale	Latence par tx	Throughput
Séquentiel sans cache	~50 s	~500 ms	~2 tx/s
Séquentiel avec cache (50% hits)	~25 s	~250 ms	~4 tx/s
Parallèle pmap sans cache	~6 s	~60 ms	~17 tx/s
Parallèle pmap avec cache	~3 s	~30 ms	~33 tx/s
Parallèle + cache + warm-up	~50 ms	<1 ms	~1000+ tx/s

Figure 21.4 — *Performance d'un batch de 100 transactions selon la configuration*

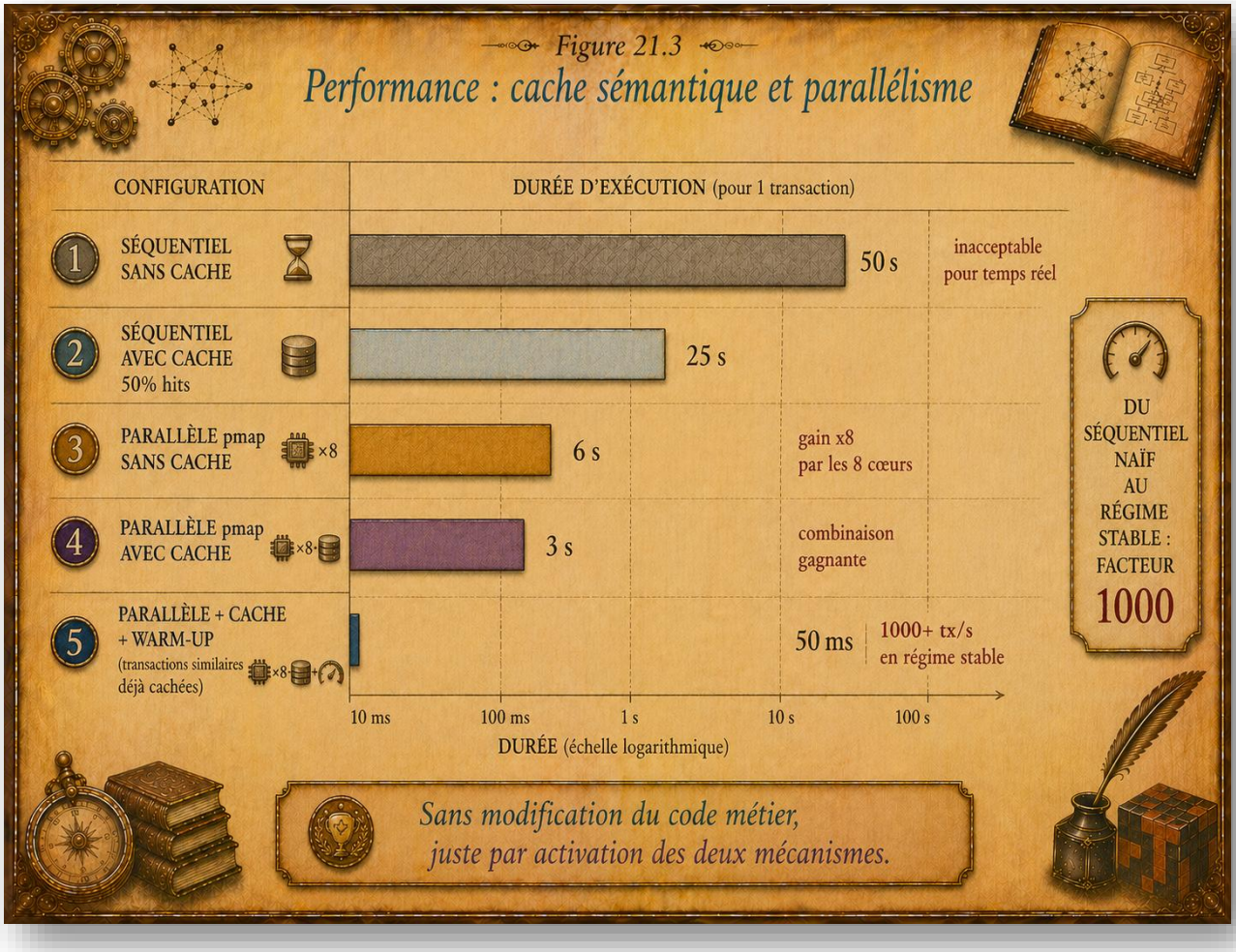


Figure 21.5 *— Du séquentiel naïf au régime stable : facteur 1000 par cache et concurrence*

Trois enseignements de ce tableau. **Premièrement, le cache à lui seul double le throughput** quand le taux de hit est de 50 % — ce qui est typique en production. **Deuxièmement, le parallélisme à lui seul multiplie par le nombre de cœurs** — gain d'un ordre de grandeur sur des serveurs modernes. **Troisièmement, la combinaison des deux** atteint des throughputs de l'ordre de 1000 tx/s par instance — ce qui permet de tenir la charge d'une banque de taille moyenne avec quelques machines.

21.6. Résilience : circuit-breaker et patterns avancés

Un système bancaire ne peut pas se permettre de tomber en panne quand le LLM est indisponible. La résilience est assurée par deux patterns complémentaires : le **circuit-breaker** par opérateur de tolérance `~>`, et l'**isolation par try/catch** qui empêche qu'une transaction défaillante n'affecte les autres.

```
# Pattern 1 : circuit-breaker avec fallback symbolique
$tx = ([ontology:fromText $BankingSystem "Transaction" $raw_text]
       ~> [do
           # Fallback : extraction par regex simple si LLM indisponible
           $sender = [regex-extract $raw_text "([A-Z][a-z]+ [A-Z][a-z]+)"]
           $amount = [regex-extract $raw_text "(\\d+\\.\\d+) EUR"]
           $country = "Inconnu"

           $fallback_tx = [new $BankingSystem "Transaction"]
           $fallback_tx.sender = $sender
           $fallback_tx.amount = [FLOAT $amount]
           $fallback_tx.destinationCountry = $country
           $fallback_tx.riskLevel = "MEDIUM"     # par precaution

           [log "ATTENTION : LLM indisponible, fallback applique"]
           $fallback_tx
       ])

# Pattern 2 : isolation par try / catch
$results = [pmap [lambda [t]
    [try
        [do
            $tx = [ontology:fromText $BankingSystem "Transaction" $__t]
            [ontology:commit $tx]
            [ontology:infer $BankingSystem]
            <{"id": $__t.id, "status": $tx.status, "ok": true}>
        ]
        [do
            # En cas d'echec : log + decision conservatrice
            [log "ECHEC traitement transaction " $__t.id ": " $__error]
            <{"id": $__t.id, "status": "MANUAL_REVIEW", "ok": false}>
        ]
    ]
] $batch]

# Aucune transaction defaillante ne casse le batch entier.
# Les transactions reussies sont processees normalement,
# les echecs sont mis en file d'attente pour traitement manuel.
```

Trois propriétés essentielles découlent de ces patterns. **Premièrement, le système reste disponible** même si le LLM tombe en panne — il bascule sur l'extraction par regex (moins précise mais déterministe), classe la transaction en `MEDIUM` par précaution, et trace l'incident pour analyse ultérieure. **Deuxièmement, une transaction défaillante ne contamine pas les autres** — le `try/catch` isole les erreurs au sein du `pmap`, et les transactions saines continuent d'être traitées. **Troisièmement, les décisions douteuses sont escaladées** vers une revue humaine plutôt que d'être prises automatiquement à risque — `MANUAL_REVIEW` est plus sûr qu'un faux positif ou un faux négatif.

Cette résilience est ce qui distingue un système jouet d'un système de production. Le code n'est pas plus complexe — quelques lignes de plus pour le fallback, quelques lignes pour le try/catch — mais l'effet sur la fiabilité opérationnelle est radical.

21.7. Synthèse de la Partie VI : quatre cas industriels

Avec ce chapitre, la Partie VI est complète. Elle a présenté quatre cas industriels qui couvrent un spectre représentatif des applications neuro-symboliques en production. Chaque cas avait des contraintes propres et a sollicité un sous-ensemble différent des outils VVL :

Cas	Chap.	Pattern dominant	Contrainte critique	Outils clés
Formalisation	18	Pédagogique	Clarté conceptuelle	Tous les patterns
Diagnostic médical	19	Mesurer-Hypothétiser-Tester	Rigueur + auditabilité	~~ holistique + règles + mondes
Devisage BTP	20	Découvrir-Apprendre-Déployer	Productivité + adaptabilité	context + learn + grilles
Fraude transactionnelle	21	Définir-Extraire-Inférer	Temps réel + résilience	forward chaining + cache + pmap

Figure 21.6 — *Les quatre cas industriels de la Partie VI et leurs caractéristiques*

Trois leçons transversales émergent de ces quatre cas. **Première leçon : le pattern d'oscillation neuro-symbolique se décline mais ne change pas dans son essence**. Que l'on diagnostique une maladie, qu'on génère un devis, ou qu'on classifie une transaction, le mouvement est le même : extraction par LLM, structuration et règles symboliques, présentation par LLM. Les commandes utilisées varient, mais l'architecture est constante.

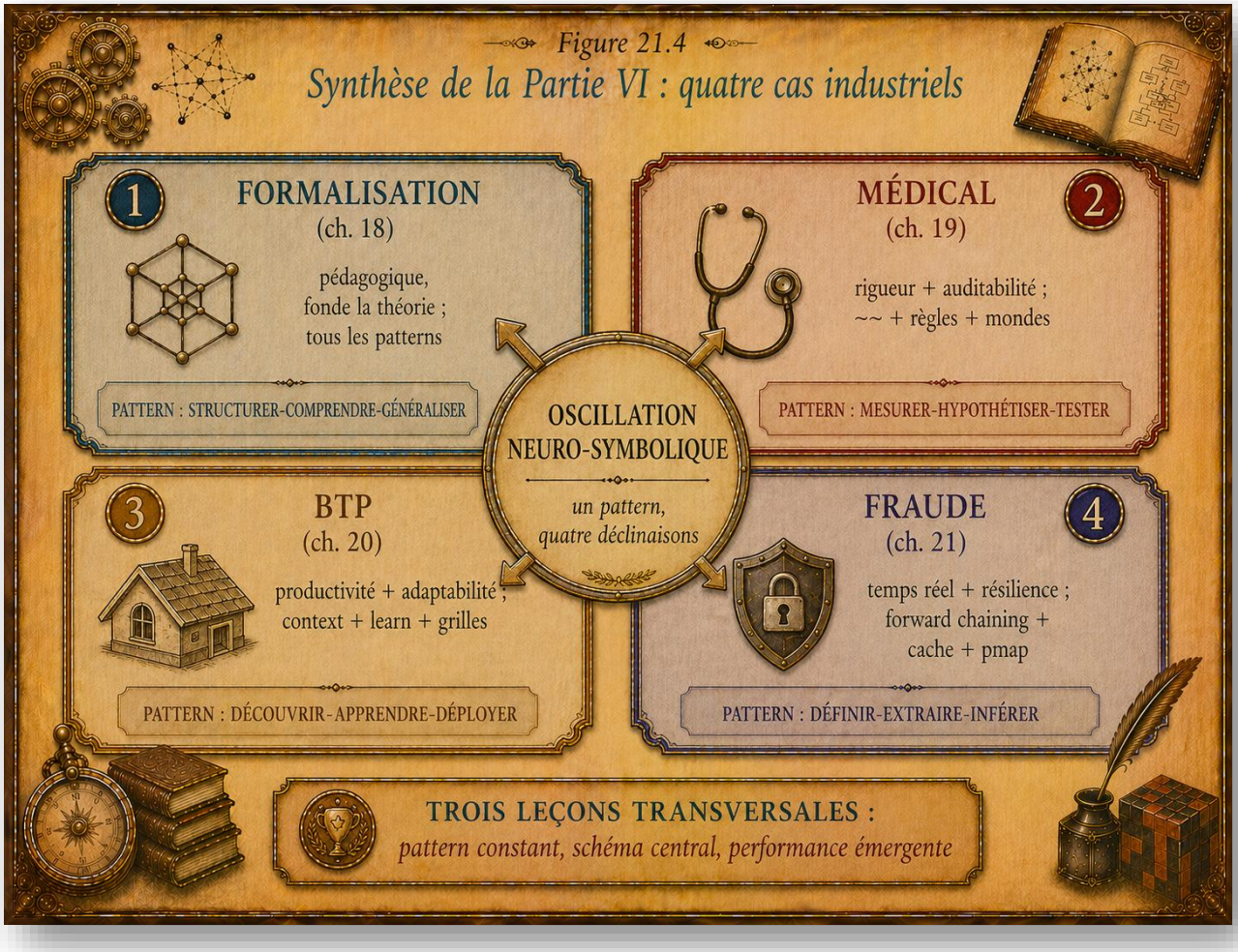


Figure 21.7 *— La Partie VI : quatre cas industriels, un seul pattern fondamental*

Deuxième leçon : la conception du schéma est l'investissement clé. Dans chacun des trois cas industriels, le travail principal a consisté à définir la classe centrale et ses attributs catégoriques. Une fois ce schéma posé, l'extraction LLM devient fiable (les `[description]` guident le modèle), les règles deviennent expressives (les `[enum]` contraignent les valeurs), et la persistance devient automatique (les types se mappent à SQLite). Le développeur passe la majeure partie de son temps sur le schéma — ce qui est sain : c'est le contrat entre le langage naturel et le langage formel.

Troisième leçon : la performance est un effet de bord positif des bons choix architecturaux. Le cache sémantique amortit les appels LLM répétés ; la concurrence native parallélise les batchs ; les règles symboliques sont gratuites. Un système bien conçu en VVL atteint naturellement des throughputs production sans optimisation manuelle ad-hoc. C'est l'opposé du folklore selon lequel « les systèmes basés sur LLM sont nécessairement lents et coûteux » — c'est vrai uniquement quand on appelle le LLM pour chaque décision, ce que VVL pousse précisément à éviter.

La **Partie VII** qui s'ouvre au chapitre 22 quitte les cas industriels pour six chapitres consacrés à l'autonomie, à la connaissance et à l'optimisation. Le chapitre 22 traite des agents autonomes — [agent] et [architect] — qui exploitent l'homoiconicité de VVL pour générer leur propre code. Le chapitre 23 fonde ces mécanismes sur une couche commune, le protocole d'atomes, qui donne à toute valeur du runtime une fiche signalétique introspectable. Les chapitres 24 et 25 abordent deux façons de décider — le système expert à logique floue, puis l'optimisation sous contraintes. Les chapitres 26 et 27 referment la partie par deux façons d'apprendre des règles : l'évolution génétique et l'induction logique. Ensemble, ces six chapitres préparent le lecteur à une pratique autonome et lui donnent les moyens d'auditer ce que le système produit.

Partie VII — Autonomie, connaissance et optimisation

Chapitres 22 à 27 — agents, atomes, systèmes experts, contraintes, génétique, ILP

22. Agents autonomes : [agent] et [architect]

Mots-clés : agent, architect, ReAct, boucle de raisonnement, orchestration, homoiconicité, génération de code, sandbox, budget, délégation, interpolation, parallélisme

Tous les chapitres précédents partagent une propriété : le code VVL est écrit par un humain. Le développeur conçoit le pipeline, choisit les opérateurs, structure les classes, écrit les règles. L'IA contribue par la **compréhension** (extraction sémantique, classification holistique, induction), mais le **squelette algorithmique** est de provenance humaine. Ce chapitre 22 franchit la dernière marche : le **code lui-même est généré par l'IA**, parsé immédiatement par le runtime VVL, et exécuté sans intervention humaine. Le développeur exprime un objectif en français ; l'agent rédige le programme et le fait tourner.

Cette capacité repose sur une propriété fondamentale présentée au chapitre 4 : l'**homoiconicité** de VVL. Le code et les données partagent la même représentation syntaxique — des listes entre crochets imbriquées —, ce qui rend trivial pour un LLM de produire du VVL valide depuis un objectif en langage naturel. Là où en Python ou en Java, faire générer du code par une IA exige des techniques élaborées de validation et de sandbox, en VVL il suffit que le LLM produise une expression syntaxiquement correcte : le parser standard l'accepte, l'évaluateur l'exécute, et le résultat retourne dans le programme appelant comme n'importe quelle autre valeur.

Trois niveaux d'autonomie programmable sont présentés dans ce chapitre. **L'opérateur `@`** (chapitre 8) génère une lambda unique et la rend disponible pour exécution répétée. La commande **`[agent]`** lance une boucle de raisonnement multi-étapes (ReAct : Reason + Act) qui résout un objectif donné. La commande **`[architect]`** opère un cran au-dessus : elle planifie en un seul coup un pipeline complet qui orchestre plusieurs agents et opérateurs de composition. Le chapitre suit la séquence : trois niveaux d'autonomie (22.1), opérateur `@` (22.2), commande `[agent]` (22.3), commande `[architect]` (22.4), patterns de composition (22.5), étude de cas d'analyse de contrat (22.6), homoiconicité comme clé de voûte (22.7), synthèse (22.8).

22.1. De l'utilisateur à l'agent autonome

Le LLM peut intervenir dans un programme VVL avec des degrés d'autonomie croissants. À l'extrémité la plus contrôlée, le développeur écrit un prompt précis et utilise le LLM pour produire **un seul artefact** (une lambda, un JSON, une chaîne) qu'il intègre dans son code. À l'autre extrémité, le développeur exprime un **objectif** et laisse le LLM concevoir, écrire et exécuter le programme entier. Entre les deux, plusieurs niveaux intermédiaires permettent de doser la part d'autonomie selon le contexte.

Niveau	Commande	Autonomie	Cas d'usage
1	@ (opérateur)	Génération d'1 artefact	Filtre, prédicat, transformation simple
2	[agent "goal"]	Boucle ReAct sur 1 objectif	Tâche complexe single-purpose
3	[architect "goal"]	Orchestration multi-agents	Pipeline multi-étapes coordonné

Figure 22.1 — *Trois niveaux d'autonomie programmable du LLM dans VVL*

La progression entre les niveaux n'est pas hiérarchique au sens où le supérieur remplacerait l'inférieur — chacun a son cas d'usage propre. **L'opérateur `@`** convient quand on sait *exactement* ce qu'on veut (« génère-moi un filtre qui garde les emails valides ») mais qu'on veut éviter la fragilité des regex. **La commande `[agent]`** convient pour des tâches dont le développeur ne sait pas formuler la solution algorithmique précise (« calcule le 100ème nombre premier ») mais peut formuler clairement l'objectif. **La commande `[architect]`** convient pour des problèmes complexes décomposables en sous-tâches (« analyse ce contrat et produis un rapport de risque »), où la planification elle-même bénéficie d'être déléguée.

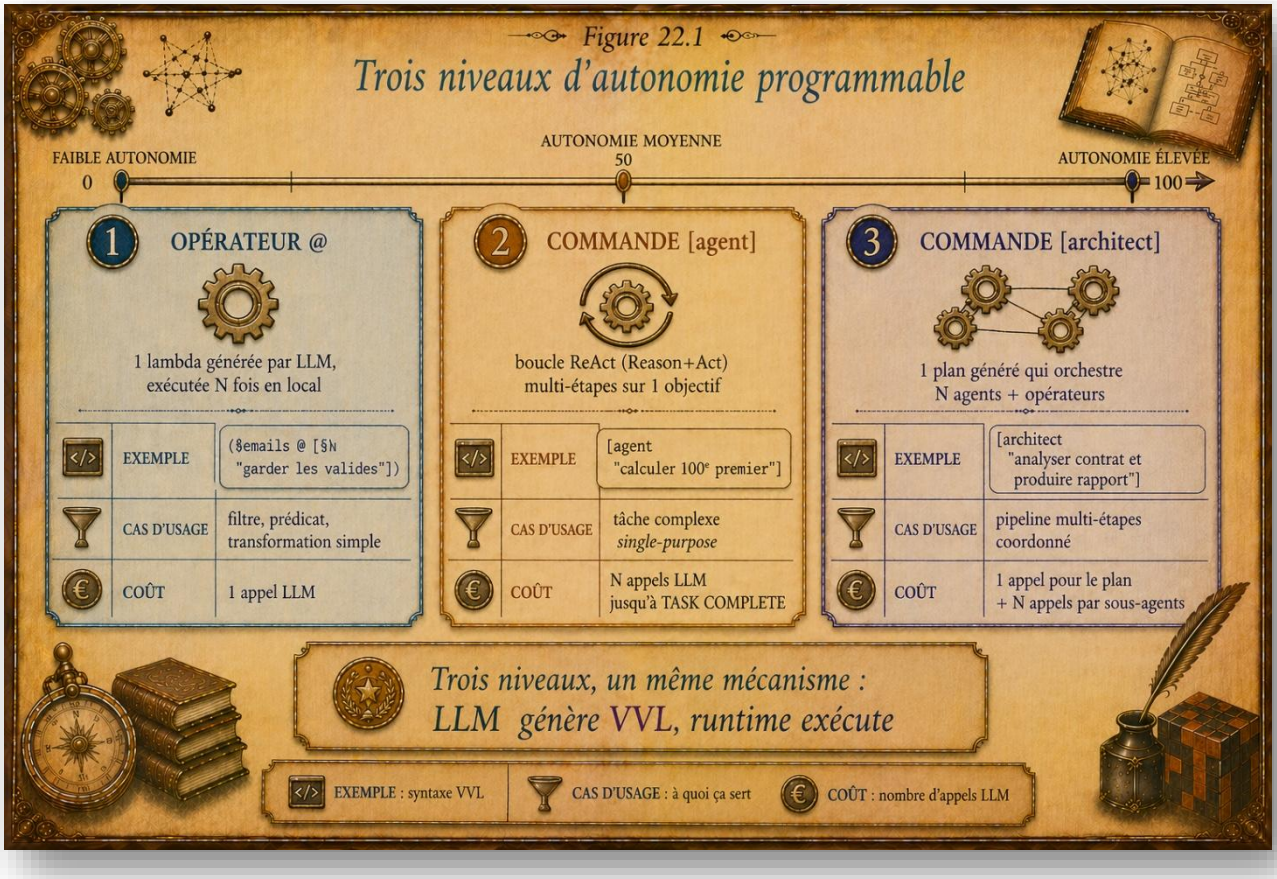


Figure 22.2 — *Spectre d'autonomie : du contrôle fin à la délégation complète*

22.2. Rappel : l'opérateur @ pour la génération ponctuelle

L'opérateur `@` a été présenté en détail au chapitre 8. Il prend à gauche une **donnée d'exemple** et à droite une **description sémantique** `[§ "..."]`, et retourne une **lambda VVL** qui implémente la transformation décrite. La lambda peut ensuite être exécutée localement autant de fois que nécessaire, sans nouvel appel LLM.

```
# 1 appel LLM : generation d'une lambda de filtrage
$exemple = <{"name": "Alice", "email": "alice@example.com"}>
$filtre = ($exemple @ [§ "garde uniquement si l'email est syntaxiquement valide"])

# Le LLM produit par exemple :
# [lambda [item]
#     [if ([strContains $__item.email "@"] && [strContains $__item.email "."])
#         $__item
#         null
#     ]
# ]

# Execution locale : 0 appel LLM par item
$valides = [filter $filtre $contacts]
[log [length $valides] " emails valides sur " [length $contacts]]
```

Cet usage est le **niveau 1 d'autonomie** : le développeur garde un contrôle complet sur le contexte d'exécution, sur ce qui se passe avant et après. Le LLM ne fait que produire l'artefact qu'on lui demande, et ne voit jamais les données réelles (seulement un exemple représentatif). C'est le mode le plus sûr et le plus économique. Pour les besoins fréquents — filtres, validateurs, prédicats —, c'est typiquement le bon choix.

22.3. La commande [agent] : boucle ReAct multi-étapes

La commande `[agent "goal" options?]` représente le niveau 2 d'autonomie. Elle prend un **objectif** en langage naturel et lance une boucle de raisonnement de type **ReAct** (Reason + Act) qui se termine quand l'objectif est atteint ou que le budget d'itérations est épuisé. À chaque itération : le LLM raisonne (Reason), produit un bloc de code VVL (Act), le runtime l'exécute, et le résultat est renvoyé au LLM comme observation pour la prochaine itération.

Syntaxe et options

```
# Forme la plus simple
$result = [agent "calculer la somme des 100 premiers entiers"]

# Avec options
$result = [agent "generer un UUID v4 valide" <{
    "budget": 2,                          # max iterations LLM
    "tools": ["log" "for" "if"],          # builtins autorises (sandbox)
    "model": "gpt-4o"                     # override du modele
}>]
```

Option	Type	Défaut	Description
goal	String	(requis)	Objectif en langage naturel
budget	Integer	8	Nombre maximum d'itérations ReAct
tools	List[String]	[] (tous)	Builtins autorisés (sandboxing)
model	String	cfg.agent_model	Override du modèle LLM

***Figure 22.3 —** Options de la commande [agent]*

Le cycle ReAct interne

L'élément central est la **boucle ReAct**. Elle alterne raisonnement et action jusqu'à atteindre l'objectif. Le pseudo-code suivant explicite le mécanisme — le développeur n'écrit jamais ce code, il est encapsulé dans la commande `[agent]` :

```
# Pseudo-code du fonctionnement interne de [agent]
contexte = snapshot(variables_locales)        # isolation
historique = []
pour i dans 1..budget [do
    # 1. Reason : appel LLM avec historique complet
    reponse = llm_call(systeme_prompt + historique)

    # 2. Extraction du bloc ```vvl ...```
    code = extract_vvl_block(reponse)

    si code est vide :
        si "TASK COMPLETE" dans reponse :
            sortir de la boucle              # objectif atteint
        sinon :
            historique.append("aucun code trouve, reformule")

    # 3. Act : parser + executer le code
    resultat = evaluate(code, contexte)

    # 4. Observe : feedback au LLM
    historique.append(reponse)
    historique.append("RESULTAT: " + resultat)
]
restore(contexte)                             # restauration
retourne dernier_resultat
```

Trois propriétés essentielles découlent de ce mécanisme. **Premièrement, l'isolation par snapshot** : les variables locales du contexte appelant sont sauvegardées avant l'appel et restaurées après, ce qui empêche l'agent de polluer accidentellement l'espace de noms du programme principal. **Deuxièmement, le sandboxing par `tools`** permet de restreindre les builtins disponibles à l'agent : si un agent n'a pas besoin d'accès réseau, on ne lui donne pas `scrape` ou `fetch` ; cette discipline réduit la surface de risque. **Troisièmement, le budget plafonné** garantit qu'un agent en boucle ne consomme pas indéfiniment d'appels LLM — la dégradation gracieuse vaut mieux qu'une explosion de coût.

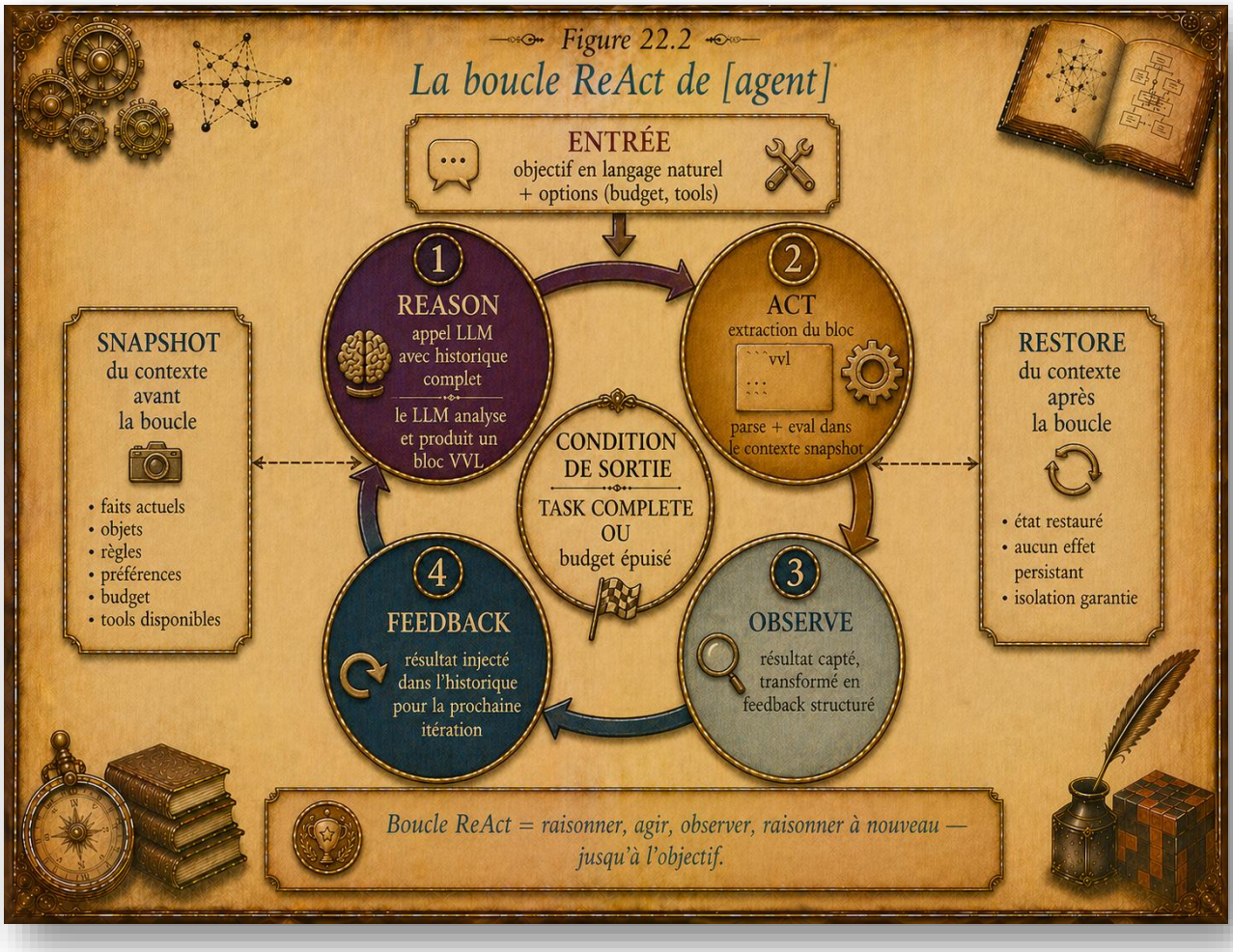


Figure 22.4 — *La boucle ReAct : alternance raisonnement / action / observation*

Exemples d'utilisation

```
# Cas 1 : agent qui calcule
$primes = [agent "retourner la liste des 10 premiers nombres premiers"]
[for $p $primes [log "  -" $p]]

# Cas 2 : agent avec budget reduit pour tache simple
$uuid = [agent "generer un UUID v4 valide" <{"budget": 2}>]

# Cas 3 : agent restreint a un sous-ensemble de builtins
$result = [agent "afficher les carres de 1 a 10"
                 <{"tools": ["log" "for" "range"]}>]
# L'agent ne peut utiliser ni reseau, ni fichiers, ni LLM imbrique.

# Cas 4 : agent dans une boucle pour tache repetitive
$tasks = ["inverser bonjour" "calculer 15!" "generer 3 premiers"]
[for $task $tasks [do
    $res = [agent $__task <{"budget": 3}>]
    [log "Tache: " $__task " => " $res]
]]
```

22.4. La commande [architect] : orchestration single-shot

La commande `[architect "goal"]` représente le niveau 3 d'autonomie. Contrairement à `[agent]` qui raisonne en boucle sur une tâche unique, l'architecte **planifie en un seul appel LLM** un script VVL complet qui coordonne plusieurs sous-tâches. Le script généré peut contenir des appels à `[agent]` pour les parties nécessitant du raisonnement, des opérateurs de parallélisme `|` pour les tâches indépendantes, et du chaînage séquentiel `>$var>` pour les dépendances entre étapes.

L'architecte est conçu autour de **quatre règles de génération** strictes, encodées dans son prompt système. Ces règles garantissent que le script produit a une structure reconnaissable et exploitable :

Règle	Mécanisme	Effet
A. Délégation	[agent "sous-tâche"]	L'architecte ne calcule pas — il délègue aux agents
B. Séquencement	>$var> chaînage	Capture et transmission des résultats intermédiaires
C. Contexte	${var} interpolation	Les sous-agents reçoivent le contexte de l'étape précédente
D. Parallélisme	(tâche1 | tâche2)	Tâches indépendantes lancées simultanément

Figure 22.5 — *Les quatre règles de génération de l'architecte*

```
# Exemple 1 : analyse de documents
[architect "analyser les factures stockees dans le dossier /documents/factures
            et produire un rapport JSON"]

# Exemple 2 : pipeline sequentiel avec capture intermediaire
[architect "scraper la page https://example.com/produits,
            extraire les prix, calculer la moyenne
            et alerter si elle depasse 500 EUR"]

# Exemple 3 : traitement parallele de sources multiples
[architect "charger simultanement ventes.csv et clients.csv,
            les croiser par client_id
            et afficher le total des ventes par client"]

# Exemple 4 : capture du resultat pour usage ulterieur
$rapport = [architect "analyser les logs du jour
                       et produire un resume des erreurs critiques"]
[log $rapport]
[file-write "rapport-quotidien.txt" $rapport]
```

Le bénéfice principal de `[architect]` n'est pas l'économie de code (un développeur expérimenté écrit le pipeline plus rapidement), mais l'**accessibilité** : un utilisateur métier qui ne connaît pas VVL peut décrire son besoin en français et obtenir un système opérationnel. Cela transforme VVL en plate-forme de **scripting par objectif**, où la frontière entre développeur et utilisateur s'estompe pour les tâches courantes.



***Figure 22.6 —** + programme VVL*

La puissance des agents réside dans leur **composabilité avec le reste du langage**. L'agent gère la partie incertaine (compréhension du langage naturel, raisonnement) tandis que le code VVL classique gère la partie déterministe (boucles, filtres, persistance). Cette séparation des responsabilités est la manifestation la plus directe de la philosophie neuro-symbolique : on ne demande pas à l'agent ce qu'on peut écrire en quelques lignes de VVL classique.

Pattern 1 — Agent pour le difficile, VVL pour le déterministe

```
# L'agent gere le NLU complexe ; VVL classique gere le post-traitement
$raw = [agent "extraire les entites nommees du texte suivant :
               'Apple annonce son nouveau produit a Cupertino'"
              <{"budget": 2}>]
$entities = [JSON $raw]                        # parsing classique
[for $e $entities [log "Entite : " $e]]     # boucle VVL classique
```

Pattern 2 — Agent + opérateur @ pour le post-traitement

```
# Agent pour la collecte ; @ pour la transformation
$cities = [agent "retourner une liste JSON de 5 villes francaises
                  avec leur population"]
$fn = ($cities @ [§ "garder les villes avec population > 1 million"])
$big = [filter $fn $cities]
[log "Grandes villes : " $big]
```

Pattern 3 — Agent dans une boucle (automatisation de tâches variées)

```
# Agent applique a une liste de taches heterogenes
$tasks = ["inverser la chaine 'bonjour'"
          "calculer 15 factorielle"
          "generer 3 nombres premiers"]
[for $task $tasks [do
    $res = [agent $__task <{"budget": 3}>]
    [log "Tache: " $__task " => " $res]
]]
```

Pattern 4 — Architect qui délègue et que l'on capture

```
# Le resultat de l'architect est utilisable comme n'importe quelle valeur
$rapport = [architect "construire un rapport hebdomadaire des ventes
                       depuis ventes.csv avec top 5 produits
                       et evolution vs semaine precedente"]

# Post-traitement classique
[file-write "rapport-S15.txt" $rapport]
[email-send "ceo@example.com" "Rapport hebdo S15" $rapport]
```

22.5. Étude de cas : analyse de contrat juridique

Pour illustrer l'orchestration complète par `[architect]`, voici un cas réaliste : un système doit analyser un contrat PDF, extraire les clauses, évaluer le risque juridique de chacune en parallèle, et produire un rapport synthétique. Le développeur exprime l'objectif en français ; l'architecte génère un script qui orchestre quatre sous-tâches.

```
# Appel unique a l'architect
$rapport = [architect "Analyser le contrat /documents/contrat_2026.pdf :
                       1. Extraire les clauses principales
                       2. Identifier les clauses a risque juridique
                       3. Evaluer le score de risque global
                       4. Produire un rapport synthetique JSON"]

# Le script genere par l'architect ressemble a :
# (l'utilisateur ne le voit pas, mais on peut l'inspecter en mode debug)
#
# Etape 1 : extraction du PDF (agent specialise)
# $clauses = [agent "extraire les clauses du contrat /documents/contrat_2026.pdf
#                    sous forme de liste JSON" <{"budget": 3}>]
#
# Etapes 2-3 : evaluation en parallele de chaque clause
# $evaluations = [pmap [lambda [clause] [do
#     $risk = [agent ("evaluer le risque juridique de la clause : "
#                     + $__clause.text) <{"budget": 2}>]
#     <{"clause": $__clause.title, "risk": $risk}>
# ]] $clauses]
```

```
#
# Etape 4 : synthese finale
# $rapport = [agent ("produire un rapport JSON de risque a partir de : "
#                    + [JSON $evaluations]) <{"budget": 2}>]

[log $rapport]
```

Ce script généré convoque **toutes les capacités** présentées dans les chapitres précédents : `[agent]` pour le raisonnement IA, `pmap` pour la concurrence, opérateurs ORC pour la composition, manipulation JSON pour la structuration. Le développeur n'a écrit qu'un objectif en français ; l'architecte a produit le pipeline complet et l'a exécuté.

Trois propriétés rendent ce pattern utile en production. **Première propriété : l'auditabilité**. Le script généré peut être inspecté avant exécution si on active le mode debug — on voit exactement quels agents seront appelés avec quels prompts. **Deuxième propriété : la reproductibilité partielle**. Si un agent intermédiaire échoue, on peut récupérer les résultats des étapes précédentes et reprendre. **Troisième propriété : la performance**. L'architecte exploite naturellement le parallélisme (`pmap` + `|`), ce qu'un humain pourrait oublier de faire pour des étapes indépendantes — l'optimisation parallèle est intégrée à la planification.

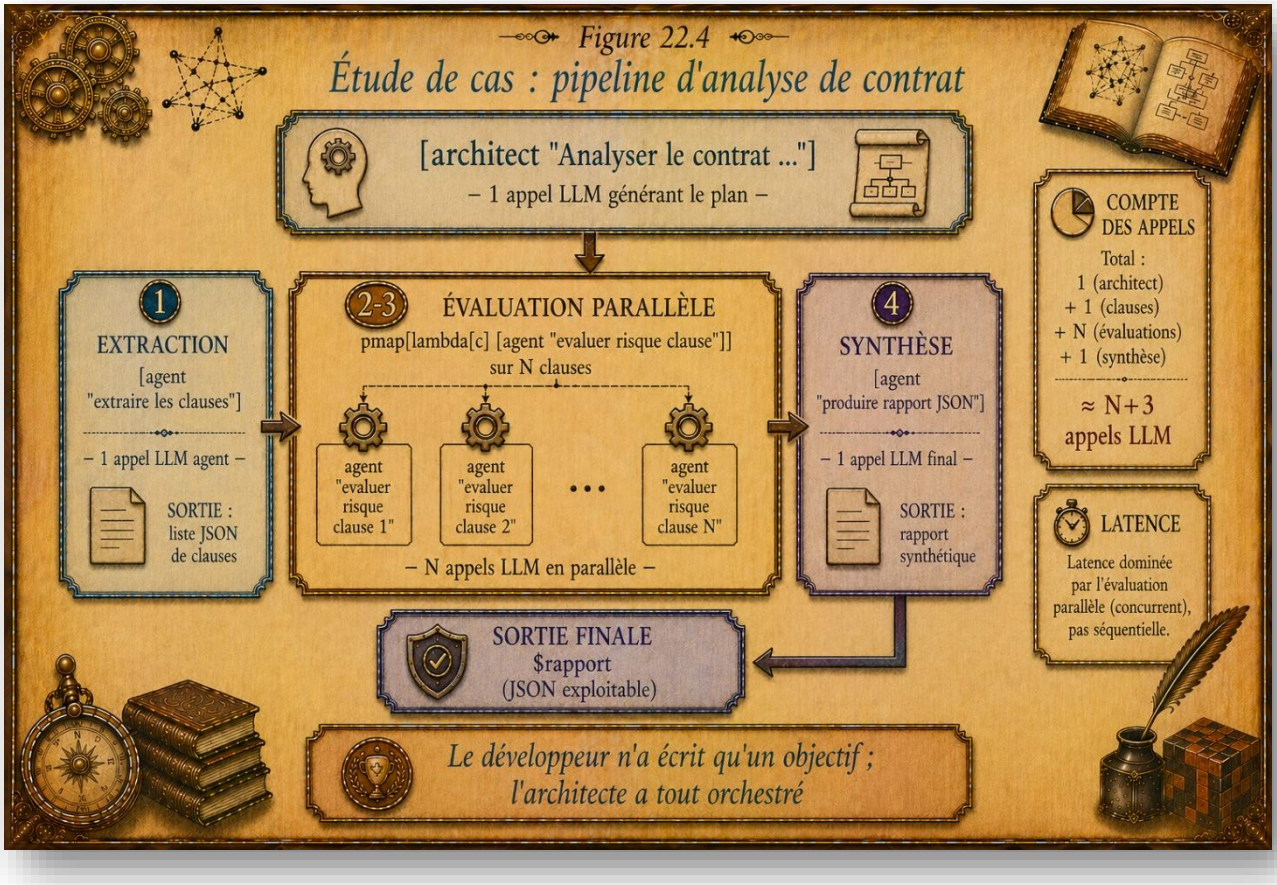


Figure 22.7 — *Pipeline d'analyse de contrat orchestré par l'architecte*

22.6. L'homoiconicité comme clé de voûte

Pourquoi `[agent]` et `[architect]` fonctionnent-ils si naturellement en VVL alors qu'ils restent des défis techniques en Python, Java ou JavaScript ? La réponse tient en un mot : **homoiconicité**. Le code VVL et les données VVL partagent la même représentation syntaxique — des listes entre crochets imbriquées. Cette propriété, présentée au chapitre 4, a deux conséquences majeures pour les agents.

Première conséquence : le LLM produit du VVL valide naturellement. La syntaxe est si simple — `[commande arg1 arg2 ...]` — qu'elle ne souffre presque pas d'erreurs de génération. En Python, le LLM doit gérer l'indentation significative, les exceptions de syntaxe (deux-points obligatoires, parenthèses, virgules), les imports qui dépendent du contexte. En VVL, il n'y a qu'une règle : tout est entre crochets. Le taux de syntaxe valide produit par le LLM est de l'ordre de 99 %, contre typiquement 80-90 % pour Python.

Deuxième conséquence : l'évaluation est triviale et sûre. Le code généré est parsé par le parser standard du langage, sans `eval()` dynamique, sans sandbox isolé. C'est l'**inverse** du pattern Python où `eval(llm_output)` est dangereux : il peut exécuter n'importe quoi, y compris importer des modules sensibles ou accéder au système de fichiers. En VVL, l'option `tools` de `[agent]` permet de **restreindre** les builtins exposés au LLM, ce qui crée un sandbox déclaratif sans complexité d'implémentation.

Les langages historiques qui ont cette propriété — Lisp, Scheme, Tcl — ont longtemps été des candidats théoriques pour la programmation par IA, mais leur écosystème industriel limité les a écartés. VVL hérite de leurs propriétés syntaxiques tout en bénéficiant d'un runtime moderne (LLM intégré, ontologie, RAG, concurrence native), ce qui en fait le **premier langage industriellement viable** pour la programmation par agents autonomes. Ce n'est pas une coïncidence : la décision originelle de VAILS d'adopter une syntaxe Lisp-like a été prise en anticipation de cet usage.

22.7. Synthèse : trois niveaux pour une autonomie programmable

Avec ce chapitre, le tableau de la programmation neuro-symbolique en VVL est complet. Aux **outils symboliques** (Prolog, ontologie, opérateurs Zalta) et aux **outils neuronaux** (RAG, extraction, Gherkin, induction) s'ajoutent désormais les **outils d'autonomie** : trois niveaux de délégation au LLM, choisis selon la complexité du besoin.

Critère	@ (opérateur)	[agent]	[architect]
Appels LLM	1 (gen. lambda)	N (boucle ReAct)	1 (plan) + N (sous-agents)
Contrôle du dev.	Élevé	Moyen (objectif fixe)	Faible (délégation totale)
Cas d'usage	Filtre, prédicat	Tâche complexe unique	Pipeline multi-étapes
Coût typique	Faible	Moyen	Élevé
Auditabilité	Lambda inspectable	Trace ReAct	Script généré inspectable
Sandboxing	Aucun (lambda pure)	Option tools	Hérité par les sous-agents

Figure 22.8 — *Synthèse comparative des trois niveaux d'autonomie*

Ce tableau aide à choisir le bon outil pour chaque contexte. Pour des **transformations simples** récurrentes, l'opérateur `@` est imbattable en coût et en contrôle. Pour des **tâches complexes ponctuelles**, `[agent]` apporte la puissance d'un raisonnement multi-étapes. Pour des **projets entiers** où le développeur peut formuler un objectif clair sans vouloir écrire le pipeline, `[architect]` permet

une délégation à très haute valeur ajoutée. Les trois cohabitent dans le même langage et peuvent être combinés librement.

Le **chapitre 23** poursuit en dotant le runtime d'une taxonomie unifiée : le protocole d'atomes, qui attribue à toute valeur — extraite, calculée ou générée par LLM — une famille ontologique, huit axes de métadonnées et des capabilities composables. C'est la couche qui rend auditable ce que les agents de ce chapitre produisent, et tous les modules présentés ensuite s'y adossent.

23. Le protocole d'atomes : taxonomie unifiée du runtime

Mots-clés : protocole d'atomes, SymbolicAtom, GroundedAtom, ReifiedAtom, axes métadonnées, capabilities composables, introspection structurée, policy déclarative, sandbox multi-tenant, audit trail, content-hash, semantic_id, identité à trois niveaux

Le chapitre 22 a ouvert la Partie VII en montrant comment l'opérateur @, la commande [agent] et la commande [architect] permettent de déléguer au LLM des fragments de programme entiers. Deux mécanismes transversaux y ont joué un rôle critique sans recevoir de traitement systématique : le sandboxing par tools, qui restreint les builtins disponibles à un agent, et l'auditabilité des décisions prises par les sous-systèmes neuro-symboliques. Ce chapitre 23 reprend ces deux mécanismes pour les fonder sur une couche d'abstraction commune.

Le protocole d'atomes ne change rien à la sémantique du langage telle qu'elle a été présentée dans les chapitres précédents. Il ajoute une **typologie formelle** des entités qui circulent dans le runtime VVL — instances ontologiques, concepts réifiés, images, PDF, sémaphores, méthodes liées, connexions WebDriver — et **expose cette typologie** au programme via six builtins d'introspection et de policy. Ce qui était jusqu'ici implicite dans le code interne (savoir si une valeur est manipulable symboliquement, si elle porte un effet de bord, si elle dépend d'un appel LLM) devient calculable depuis VVL lui-même. Toute la machinerie transversale — audit trail, sandbox, replay, cost tracking — peut alors s'écrire **sans** `isinstance` ad-hoc, par dispatch sur des axes sémantiques.

Le chapitre suit la séquence : motivation et constat empirique (23.1), les trois familles ontologiques (23.2), les huit axes métadonnées (23.3), les capabilities composables (23.4), les trois niveaux d'identité orthogonaux (23.5), les six builtins d'introspection (23.6), le DSL de policy par axes (23.7), cas d'usage opérationnels (23.8), synthèse et statut (23.9).

> **Périmètre. Le protocole d'atomes décrit dans ce chapitre fait partie du serveur courant. Sa sémantique — familles ontologiques, axes de métadonnées, capabilities composables — est couverte par une suite de 75 tests, et les quatre tutoriels vvl/atoms/01-atoms-and-axes.vvl à 04-use-cases.vvl sont exécutables tels quels. Le protocole est additif et opt-in : aucun programme VVL antérieur n'est affecté tant qu'il n'est pas explicitement utilisé. Les chapitres suivants s'y adossent — les modules génétique, ILP et cluster en font leur socle d'introspection et de traçabilité.**

23.1. Pourquoi un protocole transversal

À mesure que les chapitres précédents ont déroulé les briques du langage, le runtime VVL a accumulé des **types** d'objets dont la diversité dépasse maintenant la quinzaine. La Partie II a introduit `VVL_Instance` (chapitre 6 — système OO) et `VVL_BoundMethod` (méthodes dispatchées). La Partie III a ajouté `VVL_ConceptEntity` (chapitre 9 — réification d'un concept par LLM). La Partie IV a apporté `VVL_OntologyInstance` et `VVL_AbstractObject` (chapitre 12 — ontologie typée). Les chapitres applicatifs ont fait apparaître `VVL_Img`, `VVL_Pdf`, `VVL_WebDriverProxy`, `VVL_OfficeProxy`,

`VVL_VDOM_Connection`, `VVL_WebElementProxy`, `VVL_Semaphore`, `VVL_Channel` — autant de types portant chacun un payload natif différent.

Tous ces types **partagent un parent commun** dans le code interne : `VVLNode`, l'interface des nœuds évaluables. Mais au-delà de cet ancêtre minimal, il n'existait jusqu'ici aucune classification formelle. Les questions évidentes que se pose un programme à propos d'un objet — *est-ce manipulable symboliquement ? est-ce que cela porte un effet de bord ? est-ce déterministe ? quelle est l'origine de cette donnée ?* — n'avaient pas de réponse uniforme. Le code du runtime payait cette absence par des `isinstance(obj, VVL_Img)`, `isinstance(obj, VVL_BoundMethod)`, `isinstance(obj, VVL_ConceptEntity)` dispersés sur des dizaines de fichiers. Une mesure rapide a recensé quelque **94 occurrences** réparties sur 31 fichiers du runtime.

Le coût n'est pas que cosmétique. Trois conséquences pratiques en découlent. **Premièrement, la fermeture au monde** : ajouter un nouveau type d'atome (par exemple `VVL_GeoPoint` pour les calculs géographiques du chapitre 25) oblige à éditer tous les sites qui dispatchent sur le type, sous peine de bugs silencieux. **Deuxièmement, l'opacité du discours neuro-symbolique** : le livre parle constamment d'« oscillation neuro-symbolique », de « grounding », de « pont symbolique », mais ces concepts ne sont pas calculables — un développeur ne peut pas demander au runtime *« est-ce que cet objet vit côté symbolique ou côté neuronal ? »*. **Troisièmement, l'absence d'une politique transversale** : les efforts de sandbox (chapitre 22, option `tools`), de cache (chapitre 9, mémoïsation des concepts), d'auditabilité (cas industriels du médical et du BTP) sont chacun implémentés isolément, sans interface commune.

Le protocole d'atomes répond à ces trois manques par une couche d'abstraction *au-dessus* de `VVLNode` qui (a) **nomme** trois familles ontologiques, (b) **attache** à chaque atome une fiche signalétique structurée de huit axes, et (c) **expose** des capabilities composables que le programme peut interroger. Une fois ce contrat en place, audit trail, sandbox, replay et cost tracking s'écrivent uniformément, et l'oscillation neuro-symbolique devient *calculable* — pas seulement *narrée*.

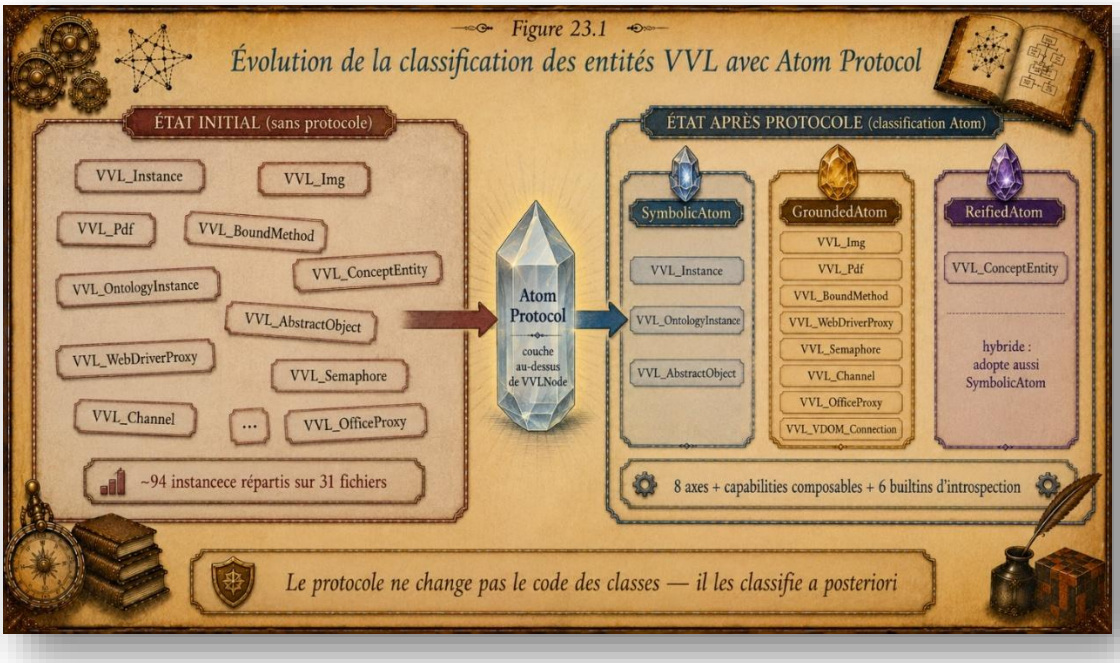


Figure 23.1 — *Le runtime VVL avant et après le protocole d'atomes : du dispatch implicite à la typologie déclarée*

23.2. Trois familles ontologiques mutuellement exclusives

Le protocole nomme **trois familles** d'atomes, chacune définie par un `typing.Protocol` distinct. L'adoption est rétroactive — aucune classe existante ne change d'arbre d'héritage — et obtenue en exposant les méthodes de marquage prévues par chaque protocole. Cette discipline de Protocol explicite garantit que le système OO du chapitre 6 reste intact : pas de hiérarchie parallèle, pas de conflit de MRO, juste un contrat d'interface.

Famille	Sémantique	Adoptants natifs
`SymbolicAtom`	Vit dans le formalisme symbolique. Manipulable par unification, dérivation, pattern matching. Projetable directement dans Prolog.	`VVL_Instance` (ch. 6), `VVL_OntologyInstance` (ch. 12), `VVL_AbstractObject` (ch. 12), futur `VVL_RuleAtom`
`GroundedAtom`	Porte un payload natif **opaque** vis-à-vis du moteur de réécriture. Un sous-kind (`payload_kind`) précise la nature : `value`, `callable`, `connection`, `reference`, `resource`.	`VVL_Img`, `VVL_Pdf`, `VVL_BoundMethod`, `VVL_WebDriverProxy`, `VVL_VDOM_Connection`, `VVL_OfficeProxy`, `VVL_WebElementProxy`, `VVL_Semaphore`, `VVL_Channel`
`ReifiedAtom`	Symbole avec **oracle de complétion sémantique**. Distinct de Grounded parce que son payload n'est pas un objet binaire mais un appel de modèle paramétrique, dont la réponse peut différer d'un appel à l'autre.	`VVL_ConceptEntity` (ch. 9), futurs concepts LLM-backed

Figure 23.2 — *Les trois familles ontologiques et leurs adoptants natifs dans VVL 5.7*

La distinction entre **GroundedAtom** et **ReifiedAtom** mérite un mot d'explication, parce qu'elle ne va pas de soi à la première lecture. Un `VVL_Img` et un `VVL_ConceptEntity` partagent un trait : tous deux portent quelque chose qui n'est pas un terme symbolique pur. Pourquoi alors ne pas les ranger dans la même famille ? La réponse est que leur **mode de payload** est radicalement différent. L'image porte des octets — déterministes, hashable, réplicables ; le concept réifié porte un *pointeur vers un modèle paramétrique* — non-déterministe, dépendant de la session LLM, susceptible d'enrichissement progressif. Les confondre reviendrait à confondre un fichier sur disque avec une entrée d'oracle Delphique : les deux donnent des réponses, mais l'un les *contient* tandis que l'autre les *invoque*. Cette distinction prend toute sa valeur quand on construit des politiques de sandbox — on autorise volontiers les Img, beaucoup moins volontiers les appels LLM dans un hot path critique.

Le cas hybride : VVL_ConceptEntity

Un cas mérite un traitement particulier : `VVL_ConceptEntity`, le type produit par la commande `[@"..."]` du chapitre 9. Il **adopte simultanément** SymbolicAtom (parce qu'il hérite de `VVL_Instance` et qu'il dispose d'un `prolog_id` projetable dans la KB) et ReifiedAtom (parce que son enrichissement passe par des appels LLM). Cette double appartenance n'est pas une bizarrerie — c'est exactement ce

que dit la définition zaltaïenne du concept réifié au chapitre 9 : **un symbole avec un oracle**. Le protocole formalise donc une vérité conceptuelle déjà présente dans le livre.

Cette propriété hybride permet à un même `VVL_ConceptEntity` d'être traité tantôt comme symbole pur (quand on l'inscrit dans la KB Prolog via `obj-assert`), tantôt comme oracle (quand on accède à un champ encore non découvert et que le lazy-load déclenche un appel LLM). Le protocole l'autorise sans heurts, et les builtins d'introspection (section 23.6) le révéleront explicitement.

23.3. Les huit axes métadonnées

Au-delà de la famille, chaque atome déclare une **fiche signalétique** de huit axes — des `ClassVar` typés sur les classes adoptantes, sérialisables JSON directement. Ces axes ne sont pas des propriétés dynamiques (leur valeur est fixée à la définition de classe et ne change pas au cours de la vie de l'objet) ; ce sont des **étiquettes de catégorie** qui permettent d'écrire des politiques sans connaître chaque type individuellement.

Axe	Valeurs	Sens
`payload_kind`	`value`, `callable`, `connection`, `reference`, `resource`	Pour `GroundedAtom` uniquement : la nature du payload natif.
`effect_kind`	`pure`, `read`, `write`, `external`	Que produit l'évaluation ? `pure` = aucun effet ; `read` = consultation d'état ; `write` = modification d'état local ; `external` = communication réseau, fichier, périphérique.
`determinism`	`deterministic`, `stochastic`, `session_dependent`	Deux évaluations successives produisent-elles le même résultat ?
`persistence`	`serializable`, `rebuildable`, `ephemeral`	L'atome peut-il survivre à un snapshot/restore ?
`trust_level`	`trusted`, `tested`, `untested`	Quel est le degré de confiance dans le contenu ?
`provenance`	`user_input`, `schema_definition`, `extraction`, `induction`, `llm_call`, `computation`, `external_api`	D'où vient cet atome ?
`cost_hint`	float (USD ou unité interne)	Coût attendu d'évaluation.
`latency_hint`	float (ms)	Latence attendue d'évaluation.

Figure 23.3 — *Les huit axes métadonnées attachés à chaque atome*

Un exemple condensé suffit pour fixer les idées. Une `VVL_Img` (image décodée en mémoire, manipulable par les opérateurs `/` et `%` du chapitre 15) déclare typiquement `payload_kind=value`, `effect_kind=pure`, `determinism=deterministic`, `persistence=serializable`, `trust_level=trusted`, `provenance=extraction`, `cost_hint=0.0`, `latency_hint=0.0`. Un `VVL_Semaphore` déclare `payload_kind=resource`, `effect_kind=write`,

`determinism=session_dependent`, `persistence=ephemeral`. Un `VVL_ConceptEntity` issu de `[@ "Steel316L"]` déclare `effect_kind=pure` (il n'altère rien à l'extérieur), mais `determinism=stochastic` et `provenance=llm_call` (sa découverte progressive dépend du modèle paramétrique).

Cette uniformité a une conséquence concrète : **on peut raisonner sur n'importe quel atome sans connaître son type précis**. Une politique qui filtre tout ce dont `effect_kind` vaut `write` refuse aussi bien un sémaphore qu'un futur `VVL_FileHandle`, sans nécessiter de mise à jour. C'est la promesse de l'extensibilité par axes sémantiques plutôt que par dispatch nominal.

23.4. Capabilities composables (anti fat-interface)

Les axes décrivent **ce qu'est** un atome ; les capabilities décrivent **ce qu'on peut en faire**. Chaque atome adopte uniquement les capabilities qui le concernent — c'est l'anti-pattern *fat interface* explicitement évité.

Capability	Méthode	Adoptants typiques
`Introspectable`	`runtime_id() -> str`	Universel (default `str(id(self))`)
`Identifiable`	`content_hash() -> str`	`VVL_Img`, `VVL_Pdf`, `VVL_AbstractObject`, `VVL_ConceptEntity`
`SemanticallyIdentifiable`	`semantic_id(ctx) -> str`	`VVL_ConceptEntity` (converging)
`Featurizable`	`encode_features(ctx) -> dict`	`VVL_Img`, `VVL_Pdf` (matching contre les abstracts Zalta du ch. 13)
`LLMRepresentable`	`to_llm_payload(ctx) -> str|dict|list`	`VVL_Img`, `VVL_Pdf` — utilisé par les opérateurs sémantiques du ch. 7
`Persistable`	`snapshot()` + `restore(snap)`	Contracted, pas implémenté en runtime VVL 5.7
`Callable`	`call(ctx, args) -> VVLNode`	`VVL_BoundMethod` (déjà présent avant le protocole)
`Traceable`	`trace(ctx) -> dict`	`VVL_Approximation`, `VVL_ConceptEntity`, `VVL_Policy`

Figure 23.4 — *Les huit capabilities composables et leurs adoptants*

L'intérêt de la composition apparaît quand on regarde **ce qu'un atome n'a pas**. Un `VVL_Semaphore` n'adopte que `Introspectable` : il n'y a rien à featurizer dans un primitif de synchronisation, rien à représenter pour le LLM, et son `runtime_id` est sa seule identité utile (deux sémaphores créés indépendamment, même avec les mêmes paramètres, sont deux ressources distinctes). À l'inverse, un `VVL_ConceptEntity` adopte `Introspectable`, `Identifiable`, `SemanticallyIdentifiable` *et* `Traceable` — sa richesse sémantique est précisément ce qui rend l'auditabilité de la section 23.8 possible.

Cette section éclaire rétroactivement plusieurs mécanismes des chapitres précédents. Le matching Zalta hybride à 4 niveaux (chapitre 13.4) entre une instance grounded — typiquement une `VVL_Img` issue d'extraction — et un `VVL_AbstractObject` repose sur la capability `Featurizable` : c'est la

méthode `encode_features` qui produit le dictionnaire de features que l'opérateur `~~` confronte aux encodages de l'archétype. De même, l'inclusion d'une image dans un prompt LLM via les opérateurs sémantiques du chapitre 7 transite par `to_llm_payload`. Le protocole donne donc des noms à des contrats déjà honorés ; ce qui change, c'est que ces contrats deviennent désormais **interrogeables depuis VVL** (section 23.6).

23.5. Trois niveaux d'identité orthogonaux

Le livre a soulevé à plusieurs reprises la question de l'**identité d'un objet** sans la traiter formellement. Au chapitre 9, deux concepts du même nom dans des contextes différents sont *deux objets abstraits différents* (désambiguïsation contextuelle). Au chapitre 13, l'opérateur d'identité intensionnelle compare deux ontologies par leur contenu encodé. Au chapitre 6, deux instances de la même classe sont distinguées par leur identité Python. Ces trois usages parlent d'identité, mais d'identités **différentes**. Le protocole d'atomes les nomme et les rend orthogonales.

`runtime_id()` identifie le **handle Python vivant**. C'est l'identité qui change à chaque création d'objet, même quand le contenu est identique. Pour un `VVL_WebDriverProxy` (connexion Selenium au navigateur), c'est tout ce qui a un sens — deux connexions ouvertes simultanément vers le même site sont deux ressources distinctes. C'est aussi le fallback universel : tout atome adopte `Introspectable`, donc tout atome a un `runtime_id`.

`content_hash()` identifie le **contenu binaire ou intensionnel**. Pour une `VVL_Pdf`, c'est le SHA-256 des octets du fichier — deux instances Python distinctes (`runtime_id` ≠) chargées du même fichier ont le même `content_hash`. Pour un `VVL_AbstractObject`, c'est le `prolog_id` Zalta calculé à partir de l'ensemble des `enc(_, _, _)` qui le définissent — deux abstracts construits avec les mêmes encodages partagent leur `content_hash`. C'est l'identité que l'on veut pour la **déduplication**, le **caching content-based**, et la **cross-validation de pipelines**.

`semantic_id(ctx)` identifie une **équivalence approximative ou modèle-dépendante**. Pour un `VVL_ConceptEntity`, c'est probabiliste : deux concepts du même nom dans des contextes différents convergent vers des `semantic_id` voisins à mesure que le LLM enrichit leurs fields, mais ne s'identifient jamais strictement. Cette identité est volontairement *fuzzy* — elle modélise le fait que deux concepts « sont à peu près le même » dans un contexte donné, ce qui suffit pour de l'agrégation, du clustering, ou de la suggestion, mais pas pour de l'égalité forte.

Pour une même `VVL_Img`, les trois identités peuvent diverger. Deux instances Python distinctes (runtime ≠) du même fichier (content =) qui sont visuellement quasi-identiques (semantic ≈) coexistent sans contradiction — chaque niveau capture une dimension différente. Le programme choisit le niveau pertinent selon la question qu'il pose.



Figure 23.5 — *runtime_id, content_hash, semantic_id : trois questions, trois identités*

23.6. Builtins d'introspection

Le protocole expose côté VVL **quatre builtins d'introspection** qui permettent à un programme d'interroger n'importe quel atome sans connaître son type Python. Tous ont une signature uniforme — un argument, un retour structuré — et un comportement gracieux sur les non-atomes (un entier, par exemple, retourne une trace minimale plutôt qu'une erreur).

[atom:family $x]

Retourne la famille comme `VVL_String` : `"SymbolicAtom"`, `"GroundedAtom"`, `"ReifiedAtom"`, ou la chaîne vide si l'atome n'adopte aucun protocole de famille. Pour les atomes hybrides (le cas `VVL_ConceptEntity` de la section 23.2), la famille la plus spécifique gagne, avec la priorité **Reified > Grounded > Symbolic**.

```
[atom:family [createImage 32 32]]                  # → "GroundedAtom"
[atom:family [semaphore 1]]                        # → "GroundedAtom"
[atom:family [@ "Paris"]]                          # → "ReifiedAtom"  (hybride)
[atom:family [ontology:abstract "X" [encoded [k "v"]]]]
                                                   # → "SymbolicAtom"
[atom:family 42]                                   # → ""             (pas un atome)
```

[atom:axes $x]

Retourne un dict JSON des huit axes métadonnées — pratique pour inspecter rapidement le profil effet/coût/déterminisme sans parser une trace complète. L'axe `payload_kind` n'est présent que pour les `GroundedAtom`.

```
[atom:axes [createImage 64 64]]
# → {payload_kind: "value", effect_kind: "pure",
#    determinism: "deterministic", persistence: "serializable",
#    trust_level: "trusted", provenance: "extraction",
#    cost_hint: 0.0, latency_hint: 0.0}

[atom:axes [@ "Steel316L"]]
# → {effect_kind: "pure", determinism: "stochastic",
#    provenance: "llm_call", ...}     # pas de payload_kind (pas Grounded)
```

[atom:capabilities $x]

Retourne la liste des capabilities effectivement adoptées par l'atome. C'est l'outil qui permet à un programme défensif de **vérifier avant d'appeler** une méthode optionnelle — par exemple, ne tenter `to_llm_payload` que si la capability `LLMRepresentable` est listée.

```
[atom:capabilities [createImage 32 32]]
# → ["Introspectable" "Identifiable" "Featurizable" "LLMRepresentable"]

[atom:capabilities [semaphore 1]]
# → ["Introspectable"]      # anti-fat-interface : rien d'autre n'est utile

[atom:capabilities [@ "Paris"]]
# → ["Introspectable" "Identifiable" "SemanticallyIdentifiable" "Traceable"]
```

[atom:trace $x]

Le builtin le plus riche. Il retourne un dict JSON complet : famille, classe Python, identités disponibles, axes, dump textuel, et — si l'atome adopte `Traceable` — la dérivation interne (assumptions, fields LLM-discoverés avec leur source, etc.). C'est la primitive sur laquelle l'audit trail de la section 23.8 est construit.

```
[atom:trace [@ "Steel316L"]]
# → {family: "ReifiedAtom", class: "VVL_ConceptEntity",
#    runtime_id: "...", content_hash: "concept_093693ebcf7e",
#    semantic_id: "sem_4ba82a78bd25",
#    axes: {provenance: "llm_call", determinism: "stochastic", ...},
#    n_discovered: 10,
#    discovered: [
#      {name: "material_type", value: "austenitic stainless steel", source: "llm_call"},
#      {name: "density_g_cm3", value: "8", source: "llm_call"},
#      ...
#    ],
#    concept_name: "Steel316L", prolog_id: "concept_093693ebcf7e"}
```

Pour les non-atomes (entiers, chaînes, listes ordinaires), `[atom:trace]` retourne au moins les champs `class` et `dump` — fallback gracieux qui permet d'écrire du code générique sans pré-vérification.

23.7. Le DSL de policy par axes

Les chapitres précédents ont rencontré le problème du **sandboxing déclaratif** sans pouvoir lui donner de réponse complète. Au chapitre 22, la commande `[agent]` accepte une option `tools` qui restreint les builtins disponibles à une liste blanche. Cette approche fonctionne mais souffre de deux limites : elle opère par *nom* (il faut connaître chaque builtin individuellement), et elle est *fragile à l'évolution* (un nouveau builtin sensible n'est pas couvert tant qu'on n'a pas mis à jour les listes blanches existantes).

Le protocole introduit deux builtins — `[policy:define]` et `[policy:check]` — qui opèrent sur les **axes sémantiques** plutôt que sur les noms. Une politique exprime des règles allow/deny sur les axes (`effect_kind`, `determinism`, `provenance`, `cost_hint`, `family`, `capabilities`) ; le verdict est calculé en walkant les règles dans l'ordre, avec la sémantique **first-match wins** et un comportement par défaut **deny** (safe-fail).

[policy:define]

```
[policy:define
   [list ACTION AXIS [OP] VALUES…]
   …
]
```

Chaque règle est une `VVL_List` avec :

ACTION ∈ `"allow"` | `"deny"`
AXIS : un des huit axes ou les pseudo-axes `"family"` / `"capabilities"`
OP : optionnel, défaut `"in"`. Valeurs : `"in"`, `"=="`, `"!="`, `">"`, `"<"`, `">="`, `"<="`
VALUES : une ou plusieurs valeurs comparées

Quatre politiques-types couvrent l'essentiel des besoins industriels :

```
# Reader-only : pas de write, pas d'external, accept Symbolic + Grounded
$reader = [policy:define
   [list "deny"  "effect_kind" "==" "write"]
   [list "deny"  "effect_kind" "==" "external"]
   [list "allow" "family"      "in" "SymbolicAtom" "GroundedAtom"]
]

# Cost-cap : refuse les atomes coûteux (par appel LLM cher, par exemple)
$cheap = [policy:define
   [list "deny"  "cost_hint" ">"  "0.10"]
   [list "allow" "family"    "in" "SymbolicAtom" "GroundedAtom" "ReifiedAtom"]
]

# No-LLM : pas d'appels paramétriques (pour reproductibilité)
$no_llm = [policy:define
   [list "deny"  "provenance" "==" "llm_call"]
   [list "allow" "family"     "in" "SymbolicAtom" "GroundedAtom" "ReifiedAtom"]
]

# Prod-safe : déterminisme requis (hot path critique)
$prod_safe = [policy:define
   [list "deny"  "determinism" "==" "stochastic"]
   [list "allow" "family"      "in" "SymbolicAtom" "GroundedAtom" "ReifiedAtom"]
]
```

[policy:check $policy $atom]

Évalue l'atome contre la politique et retourne un **verdict structuré** qui n'est pas un simple booléen — c'est un dict JSON traçable contenant la règle qui a matché et la raison du verdict.

```
$verdict = [policy:check $reader [createImage 32 32]]
# → {allowed: true, matched_rule: 1, action: "allow",
#    axis: "family", op: "in", values: ["SymbolicAtom", "GroundedAtom"],
#    reason: "rule #1 (allow): family in ['SymbolicAtom', 'GroundedAtom']"}

[policy:check $reader [semaphore 1]]
# → {allowed: false, matched_rule: 0, action: "deny",
#    axis: "effect_kind", op: "==", values: ["write"],
#    reason: "rule #0 (deny): effect_kind == ['write']"}

# Default-deny : un atome qui ne matche aucune règle est refusé
$strict = [policy:define [list "allow" "family" "==" "SymbolicAtom"]]
[policy:check $strict [createImage 32 32]]
# → {allowed: false, matched_rule: null, action: "default-deny",
#    reason: "No rule matched (default: deny)"}
```

Cette traçabilité est essentielle dans les contextes audités : un refus n'est jamais opaque, il pointe vers la règle responsable. C'est ce qui permet de répondre à *« pourquoi cet agent a-t-il été bloqué ? »* sans replonger dans les logs internes du runtime.

Critère	Sandbox tools (ch. 22)	Policy par axes (ch. 23)
Granularité	Builtin individuel par nom	Catégorie sémantique par axe

Maintenance	À étendre à chaque nouveau builtin sensible	Stable — couvre les types futurs par défaut
Verdict	Booléen implicite (le builtin fonctionne ou pas)	Dict JSON traçable avec règle responsable
Comportement par défaut	Tout autorisé sauf liste noire	Tout refusé sauf règle allow explicite (safe-fail)
Cas d'usage	Restreindre rapidement un agent à un sous-ensemble connu	Définir un profil de sécurité auditable, multi-tenant

Figure 23.6 — *Sandbox par tools vs policy par axes : deux mécanismes complémentaires*

Les deux mécanismes ne s'excluent pas — ils opèrent à des niveaux différents et se composent. Une configuration prudente d'agent en production combine les deux : `tools` pour interdire les builtins manifestement hors-périmètre (réseau, fichiers), et une policy pour refuser tout atome dont les axes ne respectent pas le contrat de sécurité du tenant.

23.8. Cas d'usage opérationnels

Quatre cas d'usage suffisent à montrer que le protocole transforme des problèmes jusqu'ici implémentés *ad hoc* en patterns uniformes et calculables.

Cas 1 — Audit trail GDPR pour un concept réifié

Un système RH (cas industriel proche du chapitre 13) extrait des candidats depuis des CV, en construisant des `VVL_ConceptEntity` ou des instances ontologiques typées. Une obligation GDPR exige de pouvoir produire, pour chaque décision automatisée, la chaîne complète de propriétés et leurs sources.

```
# 1. Le pipeline RH produit un concept enrichi par LLM
$candidat = [@ "Alice DUPONT - 9 ans Python, lead dev"]

# 2. Lors d'un audit, on demande la trace complète
$audit = [atom:trace $candidat]
[log $audit.discovered]
# → [
#     {name: "experience_years", value: 9, source: "llm_call", ts: "2026-04-02T..."},
#     {name: "primary_language", value: "python", source: "llm_call", ts: ...},
#     {name: "leadership", value: true, source: "induction:rule_42", ts: ...},
#     ...
#   ]

# 3. L'auditeur peut filtrer par source pour vérifier les chaînes critiques
$llm_only = [filter [lambda [d] ($__d.source == "llm_call")] $audit.discovered]
```

Ce qui était précédemment caché dans les logs internes du runtime devient une **donnée structurée** que l'auditeur consulte sans connaître l'implémentation. Le champ `source` distingue les propriétés issues d'extraction LLM, d'induction de règles (chapitre 17), de saisie utilisateur ou de calcul déterministe — la chaîne de causalité requise par toute compliance algorithmique.

Cas 2 — Sandbox multi-tenant déclarative

Un service SaaS qui héberge plusieurs clients sur la même instance VAILS doit garantir qu'un agent appartenant au tenant A ne peut pas accéder aux ressources du tenant B, ni produire d'effets durables hors de son workspace. La policy par axes permet de poser ce contrat en quelques lignes :

```
$tenant_a = [policy:define
   [list "deny"  "effect_kind"  "in" "external"]
   [list "deny"  "provenance"   "==" "external_api"]
   [list "deny"  "cost_hint"    ">"  "0.05"]                  # plafond par opération
   [list "allow" "family"       "in" "SymbolicAtom" "GroundedAtom" "ReifiedAtom"]
]

# Avant de soumettre une tâche à l'agent, on vérifie qu'elle est conforme
$task = [createImage 100 100]
$verdict = [policy:check $tenant_a $task]

[if $verdict.allowed
   [agent:run $task]
   [log [+ "Task denied: " $verdict.reason]]
]
```

L'avantage par rapport à la sandbox `tools` est que **les nouveaux types d'atomes sont denied par défaut** (safe-fail). Si Anthropic ajoute demain un `VVL_GeoPoint` qui appelle un service externe pour résoudre des coordonnées, ce nouvel atome aura `effect_kind=external` et sera automatiquement bloqué — sans qu'il faille mettre à jour la liste noire de chaque tenant.

Cas 3 — Identité content-based pour cross-validation

Deux pipelines d'extraction produisent des `VVL_AbstractObject` à partir du même document — par exemple une étude de cas du BTP (chapitre 20) qui passe à la fois par l'extraction LLM directe et par un système expert flou (chapitre 24). On veut vérifier que les deux pipelines convergent vers la même représentation symbolique.

```
$abs1 = [ontology:abstract "PreferredSupplier" [encoded [country "FR"] [tier "A"]]]
$abs2 = [ontology:abstract "PreferredSupplier" [encoded [country "FR"] [tier "A"]]]
$abs3 = [ontology:abstract "PreferredSupplier" [encoded [country "DE"] [tier "A"]]]

$h1 = [atom:trace $abs1].content_hash
$h2 = [atom:trace $abs2].content_hash
$h3 = [atom:trace $abs3].content_hash

[if ($h1 == $h2) [log "Pipelines A et B convergent (cross-validation OK)"]]
[if ($h1 != $h3) [log "Variantes country FR/DE bien distinguées"]]
```

Le `content_hash` (calculé à partir des `enc(_, _, _)` Zalta) fournit une **identité intensionnelle** : deux abstracts qui encodent les mêmes propriétés produisent le même hash, indépendamment du chemin par lequel ils ont été construits. C'est l'oracle de cross-validation que les chapitres applicatifs invoquaient implicitement.

Cas 4 — Filtre de déterminisme pour hot path

Un système de détection de fraude (chapitre 21) traite des milliers d'événements par seconde dans un hot path où la **jitter LLM** est inacceptable — un score qui varie d'un appel à l'autre, même légèrement, fait diverger les seuils d'alerte. La policy `prod_safe` introduite à la section 23.7 bloque tout atome stochastique en amont :

```
$prod_safe = [policy:define
   [list "deny"  "determinism" "==" "stochastic"]
   [list "allow" "family"      "in" "SymbolicAtom" "GroundedAtom" "ReifiedAtom"]
]

[policy:check $prod_safe [@ "Steel316L"]]      # → denied (LLM = stochastic)
[policy:check $prod_safe [createImage 32 32]]  # → allowed (Img = deterministic)
[policy:check $prod_safe [ontology:abstract "Fraude" [encoded [seuil 0.85]]]]
                                               # → allowed (AbstractObject = deterministic)
```

Le `VVL_ConceptEntity` produit par `[@ "Steel316L"]` est refusé parce que son `determinism` vaut `stochastic` (l'enrichissement LLM peut varier), tandis que les abstracts Zalta et les images sont autorisés. Le hot path conserve sa stabilité numérique sans renoncer à la richesse des autres atomes.

23.9. Synthèse : un protocole, dix optimisations émergentes

Ce chapitre n'a introduit aucun mécanisme algorithmique fondamentalement nouveau — toutes les capacités qu'il décrit (audit, sandbox, identité content-based, filtrage par déterminisme) étaient déjà *réalisables* dans VVL avant le protocole, au prix de code spécifique à chaque cas. Ce qu'apporte le protocole, c'est l'**uniformité** : ces capacités ne sont plus écrites contre des types Python individuels, elles sont écrites contre une typologie partagée, et elles continuent de fonctionner quand de nouveaux types d'atomes sont introduits.

Cette uniformité produit des **optimisations émergentes** que le livre n'avait pas explicitées. L'audit trail devient calculable depuis un seul builtin (`atom:trace`). Le sandboxing devient déclaratif et auditable. La déduplication par hash devient triviale (`content_hash` est un champ standardisé). Le cost tracking peut interroger `cost_hint` sans connaître le type. Le filtrage par déterminisme s'écrit sans `isinstance`. La cross-validation de pipelines devient un test d'égalité de hashes. Le replay déterministe peut filtrer en amont les atomes stochastiques. Les politiques multi-tenant deviennent des fichiers JSON — externalisables, versionnables, auditables.

Sur le plan théorique, le protocole donne enfin un **contenu calculable** au discours neuro-symbolique du livre. Quand le chapitre 1 affirmait que VVL articule trois couches — neuronale (LLM amorphe), zaltaïenne (réifié), symbolique (Prolog) —, cette articulation restait du commentaire. Avec les trois familles SymbolicAtom / GroundedAtom / ReifiedAtom, l'oscillation devient une **inspection** : `[atom:family $x]` répond à *« cet objet vit-il dans quelle couche ? »*, et toutes les optimisations transversales en découlent.

Avant le protocole	Avec le protocole
Audit trail : code spécifique par type, logs hétérogènes	`[atom:trace]` uniforme, dict JSON structuré
Sandbox : whitelist par nom de builtin	`[policy:define]` par axes sémantiques, default-deny
Identité : implicite, dispersée	`runtime_id` / `content_hash` / `semantic_id` orthogonaux
Cost tracking : invariant absent du langage	`cost_hint` / `latency_hint` interrogeables
Cross-validation : comparaison ad-hoc d'attributs	Égalité de `content_hash`
Discours neuro-symbolique : implicite (commentaires)	Calculable (`[atom:family]`)

Figure 23.7 — *Six dimensions transversales : avant et après le protocole*



Figure 23.8 — *Le protocole d'atomes comme couche d'introspection unifiée entre VVLNode et les applications*

Statut et perspectives

Trois extensions sont déjà envisagées et figurent dans la roadmap : un septième axe `tenant_id` pour matérialiser l'isolation multi-tenant directement dans la fiche signalétique, une capability `Replayable` pour les atomes qui supportent le rejeu déterministe à partir d'un snapshot, et un builtin `[policy:explain]` qui produirait, pour un atome donné, l'ensemble des règles qui le concernent dans une politique — sans attendre qu'un `[policy:check]` explicite déclenche l'évaluation. Ces extensions ne sont pas garanties — elles dépendent du retour d'usage.

Pour un développeur travaillant aujourd'hui en VVL stable (5.6 et antérieurs), le message pratique est simple : continuer d'utiliser les mécanismes des chapitres 1 à 22, qui restent pleinement supportés. Pour un développeur prêt à expérimenter, les six builtins introduits ici offrent un raccourci substantiel sur tout ce qui touche à l'auditabilité, au sandboxing déclaratif et à la traçabilité de la dérivation neuro-symbolique. Les retours d'expérience sont attendus sur la branche dédiée.

Le **chapitre 24** revient à la programmation industrielle stable avec le système expert et la logique floue — comment bootstraper un moteur de règles depuis le langage naturel en s'appuyant sur les opérateurs flous T-normes et les hedges linguistiques. La transition est volontaire : après avoir vu le protocole d'atomes formaliser la *typologie* du runtime, on revient à la *production de connaissances* — l'autre face de la programmation neuro-symbolique en VVL.

24. Système expert et logique floue

Mots-clés : système expert, logique floue, T-norme, Zadeh, probabiliste, Łukasiewicz, fuzzy:ontology, fuzzy:rule, expert, hedges, dispatcher, persistance, abstracts Zalta

La Partie VII s'est ouverte au chapitre 22 sur les agents autonomes — `[agent]` et `[architect]` —, qui exploitent l'homoiconicité de VVL pour générer du code à la volée. Ce chapitre 24 explore une autre forme d'autonomie qui s'inscrit dans la même philosophie de **délégation au LLM** mais répond à un besoin différent : la **construction de systèmes experts** par description en langage naturel. Là où l'agent improvise pour chaque tâche, le système expert construit une **structure formelle réutilisable** — un classifieur, un évaluateur de risque, un trieur de tickets — depuis quelques phrases qui décrivent le domaine et les règles métier.

L'enjeu est considérable. Les systèmes experts classiques — MYCIN, R1, CLIPS — exigeaient des **knowledge engineers** spécialement formés pour traduire la connaissance des experts métier en règles formelles. Ce travail prenait des mois ou des années, et dépassait souvent les budgets. La promesse de ce chapitre est radicale : avec les builtins `[fuzzy:ontology]`, `[fuzzy:rule]` et `[expert]`, **n'importe quel utilisateur** peut décrire son domaine en français et obtenir un système expert opérationnel en quelques secondes — sans écrire une seule ligne de VVL à la main, et avec un runtime entièrement déterministe au moment de l'exécution.

La pierre angulaire de cette capacité est la **logique floue**. Les règles métier réelles sont rarement des conditions binaires (« si température == 39 alors grippe ») ; elles sont graduelles (« si fièvre élevée et toux sèche et fatigue intense, probablement grippe »). La logique floue, formalisée par Lotfi Zadeh dans les années 1960, fournit le cadre mathématique pour ce type de raisonnement : appartenance partielle à un ensemble (un score entre 0 et 1), opérateurs AND/OR/NOT graduels, modificateurs linguistiques (« très », « plutôt »). VVL intègre tout cet appareil avec une syntaxe naturelle — opérateurs infix `~&`, `~|`, `~!` —, et le combine au système ontologique pour produire des classifieurs robustes.

Le chapitre suit la séquence : motivation (24.1), opérateurs flous et familles T-normes (24.2), hedges et quantifieurs (24.3), génération d'ontologies par [fuzzy:ontology] (24.4), génération de règles par [fuzzy:rule] (24.5), bootstrap one-shot par [expert] (24.6), persistance dans le workspace (24.7), étude de cas du triage de tickets (24.8), optimisations sous le capot (24.9), synthèse des trois paradigmes d'écriture des règles (24.10).

24.1. Motivation : du programme écrit au programme décrit

Les chapitres précédents ont présenté plusieurs façons d'introduire des règles métier dans un programme VVL. **Au chapitre 12**, le développeur écrit ses règles à la main avec `[ontology:rule]` — c'est précis mais coûteux en effort. **Aux chapitres 16 et 17**, le développeur fournit un dataset d'exemples et `[ontology:learn]` induit les règles automatiquement — c'est plus rapide mais exige un corpus d'apprentissage représentatif. Ce chapitre 24 propose une troisième voie qui s'avère souvent la plus économique en pratique : le développeur **décrit** ses règles en langage naturel, et le système les compile en règles ontologiques formelles.

Cette approche par description répond à un cas d'usage très fréquent : celui où **l'expert métier sait formuler ses règles en français** mais n'a ni le temps ni l'envie d'apprendre VVL, et où le dataset d'apprentissage n'existe pas (parce que le système est nouveau, ou parce que les cas historiques ne

sont pas étiquetés). Plutôt que de demander à l'expert de coder ou de constituer un corpus, on lui demande simplement de **dire** ce qu'il sait : « un client est à bon risque s'il a un score de crédit élevé et une ancienneté longue ». Cette phrase, traduite par `[fuzzy:rule]`, devient une règle ontologique exécutable.

Le résultat n'est pas un système à part, parallèle aux mécanismes précédents : c'est une **règle `ontology:rule` standard**, qui s'intègre au système d'inférence du chapitre 12, peut être combinée à des règles écrites à la main, peut participer à une cascade par priorités, peut être inspectée et modifiée par l'expert s'il en a la capacité. La compilation n'enferme pas le développeur dans un sous-langage — elle lui ouvre une porte d'entrée plus rapide vers le langage commun de l'ontologie.

24.2. Opérateurs flous et familles T-normes

Avant les commandes de haut niveau (`fuzzy:rule`, `expert`), il faut comprendre la couche de base : les **opérateurs flous infix** et les **familles T-normes** qui en définissent la sémantique précise. Quatre opérateurs constituent l'algèbre floue de VVL :

Opérateur	Sémantique	Type	Exemple
~&	T-norme AND	Infix binaire	($p == "fever") ~& ($p == "cough")
~|	T-conorme OR	Infix binaire	($p ~~ "Cold") ~\| ($p ~~ "Flu")
~!	Négation	Préfixe unaire	~! ($p ~~ "Allergy")
≈>	Implication floue	Infix binaire	($p ~~ "Junior") ≈> ($p.salaire <= 40000)

Figure 24.1 — *Les quatre opérateurs flous infix de VVL*

Ces opérateurs respectent les précédences VVL classiques. Mais leur **sémantique exacte** dépend de la **famille T-norme** active. Une T-norme est une fonction `[0,1] × [0,1] → [0,1]` qui généralise le AND booléen ; il en existe une infinité, mais trois familles dominent l'usage. La fonction `[fuzzy:family]` permet de basculer entre elles.

```
# Selectionner la famille active pour la session
[fuzzy:family]                      # => "zadeh" (defaut)
[fuzzy:family "probabilistic"]      # => switche le defaut
[fuzzy:family "lukasiewicz"]        # => switche encore

# Override par appel (sans changer le defaut)
[fuzzy:and 0.8 0.6 :family lukasiewicz]   # => 0.4 (Lukasiewicz)
[fuzzy:and 0.8 0.6 :family zadeh]         # => 0.6 (Zadeh = min)
[fuzzy:and 0.8 0.6 :family probabilistic] # => 0.48 (produit)
```

Les trois familles produisent des résultats différents pour les mêmes entrées. Avec `a = 0.8` et `b = 0.6`, voici le panorama :

Famille	a ~& b	a ~| b	Cas d'usage typique
zadeh (défaut)	min(a,b) = 0.6	max(a,b) = 0.8	Diagnostic, classification — robuste, idempotent
probabilistic	a · b = 0.48	a + b − ab = 0.92	Évidences indépendantes qui s'accumulent
lukasiewicz	max(0, a+b−1) = 0.4	min(1, a+b) = 1.0	Logique stricte avec seuils durs

Figure 24.2 — *Les trois familles T-normes et leur comportement comparé*

Le **choix de famille** n'est pas anodin. **Zadeh** est le défaut — il a la propriété d'idempotence (`a ~& a = a`) qui le rend robuste face à la duplication d'évidences. **Probabilistic** convient quand on **cumule** des évidences indépendantes : si un patient présente deux signes faiblement compatibles avec un diagnostic, leur conjonction probabiliste est *plus faible* que chacun pris séparément, ce qui est mathématiquement correct pour des évidences indépendantes. **Łukasiewicz** est plus strict : `a ~& b` peut tomber à zéro si la somme des scores n'atteint pas le seuil — utile quand on veut une logique « tout ou rien atténuée ».

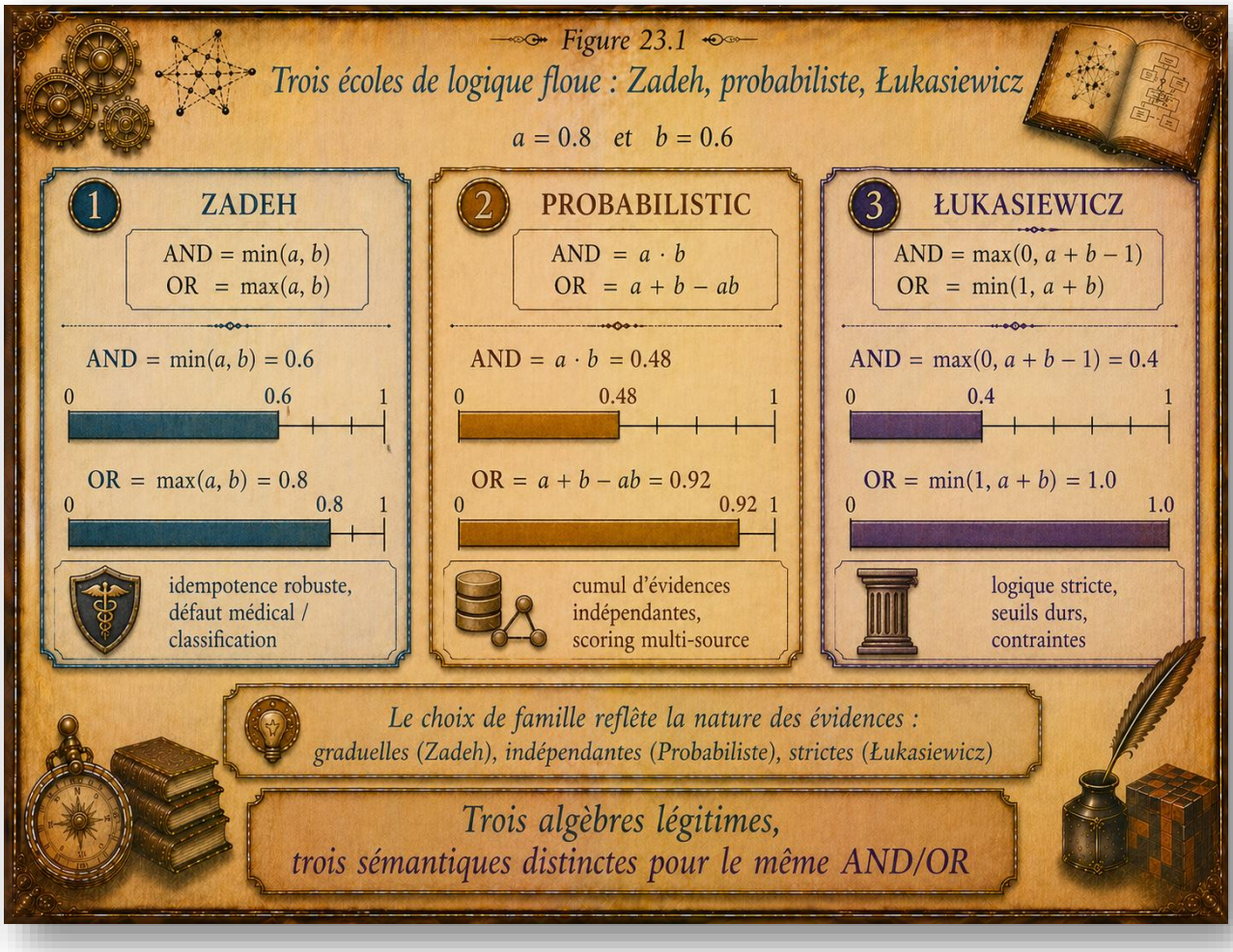


Figure 24.3 — *Les trois familles T-normes : trois sémantiques pour les mêmes opérateurs*

24.3. Hedges linguistiques et agrégateurs

Au-delà des opérateurs binaires, la logique floue de Zadeh reconnaît des **modificateurs linguistiques** — appelés *hedges* — qui ajustent le degré d'appartenance. « très grand » est plus restrictif que « grand » ; « plutôt grand » est plus permissif. Ces nuances, naturelles dans le langage humain, ont des formules précises en logique floue.

Hedge VVL	Formule	Effet	Sur 0.7 →
[fuzzy:very x]	x^2	Concentration (durcit)	0.49
[fuzzy:extremely x]	x^4	Super-concentration	0.24
[fuzzy:somewhat x]	$\sqrt{x}$	Dilation (adoucit)	0.84
[fuzzy:slightly x]	x^(1/3)	Super-dilation	0.89

Figure 24.4 — *Les quatre hedges linguistiques de VVL et leur effet quantitatif*

```
# Cas d'usage : durcir une regle pour un diagnostic certain
[fuzzy:rule "Un patient est tres probablement grippe si tres haute
             temperature et toux seche" $Med]
# => Le LLM produit une regle utilisant fuzzy:very pour la condition :
#    [when ([fuzzy:very ($this.temperature ~~ "elevee")]
#           ~& ($this.toux ~~ "seche"))]
#
# Sur un patient avec temperature=elevee (score = 0.7) et toux=seche (score = 0.9) :
#   sans hedge : 0.7 ~& 0.9 = 0.7 (Zadeh) -> match
#   avec very  : (0.7^2 = 0.49) ~& 0.9 = 0.49 -> potentiellement sous le seuil

# Inversement : assouplir pour une suspicion legere
[fuzzy:rule "Un patient est plutot fatigue si un peu eleve" $Med]
# => [when ([fuzzy:somewhat ($this.fatigue ~~ "elevee")] ...)]
```

À côté des hedges, VVL offre des **quantifieurs et agrégateurs** pour les règles qui combinent plusieurs évidences. `[fuzzy:wmean]` calcule une moyenne arithmétique pondérée — utile quand certaines évidences sont plus importantes que d'autres. `[fuzzy:most]` et `[fuzzy:few]` sont les quantifieurs flous au sens de Zadeh : « la plupart » se calcule comme la moyenne du *top half* des scores, « peu » comme la moyenne du *bottom half*.

```
# Moyenne ponderee : 5 evidences avec poids
$score = [fuzzy:wmean
            [list 0.8 0.6 0.9 0.5 0.7]
            [list 0.3 0.1 0.4 0.1 0.1]]
# = (0.8*0.3 + 0.6*0.1 + 0.9*0.4 + 0.5*0.1 + 0.7*0.1) / (0.3+0.1+0.4+0.1+0.1)
# = 0.78 / 1.0 = 0.78

# Quantifieurs : "la plupart" / "peu"
$signs = [list 0.9 0.8 0.85 0.3 0.4]
[fuzzy:most $signs]      # ~ moyenne du top-half (0.85, 0.9, 0.8) = 0.85
[fuzzy:few $signs]       # ~ moyenne du bottom-half (0.3, 0.4)    = 0.35
```

24.4. fuzzy:ontology — du domaine NL au schéma formel

La commande `[fuzzy:ontology "description"]` prend une description du domaine en langage naturel et produit un schéma `VVL_Ontology` complet — classes, attributs, abstracts Zalta (voir chapitre 13), relations. Elle utilise un seul appel LLM pour la **génération initiale**, après quoi le schéma vit dans `_ONTOLOGIES` et peut être utilisé par toutes les commandes ontologiques classiques.

```
$Med = [fuzzy:ontology "
  Domaine medical : patients (nom, temperature, toux, fatigue) avec profils
  Grippe (fievre + toux seche + fatigue intense) et
  Allergie (pas de fievre, pas de toux, fatigue legere).
"]

# Le LLM produit en interne une expression equivalente a :
#
# [ontology:define med
#     [ontology:class Patient
#         [attributes
#             [nom         String  [key true]]
#             [temperature String  [enum [normale elevee]]]
#             [toux        String  [enum [absente seche grasse]]]
#             [fatigue     String  [enum [legere moderee intense]]]
#             [diagnostic  String  [default ""]]
#         ]
#     ]
#     [ontology:abstract Grippe : Patient
#         [encode temperature "elevee"]
#         [encode toux "seche"]
#         [encode fatigue "intense"]
#     ]
#     [ontology:abstract Allergie : Patient
#         [encode temperature "normale"]
#         [encode toux "absente"]
#         [encode fatigue "legere"]
#     ]
# ]
#
# Effets de bord :
# - Le schema est inscrit dans _ONTOLOGIES["med"]
# - Les abstracts Grippe et Allergie sont commits dans le KB Prolog
#   (abstract(grippe), enc(grippe, temperature, "elevee"), ...)
```

Deux **astuces de prompt** rendent la génération beaucoup plus fiable. **Première astuce** : énumérer les valeurs typiques pour chaque attribut catégoriel (« température : normale/élevée », « toux : absente/sèche/grasse »). Sans cette indication, le LLM peut hésiter entre attributs continus et catégoriels, et produire des schémas inconsistants. **Deuxième astuce** : lister les profils sous une clause explicite (« Profils typiques : ... ») — c'est un déclencheur fiable pour la création des abstracts Zalta correspondants.

Le schéma produit n'est pas figé : il peut être étendu par d'autres commandes (ajout de règles avec `[ontology:rule]`, ajout de classes avec `[ontology:class]`, ajout de relations avec `[ontology:relation]`) ou inspecté par `[ontology:toVVL $Schema]` pour examen ou export. La compilation NL → schéma est donc un **point d'entrée**, pas une boîte noire.

*

24.5. fuzzy:rule — du langage naturel à la règle formelle

La commande `[fuzzy:rule "règle NL" $Schema]` traduit une règle métier exprimée en français en une règle `[ontology:rule]` formelle utilisant les opérateurs flous infix présentés en 24.2. Le mécanisme reconnaît **deux formes structurelles** dans le NL et génère la règle correspondante.

Shape A — règle « si X alors Y »

La forme la plus simple. Le NL contient une condition unique et une action unique. La traduction est directe.

```
[fuzzy:rule "Un patient a la grippe si fievre et toux seche" $Med]

# Genere :
# [ontology:rule "EstGrippe" $Med
#     [target "Patient"]
#     [when ((($this.temperature == "elevee") ~& ($this.toux == "seche")) >= 0.5)]
#     [then [ontology:set $this "diagnostic" "grippe"]]
# ]
```

Le seuil par défaut est `0.5` — la règle se déclenche dès que le score de la condition floue dépasse cette valeur. Pour les règles **sensibles** (médical, légal), on demande explicitement « strict » dans le NL : le LLM ajustera le seuil à 0.7+ ou 0.8+ selon le degré de certitude exigé.

Shape B — dispatcher (énumération multi-alternatives)

Quand le NL décrit **plusieurs alternatives** sur un même champ — typiquement « X est A si profil 1, B si profil 2, sinon C » —, `fuzzy:rule` génère une **règle dispatcher** unique qui choisit l'alternative au plus haut score. Ce design évite que des règles concurrentes se piétinent.

```
[fuzzy:rule "
  Le diagnostic est :
    'grippe'   si profil Grippe,
    'allergie' si profil Allergie,
    sinon 'a_etudier'.
" $Med]

# Genere :
# [ontology:rule "Diagnostic" $Med
#     [target "Patient"]
#     [when 1]                        # toujours active
#     [then [ontology:set $this "diagnostic"
#         [getValue 2 [max [
#             [($this ~~ "Grippe")    "grippe"]
#             [($this ~~ "Allergie")  "allergie"]
#             [0.5                    "a_etudier"]    # floor par defaut
#         ]]]
#     ]]
# ]
```

Trois mécanismes coopèrent dans cette règle. **`[max ...]`** sélectionne la branche au plus haut score parmi les alternatives. **`[getValue 2 ...]`** extrait le second élément du couple — le label de l'alternative gagnante. **La paire de fin `[0.5 "a_etudier"]`** sert de *floor* : si tous les vrais scores sont sous 0.5, c'est

cette branche par défaut qui l'emporte. Cette construction transforme une simple règle conditionnelle en un **mini-classifieur multi-classes**.

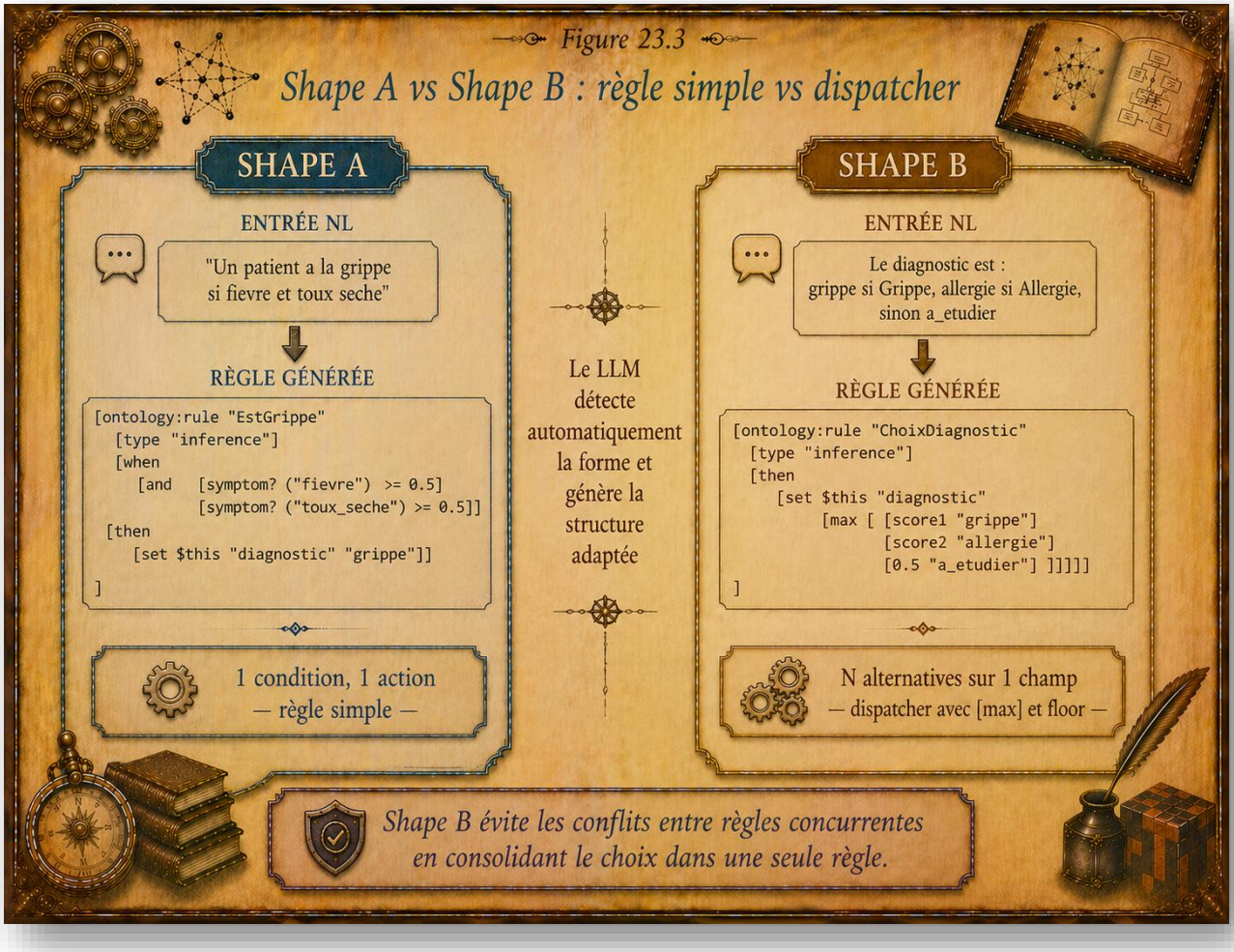


Figure 24.5 — *Les deux formes générées par fuzzy:rule selon la structure du NL*

24.6. expert — bootstrap one-shot d'un système expert

La commande `[expert "description complète" "Nom"]` est l'apothéose du système. Elle compose en un seul geste les opérations précédentes — `[fuzzy:ontology]` pour le schéma, `[fuzzy:rule]` pour les règles — et **enregistre un keyword VVL** `<Nom>` qui prend en argument une chaîne NL et exécute le pipeline complet : extraction (`fromText`), commit, inférence, retour de l'instance classifiée.

```
# Bootstrap one-shot : 1 seul appel pour tout creer
[expert "
  Domaine evaluation risque credit. Dossier client : nom, revenu,
  score_credit (eleve/moyen/faible), anciennete (longue/moyenne/recente),
  decision. Profils BonRisque (score_credit eleve, anciennete longue),
  RisqueModere (score_credit moyen), RisqueEleve (score_credit faible).
  decision : 'approuve' si BonRisque, 'a_etudier' si RisqueModere,
  'refuse' si RisqueEleve, sinon 'a_etudier'.
" "Scoring"]

# A partir de maintenant, le keyword <Scoring> est disponible :

[Scoring "Alice Dubois, revenu 4500, score eleve, anciennete longue"]
# => <Ontology:DossierClient [..., decision="approuve"]>

[Scoring "Bob Martin, revenu 2800, score moyen, anciennete moyenne"]
# => <Ontology:DossierClient [..., decision="a_etudier"]>

[Scoring "Carol Petit, revenu 1800, score faible, anciennete recente"]
# => <Ontology:DossierClient [..., decision="refuse"]>
```

Sous le capot, la commande exécute trois opérations distinctes. **(1)** `[fuzzy:ontology]` génère le schéma et l'enregistre sous le nom `$<Nom>_schema` (ici `$Scoring_schema`) — ce nom est déterministe quel que soit le binding choisi par l'utilisateur. **(2)** `[fuzzy:rule]` génère les règles et les attache au schéma. **(3)** un keyword VVL est enregistré dont le corps applique le pipeline standard à l'argument :

```
# Corps du keyword genere automatiquement :
[keyword Scoring
    `[do
        $__expert_instance = [ontology:fromText $Scoring_schema "DossierClient" @1]
        [ontology:commit $__expert_instance]
        [ontology:infer $Scoring_schema]
        $__expert_instance
    ]
]
```

Trois remarques sur ce mécanisme. **Premièrement, `@1` est l'argument de l'appel** — c'est la convention des keywords VVL pour la substitution positionnelle. **Deuxièmement, `$__expert_instance` est une variable interne** préfixée pour éviter les collisions avec les variables de l'utilisateur — l'expert ne pollue pas l'espace de noms. **Troisièmement, le retour est l'instance classifiée elle-même** — un `VVL_OntologyInstance` complet avec tous ses attributs extraits et déduits, immédiatement utilisable par le code appelant.

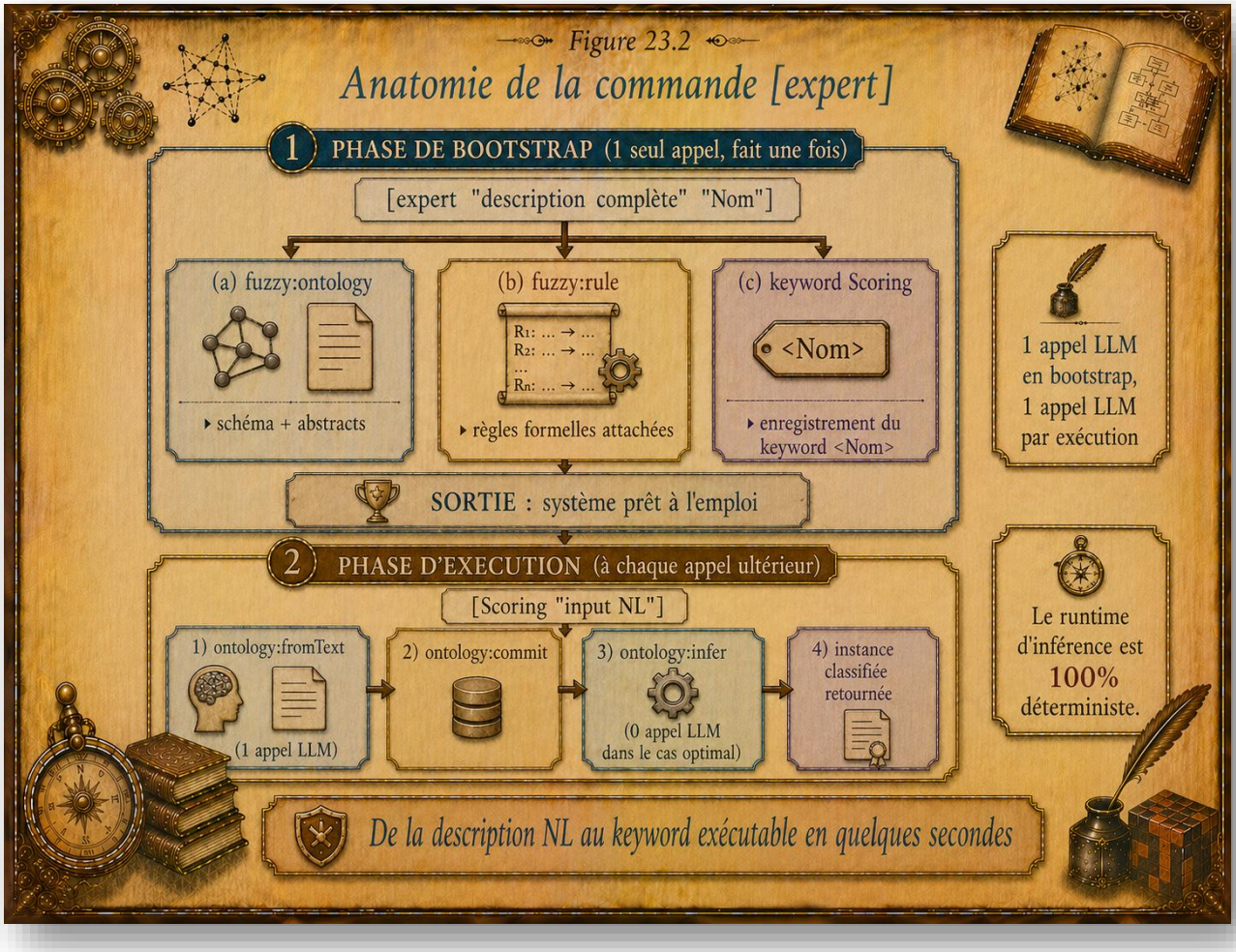


Figure 24.6 — *La commande [expert] : un système expert prêt à l'emploi en un seul appel*

24.7. Persistance : expert:save / load / list

Les experts créés par `[expert]` sont des **objets de session** par défaut — ils disparaissent à la fin du processus VVL. Pour les rendre **persistants**, trois commandes les gèrent dans le **workspace actif** : `[expert:save]`, `[expert:load]`, `[expert:list]`. Un workspace doit être activé au préalable par `[workspace go <nom>]`, sinon les commandes refusent l'opération avec un message explicite.

```
# 1. Activer un workspace
[workspace go demo]

# 2. Sauvegarder l'expert "Scoring" cree precedemment
[expert:save "Scoring"]
# => Expert 'Scoring' saved at <ws>/ontology/Scoring/
#    (kb.bin, kb.pl, schema.json, schema.vvl, expert.vvl, expert.json,
#     meta.json)

# 3. Lister les experts disponibles dans le workspace
[expert:list]
# => [{name: "Scoring",
#      schema_name: "credit_risk",
#      main_class: "DossierClient",
#      description: "Domaine evaluation risque credit...",
#      path: "<ws>/ontology/Scoring"},
```

```
#     ...]

# === Dans une nouvelle session ou apres restart ===

[workspace go demo]

[expert:load "Scoring"]
# Rejoue schema.vvl (recree classes/abstracts/regles)
# puis expert.vvl (reenregistre le keyword <Scoring>)
# => Expert 'Scoring' loaded.

# Le keyword est immediatement utilisable :
[Scoring "Eve, revenu 5000, score eleve, ancienne longue"]
# => instance avec decision="approuve"
```

Trois observations sur cette persistance. **Premièrement, sept fichiers sont produits** par `expert:save` : `kb.bin` et `kb.pl` (la base de connaissance Prolog binaire et lisible), `schema.json` et `schema.vvl` (le schéma sérialisé en JSON et reconstructible en VVL), `expert.vvl` (le code source du keyword), `expert.json` (les métadonnées), `meta.json` (l'horodatage et les versions). Cette redondance entre formats binaires et textuels permet à la fois la performance de chargement et l'auditabilité humaine. **Deuxièmement, le mécanisme est jail-protégé** par le workspace — un expert sauvé dans un workspace ne peut pas être chargé depuis un autre, ce qui garantit l'isolation des projets. **Troisièmement, `expert.vvl` est lisible et modifiable** — l'utilisateur peut éditer le keyword à la main pour ajouter des étapes de prétraitement, du logging, ou des conditions supplémentaires.

24.8. Étude de cas : triage de tickets support multi-décisions

Un cas particulièrement parlant illustre la puissance de `[expert]` : le **triage automatique de tickets de support** d'un SaaS. Chaque ticket arrive en texte libre et doit être classé sur **deux dimensions indépendantes** : la priorité (P0/P1/P2/P3) et la catégorie (incident/demande/low_prio). Quand le NL décrit plusieurs champs de décision, `fuzzy:rule` génère **une règle par champ**, chacune en Shape B dispatcher.

```
[expert "
  Domaine support SaaS. Ticket avec titre, description, type_probleme
  (panne_majeure/bug/question), impact_utilisateurs (eleve/moyen/faible),
  urgence_business (critique/haute/basse), priorite, categorie.
  Profils Critique (panne_majeure, impact eleve, urgence critique),
  Important (bug, impact moyen),
  Routine (question),
  Mineur (impact faible).
  La priorite est : 'P0' si Critique, 'P1' si Important,
                    'P2' si Routine, 'P3' si Mineur, sinon 'P2'.
  La categorie est : 'incident' si Critique ou Important,
                     'demande' si Routine, 'low_prio' si Mineur,
                     sinon 'demande'.
" "TicketTriager"]

# Test sur un ticket critique
[TicketTriager "Production en panne complete, panne_majeure,
                impact eleve, urgence critique"]
# => Ticket[priorite="P0", categorie="incident"]

# Test sur un ticket de routine
[TicketTriager "Comment changer ma photo de profil, question simple"]
# => Ticket[priorite="P2", categorie="demande"]

# Test sur un ticket mineur
```

```
[TicketTriager "Petit bouton cosmetique pale, impact faible"]
# => Ticket[priorite="P3", categorie="low_prio"]
```

Trois remarques sur ce cas d'usage. **Premièrement, deux règles dispatcher cohabitent** sur la même classe `Ticket` — une pour `priorite`, une pour `categorie` — et coopèrent sans conflit grâce au design Shape B. **Deuxièmement, les profils peuvent être réutilisés** dans plusieurs règles : le profil `Critique` apparaît dans les deux dispatchers, ce qui garantit la cohérence des décisions. **Troisièmement, le système classifie sans intervention humaine** des tickets dont la formulation est très variable, parce que l'extraction `fromText` aligne les valeurs sur les énumérations encodées dans les abstracts.

La sortie est immédiatement utilisable par le système d'orchestration des tickets — routage automatique vers les bonnes équipes, escalade selon la priorité, mise en file d'attente. En production, ce keyword peut être appelé sur des dizaines de milliers de tickets par jour pour un coût marginal d'un seul appel LLM par ticket (le `fromText`), avec décision déterministe et auditable.

24.9. Optimisations sous le capot

La promesse « 0 appel LLM au runtime sauf pour l'extraction » mérite explication. Deux optimisations majeures rendent ce comportement possible et démontrent comment un système neuro-symbolique bien conçu peut atteindre des performances opérationnelles comparables à celles d'un système purement formel.

Cache de similarité sémantique

L'opérateur `~~` (satisfies hybride) opère en **quatre niveaux** de coût croissant (chapitre 13) : matching strict, seuil numérique, substring containment, LLM batché. Le quatrième niveau — celui qui appelle le LLM pour scorer la similarité de paires non résolues symboliquement — est **mémoïsé par process** avec une clé normalisée order-insensitive : `tuple(sorted(lower(a), lower(b)))`. Conséquence : les valeurs récurrentes (« elevée », « moyen », « faible », « critique ») ne consomment qu'**un seul** appel LLM dans toute la durée de vie du process, jamais deux.

Le hit-rate du cache approche **100 %** en steady-state. Pour un système qui traite des milliers de tickets ou de transactions, cela signifie que les premiers cas paient le coût d'apprentissage des similarités sémantiques, et tous les cas suivants en bénéficient gratuitement. La fonction `reset_similarity_cache()` est exposée pour les tests unitaires qui veulent isoler chaque cas.

Extraction enum-aware

La seconde optimisation est plus subtile. `[ontology:fromText]` reçoit, pour chaque attribut String/Email/Boolean, la **liste des valeurs encodées par les abstracts** du schéma comme hint dans le prompt :

```
# Hint genere automatiquement pour le LLM lors de fromText :
"experience": "(String) L'experience du candidat. PREFER one of: ['elevee', 'faible']"
```

Le system-prompt impose au LLM de **choisir parmi cette liste** (ou de retourner une chaîne vide si aucune ne convient). La conséquence est cruciale : les attributs extraits matchent **symboliquement**

les abstracts au niveau 1 de `satisfies_hybrid` — donc gratuitement, sans aucun appel LLM. Sans cette optimisation, l'extraction produirait des paraphrases (« 8 ans d'expérience ») et chaque évaluation ~~ tomberait au niveau 4, déclenchant un appel LLM batché. Avec cette optimisation, l'inférence est instantanée.

L'effet combiné des deux optimisations transforme le profil de coût d'un expert. Une session typique sur 1000 tickets : **un appel LLM par ticket** pour l'extraction (~3 secondes chacun), **zéro appel LLM** pour les inférences (déterministe). Les premiers tickets éprouvent éventuellement un peu de latence supplémentaire pour réchauffer le cache de similarité ; les suivants tournent à pleine vitesse. C'est ce qui permet à un expert d'être déployé en production avec des coûts maîtrisés et une latence prévisible.

24.10. Synthèse : trois paradigmes d'écriture des règles

Avec ce chapitre 24, **trois paradigmes** distincts permettent désormais d'introduire des règles métier dans un système VVL. Ils ne s'opposent pas — ils se complètent et peuvent cohabiter dans le même programme.

Paradigme	Chapitre	Source	Effort dev.	Cas d'usage
Règles écrites à la main	12	Connaissance directe du dev	Élevé	Règles précises, auditables
Règles induites par dataset	16-17	Corpus d'exemples	Moyen	Quand le corpus existe
Règles décrites en NL	23	Description par expert	Faible	Bootstrap rapide, expert non-codeur

Figure 24.7 — *Les trois paradigmes d'écriture des règles métier en VVL*

Trois remarques transversales émergent de cette comparaison. **Première remarque : la cible commune est `[ontology:rule]`**. Quel que soit le paradigme d'origine, les règles finissent dans le même système d'inférence avec les mêmes mécanismes de priorité, de cascade, de mondes possibles, de persistance. Le développeur peut commencer par décrire en NL pour un prototype rapide, puis migrer vers des règles écrites à la main quand le besoin de précision augmente — sans rien refondre.

Deuxième remarque : ces paradigmes peuvent se mélanger dans un même schéma. Un système réel peut avoir une majorité de règles écrites à la main (cœur métier audité), quelques règles induites depuis l'historique (cas particuliers émergents), et quelques règles décrites en NL (politique récemment introduite par la direction). Tant que les noms et les priorités sont distincts, le moteur d'inférence les compose sans tension.

Troisième remarque : le coût décroît avec la délégation. Le paradigme NL/expert est le moins coûteux en effort développeur, mais aussi le plus risqué en termes de précision — il dépend de la qualité de la description et de la justesse de la traduction par le LLM. La discipline professionnelle consiste à choisir le bon paradigme par règle, à valider les règles décrites en NL avec un jeu de tests, et à promouvoir progressivement les règles bien rodées vers du code écrit à la main quand l'audit l'exige.

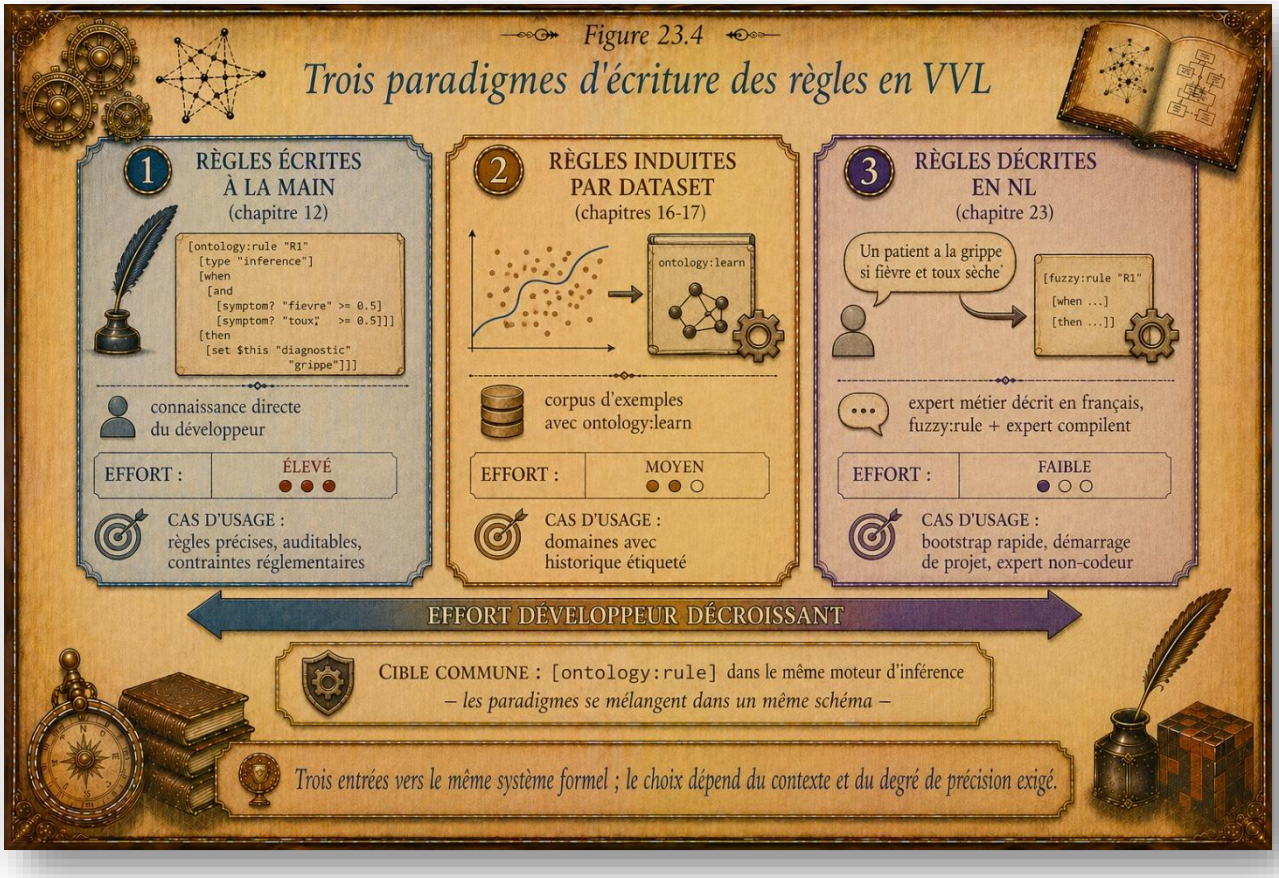


Figure 24.8 — *Trois paradigmes, une cible commune : les règles ontologiques formelles*

Le chapitre 22 a montré la délégation au LLM pour la **génération de code à la volée** (agents) ; ce chapitre 24 a montré la délégation pour la **construction de structures formelles réutilisables** (experts). Les deux capacités relèvent de la même philosophie — confier au LLM la part créative, garder pour le runtime symbolique la part rigoureuse. Le **chapitre 25 aborde une troisième façon de décider, cette fois sans LLM du tout : l'optimisation sous contraintes, où le problème est décrit en variables et en contraintes dures, et résolu exactement par un solveur. Système expert et solveur ne s'opposent pas : l'un excelle quand les règles sont nombreuses et floues, l'autre quand elles sont peu nombreuses et strictes**.

25. Optimisation par contraintes

Mots-clés : optimisation, CP-SAT, OR-Tools, contraintes, décisions, hard/soft, NL → modèle, intent, lexicographique, multi-instance, paramétrique, why-infeasible, géocodage, routing, geo:distance, geo:haversine, route:matrix

Les chapitres précédents ont abordé deux grandes familles de problèmes traités par VVL : la **classification** (qui appartient à quelle catégorie ?) avec les opérateurs Zalta et le système ontologique, et la **décision contextualisée** (que faire face à ce cas ?) avec les règles métier en cascade. Ce chapitre 25 introduit une troisième famille, peut-être la plus exigeante mathématiquement : l'**optimisation sous contraintes** — trouver la meilleure affectation possible de valeurs à des variables, dans un espace défini par des règles dures et adouci par des préférences.

L'optimisation sous contraintes est un domaine ancien et bien étudié de l'informatique, avec des solveurs spécialisés très matures : **CP-SAT** (Google OR-Tools), Z3, MiniZinc, Gurobi. Ces solveurs résolvent de manière prouvablement optimale des problèmes NP-difficiles que les approches heuristiques (algorithmes génétiques, recuit simulé) ne peuvent garantir. Mais ils partagent un défaut récurrent : leur **interface utilisateur est austère**. Construire un modèle CP-SAT en Python exige des centaines de lignes de code de plomberie, des erreurs subtiles aux conséquences silencieuses (un solveur qui retourne une solution infaisable parce qu'on a oublié de réifier un booléen), et un mapping pénible entre les variables symboliques du métier et les `IntVar` indexés du solveur.

VVL apporte une approche différente. Plutôt que de cacher le solveur derrière une abstraction qui le déforme, VVL **expose CP-SAT** comme backend nommé tout en proposant une syntaxe conceptuelle de haut niveau : décisions typées, contraintes dures et molles, opérateurs flous, intentions en langage naturel. Le compilateur traduit un sous-ensemble documenté de VVL vers les contraintes entières du solveur, et **toute résolution sémantique** — opérateurs `~~`, `[§ ...]`, intentions NL — est faite **au moment de la compilation**, jamais à l'intérieur de la boucle solveur. Le résultat est un système où le métier se rapproche du modèle, sans rien sacrifier de la rigueur du solveur sous-jacent.

Le chapitre suit la séquence : motivation et architecture (25.1), formes du langage et garanties opérationnelles (25.2), sous-ensemble compilable et validations (25.3), du langage naturel au modèle compilé (25.4), opérateurs flous au compile-time (25.5), patterns avancés — multi-objectif, globales, multi-instance, paramétrique (25.6), routing géographique avec [geo:distance] et [route:matrix] (25.7), trois niveaux d'expression (25.8), synthèse (25.9).

25.1. Architecture : VVL conceptuel sur CP-SAT

Le principe de design du sous-système `optimize` est la **transparence backend** : on n'invente pas un nouveau solveur, on n'enferme pas l'utilisateur dans une abstraction au-dessus d'OR-Tools. Le backend CP-SAT est nommé explicitement par `[backend "cp-sat"]` dans les options de résolution, et d'autres backends (Z3, MiniZinc) sont réservés pour le futur — leur tentative d'usage est rejetée avec un message d'erreur clair. Cette discipline évite la dérive habituelle des wrappers d'optimisation, qui finissent par n'exposer qu'une fraction des capacités du solveur sous-jacent.

La couche que VVL ajoute par-dessus CP-SAT n'est donc pas un solveur, c'est un **compilateur**. L'utilisateur manipule des formes conceptuelles — `[decision]`, `[assert]`, `[soft]`, `[maximize]` —, le compilateur traduit ces formes vers le modèle CP-SAT, lance le solveur, et **réinjecte le résultat** sous

forme exploitable : un objet ontologique typé, un `VVL_Json` structuré, ou un rapport en langage humain. À aucun moment l'utilisateur ne voit la sortie brute du solveur.

Quatre principes définissent l'architecture :

Premier principe — backend exposé, pas caché. Le solveur a un nom, des options, des messages d'erreur explicites. Ce n'est pas un détail d'implémentation.

Deuxième principe — résolution sémantique au compile-time, pas au runtime. Toute expression contenant `[§ "..."]`, ~~, ou un autre opérateur médié par un LLM est résolue **avant** le démarrage du solveur, et le résultat est mis en cache sur disque. **Aucun appel LLM n'a jamais lieu à l'intérieur de la boucle solveur** — c'est une garantie opérationnelle qui rend les performances prédictibles.

Troisième principe — artefacts inspectables. `[optimize:explain name]` affiche le modèle CP-SAT compilé sous forme lisible ; `[optimize:code name]` retourne le corps VVL enregistré en chaîne de caractères ; `[optimize:validate name]` rapporte chaque expression non compilable avec une suggestion de correction. Le développeur n'est jamais devant une boîte noire.

Quatrième principe — NL → VVL en couches, pas par magie. La commande `[optimize "NL" name]` appelle le LLM pour produire un bloc `[optimize:problem ...]`, le parse comme du VVL standard, et l'enregistre. L'utilisateur peut inspecter ce que le LLM a produit avec `[optimize:code]` et l'éditer si besoin. Le LLM n'est qu'un assistant à la rédaction du modèle, pas un substitut au solveur.

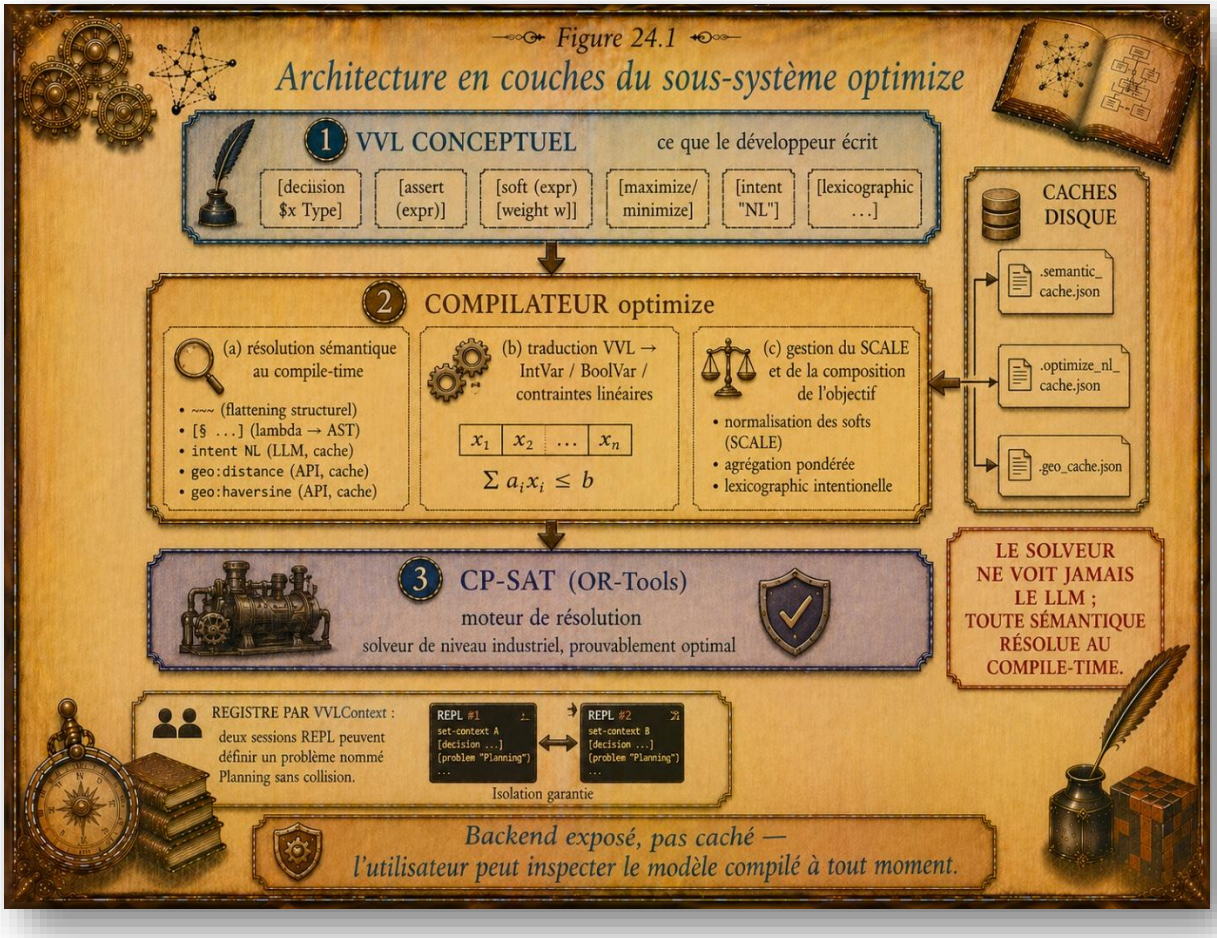


Figure 25.1 — *Architecture en couches : VVL conceptuel, compilateur intermédiaire, CP-SAT*

25.2. Les formes du corps de problème

Un problème d'optimisation est déclaré par `[optimize:problem NAME body...]`. Le corps est composé d'une suite libre de **formes** spécifiques à `optimize`, dans n'importe quel ordre, en quantité quelconque. Le tableau suivant en récapitule les principales :

Forme	Rôle
[decision $x Type [opts]]	Déclare une variable de décision
[assert (expr)]	Contrainte dure (doit être satisfaite)
[soft (expr) [weight w]]	Contrainte molle (contribue à l'objectif)
[maximize (expr)] / [minimize (expr)]	Objectif primaire à maximiser ou minimiser
[scale N]	Échelle de discrétisation (défaut 100)
[intent "NL"]	Forme NL → corps via LLM, expansée au compile-time
[lexicographic [maximize ...] ...]	Multi-objectif séquentiel
[all-different $a $b ...]	Toutes valeurs distinctes (CP-SAT natif)
[cumulative starts dur demands cap]	Contrainte cumulative (CP-SAT natif)
[circuit [(from to bool) ...]]	Contrainte de circuit hamiltonien
[instance ClassName id]	Décision multi-instance ontologique
[for $var $list body]	Expansion paramétrique au parse-time

Figure 25.2 — *Les formes du corps d'un problème d'optimisation*

Types de décisions

Trois types de décisions couvrent la majorité des besoins. **Integer** est le cheval de bataille : un entier borné, utilisable comme binaire (`[min 0] [max 1]`) ou comme compteur. **Enum** déclare une décision sur un ensemble fini d'étiquettes (« junior », « senior », « manager ») ; en interne c'est un `IntVar` dans `[0, n-1]`, mais le résultat exposé est l'**étiquette** correspondante, pas l'index. C'est le seul type qui permet le matching flou par `~~` au compile-time. **Fuzzy** déclare une grandeur continue dans un intervalle `[min, max]` ; en interne c'est un `IntVar` discrétisé selon le `scale`, et le résultat est exposé comme un float dans la plage déclarée.

```
1  [decision $x Integer [min 0] [max 10]]
2  [decision $r Enum    [values "junior" "senior" "manager"]]
3  [decision $d Fuzzy   [min 0.0] [max 1.0]]
```

Une discipline importante : ne pas utiliser `Fuzzy` comme un *wrapper* autour d'un binaire. Si la décision est conceptuellement un oui/non, déclarer un `Integer [min 0] [max 1]` ; si c'est une étiquette parmi un petit ensemble, utiliser `Enum` ; ne réserver `Fuzzy` qu'aux véritables grandeurs continues (un niveau de satisfaction, une probabilité, une température). Cette discipline rend le modèle plus lisible et le solveur plus efficace.

Décisions ontologiques. Lorsque le problème est attaché à un schéma ontologique par `[optimize:problem name $schema "ClassName" body...]`, les attributs de la classe marqués `[decision true]` deviennent automatiquement des variables de décision CP-SAT. Décisions locales (`[decision $x ...]`) et décisions ontologiques peuvent cohabiter dans le même problème — c'est ce qui permet d'attacher des décisions auxiliaires à un objet métier persistant. Une nuance

d'implémentation actuelle : pour les attributs ontologiques, **seul le type `Integer` est supporté** ; les types `Enum` et `Fuzzy` sur attributs `[decision true]` sont rejetés avec un message qui suggère de déclarer la décision **localement** (`[decision $x Enum [values ...]]`) plutôt que sur la classe.

Garanties opérationnelles

Plusieurs propriétés du sous-système ne sont pas immédiatement visibles à la lecture des formes du langage, mais conditionnent le comportement en production. Ce paragraphe les rassemble pour qu'aucune ne soit oubliée :

Registre isolé par contexte. Le dictionnaire des problèmes enregistrés n'est pas un singleton global du module : il est porté par le `VVLContext` actif. Deux sessions REPL parallèles, deux requêtes HTTP simultanées, deux workers concurrents peuvent **chacun** définir un problème nommé `Planning` sans collision — chaque contexte voit son propre `Planning`. Cette discipline est essentielle dès que le serveur VAILS est exposé en multi-session.

Compilation non fatale. Si `[optimize:problem ...]` échoue à compiler — référence à une décision inexistante, expression non compilable, contrainte mal formée —, le problème **est tout de même enregistré**, avec un champ interne `_compile_error` qui consigne le message. Cela permet d'utiliser ensuite `[optimize:validate name]`, `[optimize:explain name]` ou `[optimize:code name]` pour diagnostiquer ce qui s'est passé. Un appel ultérieur à `[optimize:solve name]` retournera une chaîne d'erreur qui pointe explicitement vers `[optimize:validate]`, plutôt que de lever une exception qui interromprait le programme.

Whitelist fermée des formes. Le parseur du corps reconnaît un ensemble strictement délimité de formes : `decision`, `assert`, `soft`, `intent`, `scale`, `maximize`, `minimize`, `lexicographic`, `all-different`, `cumulative`, `circuit`, `instance`, `for`. Toute forme inconnue déclenche un rejet explicite avec **suggestion de correction** : un `[maximise ...]` (orthographe britannique) ou un `[max ...]` (raccourci tentant) sont refusés avec « unknown body form `[maximise ...]` — did you mean `[maximize ...]` ? ». Aucune forme n'est silencieusement ignorée — un point essentiel pour la lisibilité des modèles produits par le LLM.

Détection des collisions de noms exposés. Le compilateur détecte au plus tôt les conflits entre une décision locale `$b` et un attribut ontologique `b` flagué `[decision true]` sur la classe principale du problème. Sans cette garde, les deux variables aboutiraient à la même clé `b` dans le résultat, et la dernière écriture l'emporterait silencieusement. Le compilateur préfère un échec clair (`decision name collision: $b vs class.b — rename one of them`) à un comportement non déterministe.

25.3. Le sous-ensemble compilable

Toutes les expressions VVL ne sont pas compilables vers CP-SAT — c'est une réalité du domaine, pas une limitation arbitraire. Un solveur de programmation par contraintes raisonne sur des entiers et des contraintes linéaires ; les multiplications de variables, les divisions, les puissances, les lambdas, sont hors de son domaine d'expressivité. VVL définit un **sous-ensemble compilable documenté** et la commande `[optimize:validate]` rapporte précisément ce qui sort de ce sous-ensemble, avec une suggestion de correction par cas.

Toujours compilable

Les éléments suivants se compilent sans difficulté vers CP-SAT :

- **Littéraux entiers** et **références** : `$x` (décision locale) ou `$this.attr` (attribut ontologique avec `[decision true]`).

- **Arithmétique linéaire** : `a + b`, `a - b`, `k * x` où `k` est une constante entière. La somme et la différence de décisions sont permises ; la multiplication d'une constante par une décision aussi.

- **Comparaisons** : `==`, `!=`, `<`, `<=`, `>`, `>=`. Permises au sommet d'un `[assert]` (contrainte dure), comme booléen réifié au sommet d'un `[soft]`, ou imbriquées sous des combinateurs flous.

- **Combinateurs flous** : `a ~& b`, `a ~| b`, `a ~> b`, `~! a`, à l'intérieur de `[soft]`. Sur des booléens réifiés, les trois familles T-normes (Łukasiewicz, Zadeh, probabiliste) coïncident avec la logique standard.

Slog 3 — Matching flou ~~

L'opérateur ~~ du chapitre 13 est intégré au sous-ensemble compilable, mais avec une particularité essentielle : il est résolu **au compile-time**, jamais à l'intérieur du solveur. Trois cas se présentent :

Cas 1 : constant ~~ constant (deux littéraux ou un littéral et un `[§ "..."]`). Le compilateur calcule une seule fois le degré entier dans `[0, scale]` via le mécanisme `semantic_binop` (LLM si disponible, sinon fallback substring/token), et l'injecte comme constante dans le modèle.

Cas 2 : Enum_decision ~~ constant. Le compilateur calcule une **table de degrés** — un degré par valeur possible de l'Enum —, et l'expose comme un `IntVar` via la contrainte CP-SAT `AddElement`. Un appel LLM par paire (valeur Enum, cible), mis en cache dans `.semantic_cache.json` ; les exécutions suivantes sont gratuites.

Cas 3 : à l'intérieur de combinateurs flous (`~~ ~| ~~`, etc.). Le ~~ est réifié en booléen avec un seuil par défaut de `scale // 2` (50 % de match avec le scale par défaut). Cela permet de combiner plusieurs préférences floues en une expression unique.

```
# Toujours compilable
[assert ($x + 2 * $y >= 10)]                              # arithmetique lineaire
[assert ($size >= 5)]                                     # comparaison simple en hard
[soft (($age >= 30) ~& ($experience >= 5)) [weight 1]]    # combinateurs flous en soft

# Slog 3 : ~~ au compile-time
[assert (($role ~~ "senior") >= 0.5)]
# Le compilateur calcule un degre par valeur d'Enum, expose comme IntVar.

[soft ($role ~~ [§ "experienced architect"]) [weight 1]]
# Idem, avec le concept réifié comme cible.

# Slog 4 : [§] libre dans expressions
[soft ([§ "rich"] && [§ "wealthy"]) [weight 1]]
# Le compilateur appelle semantic_binop une fois, met en cache, injecte la constante.
```

Auto-scaling des seuils

Quand une comparaison contient ~~, un seuil flottant dans `[0, 1]` du côté opposé est **automatiquement mis à l'échelle** par le compilateur. L'utilisateur écrit `(($r ~~ "senior") >= 0.7)` ; en interne, la comparaison devient `(... >= 70)` avec un scale de 100. Cette discrétion permet d'écrire des seuils naturels tout en gardant la précision entière requise par CP-SAT.

Non compilable — rejeté à validate ou compile

Plusieurs constructions sortent du sous-ensemble compilable et sont rejetées explicitement :

- **Multiplication var × var** (non linéaire) — il faut soit déplacer le calcul dans `[check ...]` post-solve, soit pré-calculer le produit comme coefficient.

- **Littéraux flottants non entiers** hors d'un contexte de comparaison floue — convertir en entier discrétisé via `scale`.

- **Comparaisons imbriquées dans des expressions arithmétiques** — les comparaisons ne sont permises qu'au sommet d'un `[assert]` ou sous un combinateur flou.

- **Autres opérateurs VVL** non listés (`%`, `/`, `**`, lambdas, etc.) — ils n'ont pas de sémantique CP-SAT directe.

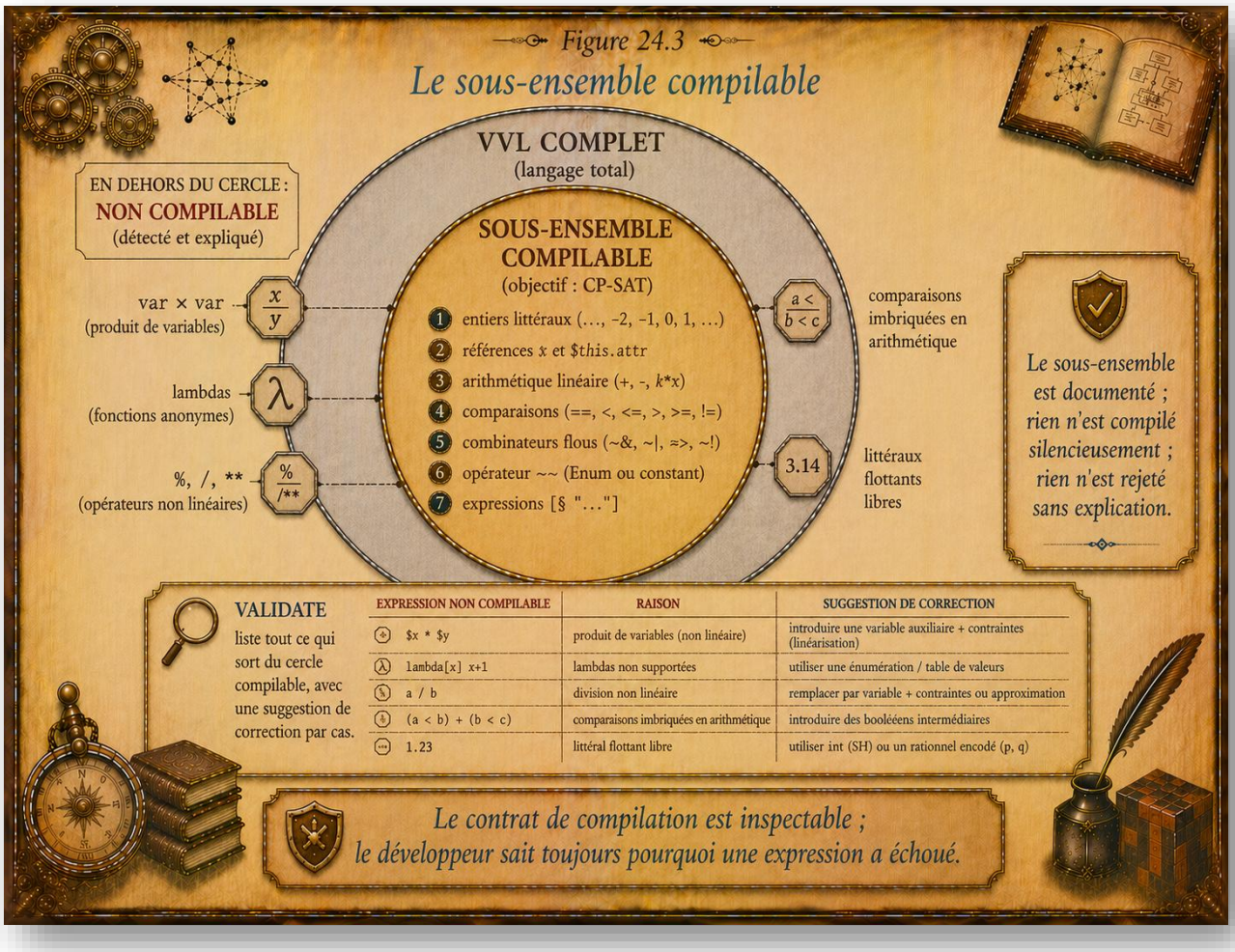


Figure 25.3 — *Le sous-ensemble compilable : ce qui passe à CP-SAT, ce qui ne passe pas*

La discipline de validation est complétée par une **recherche de parité validate ↔ compile** : le compilateur s'efforce de faire en sorte que chaque garde présent dans la compilation soit aussi rapporté par `[optimize:validate]` (collisions de noms, domaines vides, types non supportés, Enum/Fuzzy ontologique). En pratique, cette parité est largement atteinte sur le sous-ensemble cœur — arithmétique linéaire, comparaisons, combinateurs flous, types de décisions. Elle reste **moins complète sur les contraintes globales** (`all-different`, `cumulative`, `circuit`) où certains diagnostics n'apparaissent qu'au moment de la compilation effective. Le réflexe sain consiste donc à appeler `[optimize:validate]` **et** une compilation effective via `[optimize:explain]` ou

`[optimize:solve]` pour s'assurer qu'un modèle est sain — `validate` détecte la majorité des problèmes mais ne les couvre pas tous.

25.4. Du langage naturel au modèle compilé

La commande `[optimize "NL" name]` est l'un des deux points d'entrée naturels du sous-système. Elle prend une description en français (ou en anglais — peu importe) et produit un bloc `[optimize:problem ...]` complet, parsé et enregistré. Le LLM utilisé est l'**agent model** configuré dans `vvl.config`, et le résultat est mis en cache dans `.optimize_nl_cache.json` (clé = MD5 de NL + nom + version du prompt système), si bien que les ré-exécutions du même NL sont gratuites.

```
# Etape 1 : NL -> VVL -> CP-SAT (1 appel LLM, mis en cache)
[optimize "Une equipe de trois personnes — Alice, Bob, Carol — doit
           assurer la presence pendant deux jours. Chaque jour il faut
           au moins deux personnes. Alice ne peut travailler qu'un seul
           jour. Bob prefere le matin, Carol prefere la fin. Minimiser
           le nombre total de jours travailles."
         "Planning"]
# => "Problem 'Planning' compiled from NL (6 vars, 2 softs).
#     Inspect with [optimize:code Planning]."

# Etape 2 : Inspecter le VVL produit par le LLM
[log [optimize:code Planning]]
# => Affiche un bloc [optimize:problem Planning [decision $alice_j1 ...] ...]
#    relisible et editable.

# Etape 3 : Valider le sous-ensemble
[log [optimize:validate Planning]]
# => "OK — 'Planning' compiles cleanly to CP-SAT (6 decision var(s))."

# Etape 4 : Resoudre
$sol = [optimize:solve Planning]
# => <{"alice_j1": 0, "alice_j2": 0, "bob_j1": 1, "bob_j2": 1,
#      "carol_j1": 1, "carol_j2": 1,
#      "__status__": "OPTIMAL", "__objective__": 4.0}>

# Etape 5 : Rapport humain
[log [optimize:report Planning]]
# => "Statut : OPTIMAL.
#       Alice : J1=non, J2=non (0 jour)
#       Bob   : J1=oui, J2=oui (2 jours)
#       Carol : J1=oui, J2=oui (2 jours)
#     Contraintes dures :
#       OK couverture J1 : 0+1+1 = 2 (>= 2 requis)
#       OK couverture J2 : 0+1+1 = 2 (>= 2 requis)
#       OK Alice : 0 jour (<= 1 requis)
#     Preferences :
#       OK Bob prefere le matin : Bob travaille J1
#       OK Carol prefere la fin : Carol travaille J2
#     Total : 4 jours travailles (minimum atteint sous contraintes)."

# Etape 6 : Raffinement post-hoc via [intent]
[intent "Alice doit travailler au moins un jour" "Planning"]
# => "Intent compiled to [assert ($alice_j1 + $alice_j2 >= 1)] and
#     added to problem 'Planning'."

# Etape 7 : Re-resolution avec la nouvelle contrainte
$sol2 = [optimize:solve Planning]
[log [optimize:report Planning]]
```

Ce pipeline en sept étapes est le **scénario nominal** d'utilisation. Trois choses méritent d'être soulignées.

Premièrement, l'inspection est essentielle. Après chaque appel à `[optimize "NL"]`, lire `[optimize:code]` permet de vérifier ce que le LLM a produit. Le LLM peut interpréter de travers une formulation ambiguë, oublier une contrainte, ou choisir un type de décision sous-optimal. La sortie reste **éditable** : on peut copier le code, le corriger, et le re-évaluer comme du VVL standard.

Deuxièmement, `[intent]` est le mécanisme de raffinement. Plutôt que de tout reformuler quand on veut ajouter une contrainte, on appelle `[intent "NL" "ProblemName"]`. Le LLM compile cette intention en **une seule** forme (`[soft ...]`, `[assert ...]`, `[maximize ...]` ou `[minimize ...]`) et l'ajoute au problème déjà enregistré, qui est automatiquement recompilé. Les décisions existantes ne sont pas re-parsées — c'est une garantie d'isolation qui évite le piège classique du « déclaré deux fois ».

Troisièmement, `[optimize:report]` produit un rapport contextuel. Le LLM reçoit la description NL d'origine, le corps VVL compilé, et le résultat JSON du solveur. Il **mappe les variables abstraites** aux entités du domaine (par exemple `$alice_j1` → « Alice Jour 1 »), vérifie chaque contrainte dure avec les valeurs effectives, et exprime la satisfaction des préférences en langage naturel — dans la même langue que la description d'origine.

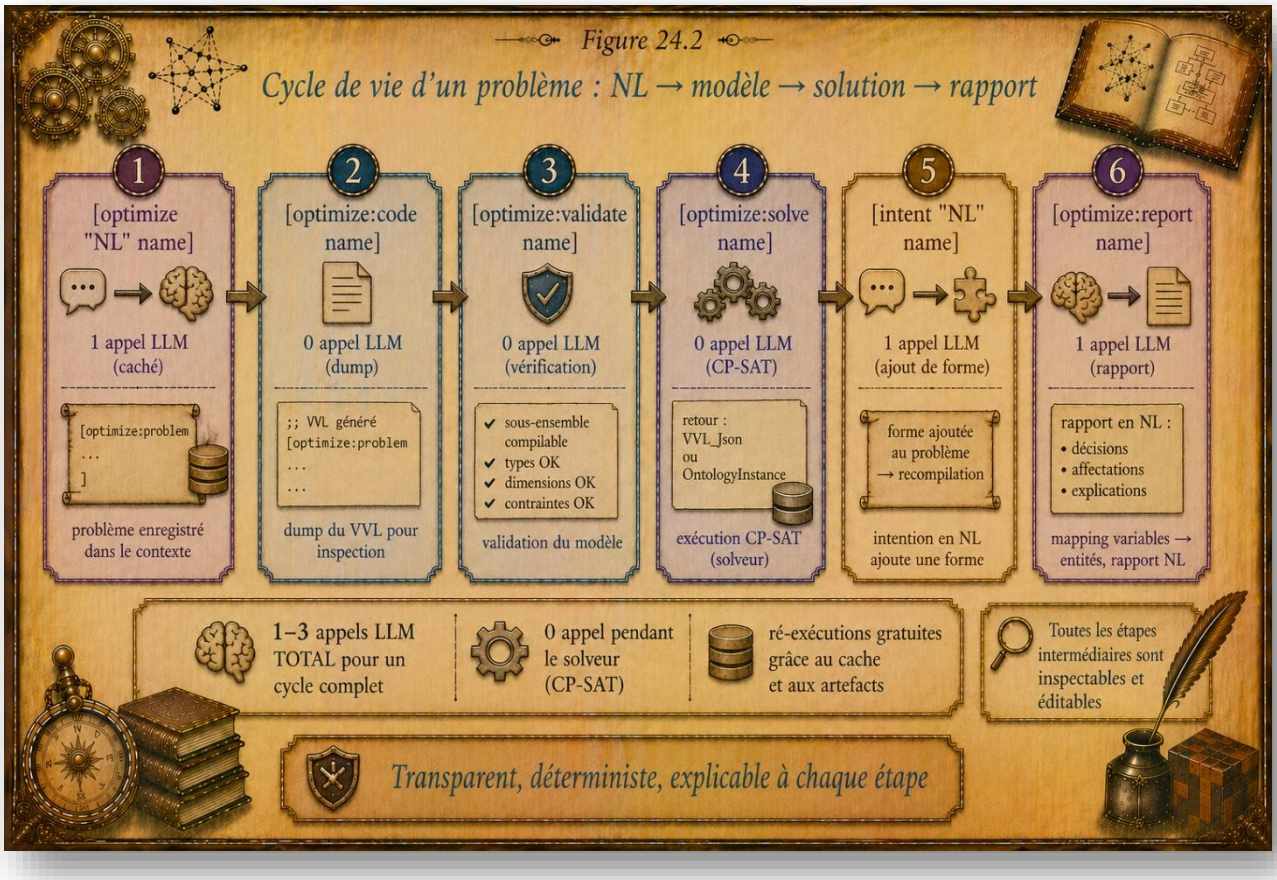


Figure 25.4 — *Le cycle de vie complet : NL en entrée, rapport en sortie, inspection à chaque étape*

25.5. Opérateurs flous résolus au compile-time

Les opérateurs flous présentés au chapitre 24 — ~&, ~|, ~!, ≈>, ~~ — sont admis par le compilateur `optimize`, mais avec une nuance fondamentale : **leur sémantique est résolue au compile-time**, jamais à l'intérieur du solveur. Cette discipline est ce qui permet de combiner la richesse expressive de la logique floue avec la performance déterministe du solveur entier.

Composition de l'objectif (mathématique du SCALE)

La constante `SCALE` (par défaut 100, configurable via `[scale N]`) contrôle la précision des poids et l'équilibre entre objectif primaire et contraintes molles. Le tableau suivant détaille la composition de l'objectif compilé selon les formes utilisées :

Situation	Objectif CP-SAT compilé
[maximize p] seul	Maximize(p)
[minimize p] seul	Minimize(p)
[maximize p] + softs	Maximize(SCALE·p + Σ int(w·SCALE)·t_i)
[minimize p] + softs	Minimize(SCALE·p − Σ int(w·SCALE)·t_i)
Softs uniquement	Maximize(Σ int(w·SCALE)·t_i)
Aucun objectif (ni soft)	Aucun objectif ; première solution faisable

Figure 25.5 — *Composition de l'objectif CP-SAT selon les formes du corps*

Chaque `t_i` est : le `BoolVar` réifié si l'expression molle est une comparaison ou un combinateur flou (contribution maximale = `w · SCALE`) ; ou directement l'expression linéaire si ce n'est pas une comparaison ; ou — cas spécial pour `[soft (~~)]` — une table de contribution précalculée par valeur d'Enum, injectée via `AddElement`. Un poids de `1.0` signifie « aussi important qu'une unité d'objectif primaire » ; les poids fractionnaires et négatifs sont supportés.

```
[optimize:problem TeamComp
    [decision $size  Integer [min 3] [max 10]]
    [decision $role  Enum    [values "junior" "senior" "manager"]]

    # Hard : taille minimale
    [assert ($size >= 5)]

    # Soft 1 : preference role senior, poids 1
    [soft ($role ~~ "senior") [weight 1]]

    # Soft 2 : preference taille pas trop grande, poids 0.5
    [soft ($size <= 7) [weight 0.5]]

    # Objectif primaire : maximiser la taille
    [maximize ($size)]
]

# Le compilateur produit l'objectif :
#   Maximize(100·size + 100·t_role + 50·t_size_le_7)
# ou t_role est une IntVar exposant la table degrés-de-similarite avec "senior"
# et t_size_le_7 est le BoolVar reifie de la comparaison.

$sol = [optimize:solve TeamComp]
```

25.6. Patterns avancés : multi-objectif, globales, multi-instance, paramétrique

Multi-objectif lexicographique

Quand plusieurs objectifs ont des **priorités** différentes (« d'abord minimiser le coût, puis maximiser la qualité une fois le coût optimal »), la forme `[lexicographic ...]` exécute des résolutions séquentielles. Chaque niveau optimise son objectif ; une fois l'optimum atteint, il est verrouillé comme contrainte dure pour le niveau suivant. Cette forme ne peut pas être combinée avec un `[maximize]` / `[minimize]` simple ni avec des `[soft]` (rejet explicite avec message clair).

```
[optimize:problem PLEX
    [decision $x Integer [min 0] [max 10]]
    [decision $y Integer [min 0] [max 10]]
    [assert (($x + $y) <= 10)]
    [lexicographic
        [maximize ($x)]              # niveau 1 : maximiser x d'abord
        [maximize ($y)]              # niveau 2 : puis maximiser y
    ]
]
# => x=10 (niveau 1 : max), y=0 (niveau 2 : max sous x+y<=10 avec x verrouille a 10)
```

Contraintes globales natives CP-SAT

Trois contraintes globales sont exposées directement : `[all-different ...]` mappe vers `AddAllDifferent` ; `[cumulative starts dur demands cap]` mappe vers `AddCumulative` (les intervalles sont synthétisés en interne) ; `[circuit [(from to bool) ...]]` mappe vers `AddCircuit`. Ces contraintes sont **plus efficaces** que leur encodage manuel via `[assert]` quand le solveur sait les exploiter — c'est typiquement le cas pour les problèmes d'ordonnancement et de routage.

```
# All-different : trois personnes, trois taches, chacun fait une tache distincte
[optimize:problem TaskAssign
    [decision $alice_task Integer [min 1] [max 3]]
    [decision $bob_task   Integer [min 1] [max 3]]
    [decision $carol_task Integer [min 1] [max 3]]
    [all-different $alice_task $bob_task $carol_task]
    [maximize ($alice_task + $bob_task + $carol_task)]
]
# => permutation de {1,2,3}, somme = 6
```

Multi-instance ontologique

Quand plusieurs instances d'une même classe ontologique participent au problème, la forme `[instance ClassName id]` émet des variables nommées de manière isolée. Pour une classe `Employee` avec un attribut `salary` flagué `[decision true]`, deux instances `senior` et `junior` produisent les variables `$senior.salary` et `$junior.salary`. Les contraintes croisées s'écrivent naturellement.

```
[ontology:define HRSchema
    [ontology:class Employee
        [attributes [salary Integer [min 30000] [max 120000] [decision true]]]
    ]
]

[optimize:problem PayPlan "HRSchema"
    [instance Employee senior]
    [instance Employee junior]
    [assert ($senior.salary >= ($junior.salary + 20000))]   # senior gagne >= 20k de plus
    [assert (($senior.salary + $junior.salary) <= 150000)]  # budget total
    [maximize ($senior.salary + $junior.salary)]
]
$payroll = [optimize:solve PayPlan]
# => <{"senior_salary": 85000, "junior_salary": 65000, "__status__": "OPTIMAL"}>
```

Paramétrique : la forme [for]

Les modèles d'optimisation réels génèrent souvent un grand nombre de variables suivant un schéma régulier (« une décision binaire par employé et par jour »). La forme `[for $var $list body]` permet une **expansion paramétrique au parse-time** : la liste est évaluée dans le contexte VVL appelant, la variable de boucle est liée pour chaque itération, et les formes du corps sont émises avec interpolation `${...}` résolue dans leurs chaînes. Le résultat est entièrement inspectable via `[optimize:code]` après expansion — il n'y a pas de `[for]` résiduel.

```
$days  = [list "mon" "tue" "wed" "thu" "fri"]
$staff = [list "alice" "bob" "carol"]

[optimize:problem ShiftPlan
    # 1. Une decision binaire par (employe, jour) : 3 x 5 = 15 vars generees
    [for $e $staff
        [for $d $days
            [decision "works_${e}_${d}" Integer [min 0] [max 1]]
        ]
    ]

    # 2. Couverture minimale par jour
    [for $d $days
        [assert ([+ $works_alice_${d} $works_bob_${d} $works_carol_${d}] >= 2)]
    ]

    # 3. Contrainte personnelle : Alice maximum 3 jours par semaine
    [assert ([+ $works_alice_mon $works_alice_tue $works_alice_wed
                $works_alice_thu $works_alice_fri] <= 3)]

    # 4. Multi-objectif : d'abord minimiser les jours d'Alice,
    #    puis maximiser ceux de Bob qui aime travailler
    [lexicographic
        [minimize ([+ $works_alice_mon $works_alice_tue $works_alice_wed
                      $works_alice_thu $works_alice_fri])]
        [maximize ([+ $works_bob_mon   $works_bob_tue   $works_bob_wed
                      $works_bob_thu   $works_bob_fri])]
    ]
]

# [optimize:code ShiftPlan] montre le corps APRES expansion,
# avec les 15 [decision], les 5 [assert] de couverture, etc.

$sol = [optimize:solve ShiftPlan]
```

`[optimize:code]` après expansion : la garantie d'audit. Pour un problème paramétrique, la commande `[optimize:code ShiftPlan]` ne renvoie pas le code source contenant des `[for]` — elle renvoie le **corps après expansion**, c'est-à-dire la liste exacte des `[decision]`, `[assert]`, `[soft]` qui seront effectivement compilés vers CP-SAT. Pour un planning de 3 employés × 5 jours, on voit défiler les 15 décisions binaires, les 5 contraintes de couverture, les autres formes, sans aucun `[for]` résiduel. C'est précisément ce qui sera envoyé au compilateur. Pour les modèles générés via `[optimize "NL"]` qui produisent des `[intent]` internes, ces intentions sont également remplacées par leurs formes expansées dans la sortie. Le développeur peut ainsi **auditer le modèle réellement compilé**, le copier, le coller dans une nouvelle session, le réviser à la main — la chaîne de transformation est entièrement traçable.

Pour réutiliser un schéma sur plusieurs jeux de données (par exemple, un planning hebdomadaire répété chaque semaine), il suffit d'encapsuler le tout dans une lambda VVL — pas besoin d'un cycle de vie paramétrique propre :

```
$build = [[$emps $days $name] => [optimize:problem ${name}
    [for $d $days [for $e $emps
        [decision "works_${e.name}_${d}" Integer [min 0] [max 1]]
    ]]
    ...
]]

[$build $emps_w12 $days "Week12"]
[$build $emps_w13 $days "Week13"]
```

Diagnostic d'infaisabilité

Quand `[optimize:solve]` retourne `INFEASIBLE`, la commande `[optimize:why-infeasible name]` isole le **sous-ensemble minimal** d'asserts dont l'intersection est insatisfiable. Chaque `[assert]` est encapsulé au compile-time dans un littéral d'hypothèse CP-SAT ; sur infaisabilité, le solveur appelle `SufficientAssumptionsForInfeasibility()` et le compilateur remappe les indices vers les expressions sources.

```
[optimize:problem PINF
    [decision $x Integer [min 0] [max 100]]
    [assert ($x >= 80)]
    [assert ($x <= 50)]
    [maximize ($x)]
]
$inf = [optimize:solve PINF]
# => "no solution (status=INFEASIBLE). Run [optimize:why-infeasible PINF] ..."

[log [optimize:why-infeasible PINF]]
# => "=== Conflict in 'PINF' : 2 constraint(s) ===
#       1. [assert] (from problem): $x >= 80
#       2. [assert] (from problem): $x <= 50
#     Removing any one of these makes the problem feasible."
```

25.7. Routing géographique : geo:distance, geo:haversine, route:matrix

Une famille importante de problèmes d'optimisation industriels relève du **routing** : tournées de livraison, affectation de techniciens à des sites, planification d'interventions sur le terrain. Ces problèmes partagent une caractéristique commune — leur fonction objectif comporte des **distances**

géographiques entre adresses, qu'il faut traduire en coefficients entiers exploitables par CP-SAT. VVL répond à ce besoin avec trois builtins du module `geo/` que le compilateur `optimize` sait **replier au compile-time**.

Repli au compile-time des builtins géographiques purs

Le compilateur `optimize` reconnaît une catégorie particulière de builtins — les builtins **purs et déterministes** — qu'il peut évaluer au moment de la compilation et dont il injecte le résultat comme constante dans le modèle CP-SAT. Trois builtins géographiques entrent dans cette catégorie :

Builtin	Signature	Sémantique au compile-time
geo:distance	[geo:distance "src" "dst"]	Distance routière en mètres entre deux adresses textuelles
geo:haversine	[geo:haversine $A $B]	Distance orthodromique entre deux paires (lat, lon)
route:matrix	[route:matrix [list src...] [list dst...]]	Matrice de distances source × destination

Figure 25.6 — *Builtins géographiques repliés au compile-time*

Lorsqu'une de ces expressions apparaît dans le corps d'un `[optimize:problem ...]` — typiquement comme coefficient dans un `[minimize ...]` ou un `[soft ...]` —, le compilateur **l'évalue une seule fois** lors de la phase de compilation, mémorise le résultat dans `.geo_cache.json`, et l'injecte comme entier dans le modèle CP-SAT. Le solveur ne voit jamais d'appel à un service de géocodage ; il voit des constantes entières. Cette discipline est cohérente avec celle adoptée pour `~~`, `[§ ...]` et `[intent ...]` au compile-time : aucun aller-retour réseau pendant la résolution.

```
# Distance simple : mise en cache de la 1ere requete Google Geocoding
[geo:distance "Paris, France" "Lyon, France"]
# => 463000  (metres, en cache : .geo_cache.json)

# Haversine : pas de geocoding, calcul pur sur coordonnees
[geo:haversine [list 48.8566 2.3522] [list 45.7640 4.8357]]
# => 391500  (metres, distance orthodromique)

# Matrice : prefetch de toutes les paires en une passe
$sites = [list "Lyon" "Marseille" "Toulouse"]
$techs = [list "Bordeaux" "Nantes"]
$dmat  = [route:matrix $techs $sites]
# => liste imbriquee : [[d_bord_lyon d_bord_mars d_bord_toul]
#                       [d_nant_lyon d_nant_mars d_nant_toul]]
```

Le **cache `.geo_cache.json`** mérite une mention dédiée : il stocke les résultats des appels au service de géocodage (Google Geocoding API par défaut) sous une clé hash de l'adresse normalisée, ce qui rend les ré-exécutions du même modèle gratuites et offline. Le tableau ci-dessous récapitule l'ensemble des caches du sous-système :

Cache	Contenu	Invalidation
.semantic_cache.json	Résultats de `~~`, `[§...]`, semantic_binop	Hash de la paire normalisée + version du prompt
.optimize_nl_cache.json	VVL généré par `[optimize "NL"]` et `[intent]`	Hash de NL + nom + version du prompt système
.geo_cache.json	Résultats de `geo:distance`, `route:matrix`	Hash de l'adresse normalisée

Figure 25.7 — *Les trois caches du sous-système optimize*

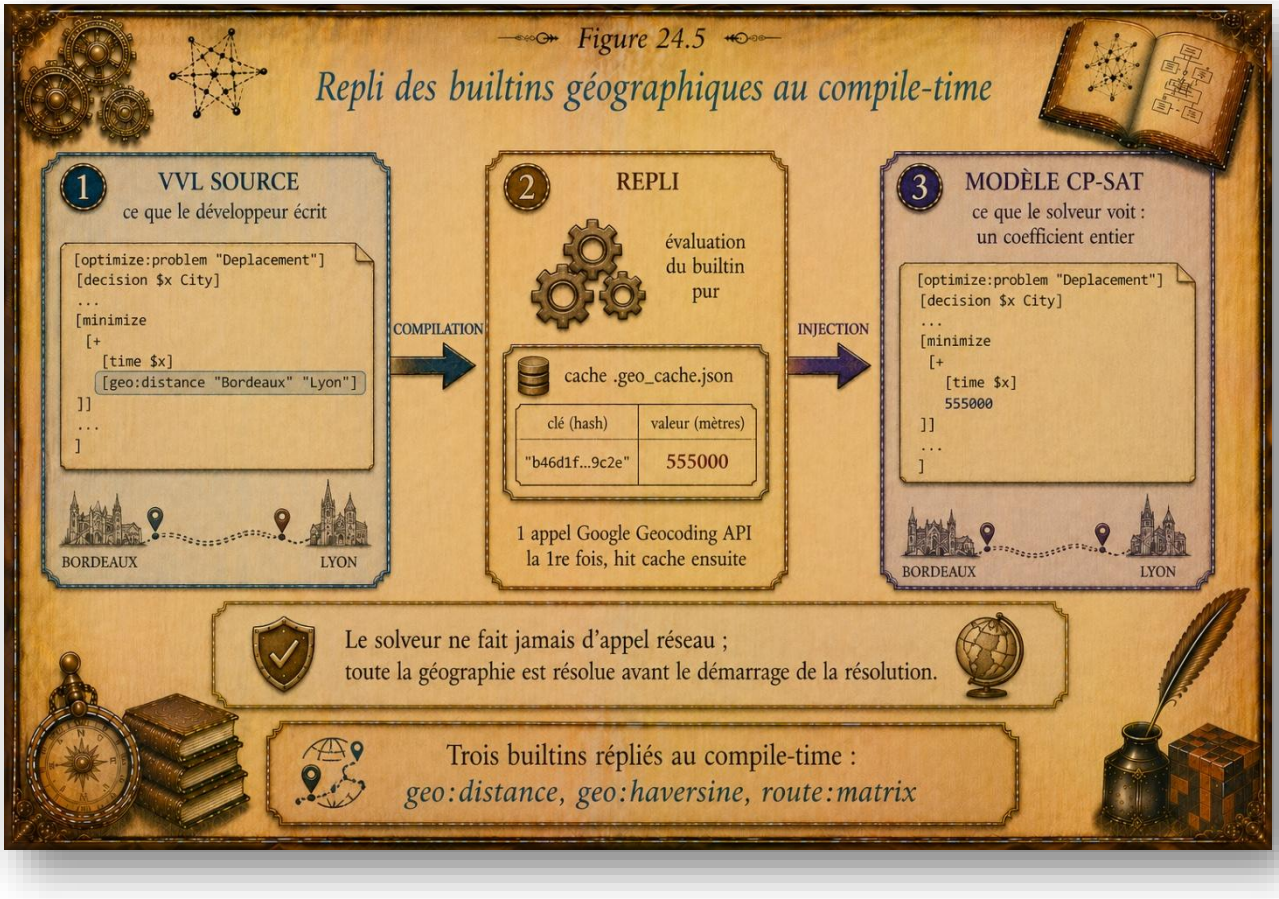


Figure 25.8 — *Le repli au compile-time : du builtin VVL au coefficient entier CP-SAT*

Pattern paramétrique : tournée de techniciens

Le pattern naturel combine `[route:matrix]` pour précharger les distances, puis `[for]` pour générer les variables et contraintes du modèle. Voici un cas réel d'**affectation de techniciens à des sites** où l'on minimise le kilométrage total :

```
# 1. Prefetch des distances en une passe (geocoding cache)
$techs = [list "Bordeaux" "Nantes" "Lille"]
$sites = [list "Lyon" "Marseille" "Toulouse"]
$dmat  = [route:matrix $techs $sites]

# 2. Modele d'affectation : 1 technicien par site, 1 site par technicien max
[optimize:problem TourneeTechs
    # Variable binaire : tech t affecte au site s ?
    [for $t $techs
        [for $s $sites
            [decision "x_${t}_${s}" Integer [min 0] [max 1]]
        ]
    ]

    # Chaque site doit etre couvert par exactement 1 technicien
    [for $s $sites
        [assert ([+ $x_Bordeaux_${s} $x_Nantes_${s} $x_Lille_${s}] == 1)]
    ]

    # Chaque technicien fait au plus 1 affectation
    [for $t $techs
        [assert ([+ $x_${t}_Lyon $x_${t}_Marseille $x_${t}_Toulouse] <= 1)]
    ]

    # Objectif : minimiser le kilometrage total
    # Le compilateur replie [geo:distance ...] en coefficients entiers
    [minimize ([+ ([geo:distance "Bordeaux" "Lyon"]      * $x_Bordeaux_Lyon)
                  ([geo:distance "Bordeaux" "Marseille"] * $x_Bordeaux_Marseille)
                  ([geo:distance "Bordeaux" "Toulouse"]  * $x_Bordeaux_Toulouse)
                  ([geo:distance "Nantes"   "Lyon"]      * $x_Nantes_Lyon)
                  ([geo:distance "Nantes"   "Marseille"] * $x_Nantes_Marseille)
                  ([geo:distance "Nantes"   "Toulouse"]  * $x_Nantes_Toulouse)
                  ([geo:distance "Lille"    "Lyon"]      * $x_Lille_Lyon)
                  ([geo:distance "Lille"    "Marseille"] * $x_Lille_Marseille)
                  ([geo:distance "Lille"    "Toulouse"]  * $x_Lille_Toulouse)])]
]

# 3. Au moment de la compilation :
#  - chaque [geo:distance ...] est evalue une fois (cache .geo_cache.json),
#  - le resultat (metres) devient un coefficient entier dans le modele CP-SAT,
#  - le solveur ne voit que des entiers et ne fait aucun appel reseau.

$sol = [optimize:solve TourneeTechs]
```

Prompt NL spécialisé routing

La commande `[optimize "NL" name]` ne traite pas les problèmes de routing comme un cas générique : le prompt système contient des **instructions spécialisées** pour détecter les indices d'un problème de déplacement (mots-clés *tournée*, *adresses*, *kilomètres*, *itinéraire*, *livreur*, *technicien*, *visiter*, *distance*, etc.) et générer automatiquement la structure adaptée — variables binaires d'affectation, contraintes de couverture, et coûts exprimés via `[geo:distance ...]` plutôt que comme constantes magiques.

```
# Le LLM detecte les indices "tournee", "adresses", "minimiser km"
# et genere directement un modele d'affectation avec [geo:distance].

[optimize "Trois techniciens — basés à Bordeaux, Nantes et Lille — doivent
           visiter trois sites clients à Lyon, Marseille et Toulouse.
           Chaque site doit recevoir un technicien et chaque technicien
           ne fait qu'une visite. Minimiser le kilometrage total."
         "TourneeTechs"]
# => "Problem 'TourneeTechs' compiled from NL (9 vars, 6 asserts).
#     Inspect with [optimize:code TourneeTechs]."

[log [optimize:code TourneeTechs]]
# => Affiche un bloc avec :
#      - 9 [decision "x_<tech>_<site>" Integer [min 0] [max 1]]
#      - 6 [assert] (3 sites couverts une fois + 3 techs <= 1 visite)
#      - 1 [minimize ...] avec 9 termes [geo:distance ...] * binaire
```

Cette spécialisation transforme la commande `[optimize "NL"]` d'un compilateur générique en **assistant routier** dès qu'il en détecte les indices. L'utilisateur reste libre d'inspecter le résultat, de l'éditer, ou de basculer sur le mode body manuel s'il veut un contrôle plus fin — mais dans le cas typique, le pattern « décrire la tournée en français → obtenir un modèle d'affectation correct → résoudre » fonctionne du premier coup.

Trois disciplines complètent ce pattern. **Première discipline** : préférer `[route:matrix]` à des appels `[geo:distance]` répétés quand on construit un modèle paramétrique large — la matrice est précalculée en une passe, ce qui réduit le nombre d'aller-retours au service de géocodage. **Deuxième discipline** : penser à `[geo:haversine]` plutôt que `[geo:distance]` quand on dispose déjà des coordonnées géographiques — c'est un calcul pur, instantané, sans dépendance à un service externe. **Troisième discipline** : versionner `.geo_cache.json` avec le code du modèle si l'on veut une reproductibilité bit-à-bit ; sans cette précaution, un changement d'API du service de géocodage ou une révision des bases d'adresses peut faire dériver les coefficients.

25.8. Trois niveaux d'expression

L'utilisateur du sous-système `optimize` dispose de trois niveaux d'expression, qu'il choisit selon le contexte. Ces niveaux ne s'opposent pas — ils peuvent même cohabiter dans une même session — mais chacun a son moment :

Niveau	Forme	Cas d'usage	Coût LLM
1 — NL pur	[optimize "NL" name]	Prototype rapide, exploration métier	1 appel (caché)
2 — Body manuel	[optimize:problem ... body]	Contrôle fin, contraintes complexes	0 appel
3 — Paramétrique	[for] / [instance] / lambda VVL	Problèmes structurés à grande échelle	0 appel

Figure 25.9 — *Trois niveaux d'expression d'un problème d'optimisation*

Le niveau 1 (NL pur) est idéal pour explorer un problème métier dont on n'a pas encore stabilisé les contraintes. Le LLM produit un premier modèle qu'on inspecte, valide, raffine via `[intent]`. Une fois le modèle stable, on peut `[optimize:code]` pour récupérer le VVL et migrer vers le niveau 2 si l'on veut un contrôle plus fin.

Le niveau 2 (body manuel) est le mode de production pour les problèmes dont la structure est connue. On écrit `[optimize:problem ... ]` directement, avec ses décisions, asserts et softs. Aucun appel LLM, performance maximale, lisibilité optimale pour le développeur expérimenté.

Le niveau 3 (paramétrique) s'impose quand la structure du problème dépend des données : nombre d'employés, nombre de jours, nombre de tâches, tous variables. La forme `[for]` génère mécaniquement les décisions et contraintes ; la forme `[instance]` permet le multi-instance ontologique ; une lambda VVL réutilise tout cela à travers plusieurs jeux de données. Ce niveau atteint des problèmes industriels avec des centaines voire des milliers de variables, sans quitter VVL

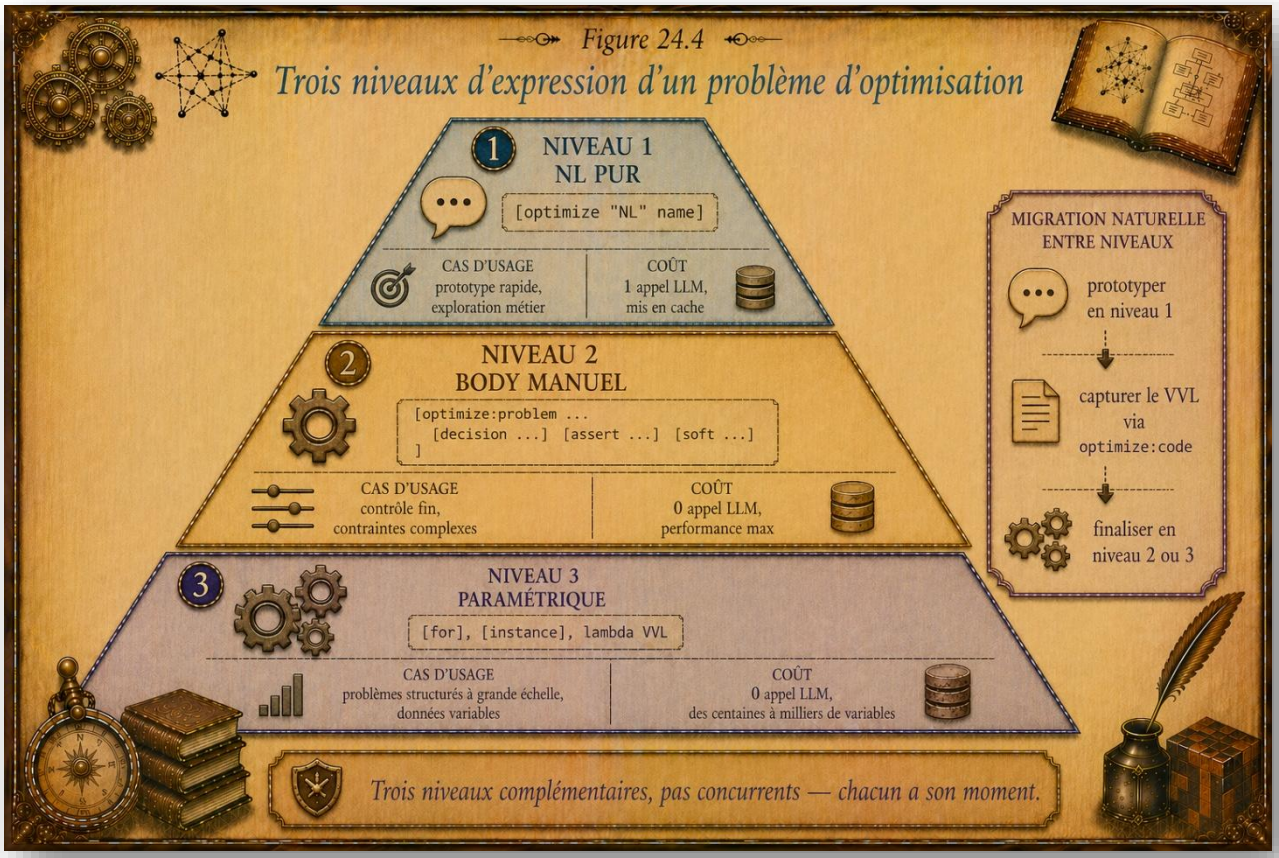


Figure 25.10 — *La pyramide des trois niveaux d'expression : NL, manuel, paramétrique*

25.9. Synthèse : l'optimisation au sein de l'oscillation neuro-symbolique

Le sous-système `optimize` complète l'éventail des outils neuro-symboliques de VVL avec une troisième famille de problèmes — l'optimisation sous contraintes —, qui s'ajoute à la classification (chapitres 13, 19) et à la décision règle-par-règle (chapitres 12, 21). Ces trois familles partagent la même philosophie : déléguer au LLM la part créative et conceptuelle, garder pour le runtime symbolique la part rigoureuse et déterministe.

Famille	Mécanisme principal	Solveur	Garantie
Classification	Opérateurs Zalta `~~`, archétypes	Score 0–1 par syndrome	Match cliniquement attendu

Famille	Mécanisme principal	Solveur	Garantie
Décision en cascade	Règles avec priorité, infer-fixpoint	Forward chaining	Règles auditables
Optimisation	CP-SAT (OR-Tools)	Modèle entier linéaire	Optimum prouvé

Figure 25.11 — *Trois familles de problèmes neuro-symboliques*

Trois propriétés transversales émergent. **Première propriété : le LLM résout au compile-time, jamais à l'intérieur des boucles critiques.** L'extraction `ontology:fromText` se fait une fois par instance, le matching ~~ est mémoïsé par paire, l'expansion `[optimize "NL"]` est cachée par hash. Les coûts LLM sont marginaux et prévisibles ; les performances en steady-state sont déterministes.

Deuxième propriété : l'inspection est constante. Que ce soit `[optimize:code]` pour le VVL généré, `[optimize:explain]` pour le modèle CP-SAT compilé, `[optimize:validate]` pour la conformité au sous-ensemble compilable, ou `[optimize:why-infeasible]` pour le diagnostic d'infaisabilité, le développeur n'est jamais devant une boîte noire. Cette transparence est ce qui distingue un système de production d'un prototype.

Troisième propriété : la composabilité avec le reste du langage. Un problème d'optimisation peut consommer des données de `[CSV ...]`, `[JSON ...]`, `[ontology:fromText ...]`, `[geo:distance ...]`, `[route:matrix ...]` ; il peut produire un résultat qui devient une instance ontologique committée dans la KB Prolog ; il peut être encapsulé dans un keyword utilisateur, persisté dans un workspace, exposé via un agent. Le sous-système ne vit pas en îlot — il participe au tissu cohérent que forment les autres parties du langage.

Le **chapitre 26 poursuit l'exploration de l'optimisation avec les algorithmes génétiques. Là où la programmation par contraintes exige un modèle exprimable en variables discrètes et en contraintes dures, l'approche évolutionnaire s'applique aux objectifs qui ne se mesurent qu'a posteriori, sur des structures que l'on ne sait pas dériver. Les deux chapitres se complètent plus qu'ils ne se concurrencent : contraintes dures et linéaires d'un côté, fonction objectif opaque de l'autre**.

26. Algorithmes génétiques : optimisation boîte-noire de structures discrètes

Mots-clés : algorithmes génétiques, optimisation boîte-noire, raffinement de règles induites, fitness holdout, opérateurs déterministes, mutations LLM, contre-exemples, boucle adversariale, multi-objectif, frontière de Pareto, élitisme, plateau, content-hash, FitnessCache

Le chapitre 25 a présenté l'optimisation par contraintes sur backend CP-SAT — un solveur déterministe qui résout exactement les problèmes *dérivables* à *contraintes dures* : ordonnancement, allocation de ressources, planification. Pour ces problèmes, on dispose d'un encodage propre en variables booléennes ou entières, et le solveur explore l'espace par branch-and-bound. Mais une grande part de l'optimisation que rencontre un développeur neuro-symbolique ne ressemble pas à cela. Quand on raffine un ensemble de **règles métier induites** par `induction.py` (chapitre 17), un **prompt d'agent** (chapitre 22), ou une **policy de sandbox** (chapitre 23), la fonction objectif n'est pas dérivable — elle vient d'évaluations holdout, de votes humains, ou de tests sur des cas réels. Le solveur CP-SAT n'a aucune prise là-dessus.

Ce chapitre 26 introduit le module **algorithmes génétiques** qui répond à ce manque. C'est un moteur d'optimisation **boîte-noire** sur structures discrètes — populations d'individus représentés comme des genomes JSON, sélection par tournament, crossover et mutation sur les genomes, élitisme, détection de plateau. Le module a été conçu d'emblée pour s'intégrer au **protocole d'atomes** du chapitre 23 : chaque `Individual`, chaque `Population`, chaque `CounterStore` est un atome introspectable, hashable, traçable, et **gateable par policy**. L'oscillation neuro-symbolique du livre trouve ici sa version la plus dynamique : un agent LLM qui propose des mutations stochastiques, une fonction de fitness symbolique qui les juge, une boucle qui converge vers un Pareto-optimum auditable.

Le chapitre suit la séquence : motivation et complémentarité avec CP-SAT (26.1), le modèle évolutionnaire standard (26.2), vue d'ensemble des quatre phases du module (26.3), Phase 1 — opérateurs déterministes et fitness holdout (26.4), Phase 2 — mutations LLM avec budget et cost tracking (26.5), Phase 3 — boucle adversariale par contre-exemples (26.6), Phase 4 — multi-objectif et frontière de Pareto (26.7), architecture interne et synergie avec le protocole d'atomes (26.8), pattern end-to-end de production et synthèse (26.9).

> **Périmètre. Le module algorithmes génétiques décrit dans ce chapitre fait partie du serveur courant. Ses quatre phases sont couvertes par 48 tests et illustrées par cinq tutoriels exécutables, de vvl/genetic/01-synthetic-classification.vvl à 05-end-to-end.vvl. Comme le protocole d'atomes du chapitre 23, il est additif et opt-in : aucun programme VVL antérieur n'est affecté tant que le module n'est pas explicitement utilisé.**

26.1. Pourquoi un solveur évolutionnaire complémentaire à CP-SAT

Le chapitre 25 a déroulé un argument fort : pour les problèmes industriels qui s'expriment proprement en variables booléennes ou entières et en contraintes linéaires (ordonnancement d'ateliers, packing, routing, planification), CP-SAT est le bon outil. Il garantit l'optimum global, sait certifier l'infaisabilité, et passe à l'échelle de plusieurs milliers de variables. Cette puissance vient cependant d'un prix d'entrée : l'**encodage**. Un problème ne devient CP-SAT-traitable qu'une fois transcrit dans son langage — ses domaines, ses contraintes, sa fonction objectif *linéaire*. Pour beaucoup de problèmes que rencontre un développeur neuro-symbolique, cet encodage n'existe tout simplement pas.

Considérons trois exemples concrets. **Premier exemple** : on dispose d'un classifieur de candidats produit par induction LLM (chapitre 17) — une dizaine de règles SI-ALORS, chacune avec un seuil et un prédicat. Sur un dataset holdout, ces règles obtiennent 78 % de précision. On voudrait monter à 90 %. Le seuil optimal d'une règle individuelle dépend de l'interaction avec les autres règles, et la précision se mesure par évaluation sur des données — il n'y a pas de gradient à descendre, pas de contrainte linéaire à poser. **Deuxième exemple** : un agent LLM (chapitre 22) reçoit un prompt système qui définit son comportement. Le prompt actuel produit des réponses moyennes. On voudrait l'améliorer, mais l'espace des prompts est combinatoire et évalué par des humains qui jugent la qualité a posteriori. **Troisième exemple** : on a écrit une policy de sandbox (chapitre 23) qui rejette 5 % des requêtes légitimes en plus des malveillantes. On voudrait raffiner ses règles allow/deny pour réduire les faux positifs, mais la fitness se mesure sur un trafic réel, pas en algèbre.

Ces trois cas partagent une structure commune : **un espace de candidats discret et combinatoire**, **une fitness évaluée par holdout ou jugement humain**, **aucun gradient analytique**. C'est exactement le territoire où les **algorithmes génétiques** excellent depuis cinquante ans. La métaphore biologique — populations, sélection, crossover, mutation — opérationnalise une exploration parallèle de l'espace, où des « bons gènes » se recombinent et où des perturbations aléatoires explorent localement. Pour des espaces structurés comme des règles ou des prompts, le GA produit des résultats utiles là où CP-SAT n'a même pas de prise.

Le module GA présenté ici ne **remplace** pas `[optimize:problem]` du chapitre 25 — il le complète. Quand vos contraintes sont dures et linéaires, restez avec CP-SAT. Quand votre objectif n'est mesurable qu'*a posteriori* sur un dataset, passez au GA. Les deux solveurs sont disponibles côte à côte, et un même programme peut les chaîner — par exemple résoudre l'ordonnancement par CP-SAT puis raffiner les règles d'éligibilité par GA.



Figure 26.1 — *Carte de l'optimisation neuro-symbolique : CP-SAT pour le dérivable, GA pour le boîte-noire*

26.2. Le modèle évolutionnaire standard

Le module suit fidèlement le manuel — pas d'innovation algorithmique, l'apport est dans l'**intégration** avec le reste de VVL. La boucle se décompose en sept étapes.

(1) Population initiale. N individus, chacun portant un genome qui encode une solution candidate. Pour le cas de classification, un genome est une liste JSON de règles ; pour un prompt, ce serait une chaîne ; pour une policy, une liste de règles allow/deny. La diversité initiale conditionne tout l'avenir — on évite les populations dégénérées de moins de 10 individus.

(2) Évaluation. Chaque individu reçoit un score de fitness — scalaire (Phase 1, 2, 3) ou tuple (Phase 4 multi-objectif). Le score est **mémoïsé** sur le `content_hash` du genome (capability `Identifiable` du protocole d'atomes) ; un individu identique à la génération suivante n'est pas réévalué. Cette mémoïsation est cruciale en Phase 2 où les évaluations comportent des appels LLM coûteux.

(3) Sélection. Tournament k=3 par défaut. Pour fitness scalaire, le tournoi sélectionne le maximum. Pour fitness tuple, on utilise la **dominance de Pareto non-dominée** : un individu domine un autre s'il est meilleur sur au moins un objectif et pas pire sur les autres ; le tournoi compte les dominations et tranche aléatoirement les ex æquo.

(4) Crossover. Splice : on prend une portion du genome A (du début à un point de coupe) et la queue du genome B. Le ratio de coupe est tiré aléatoirement dans [0.3, 0.7] — biais central pour éviter les coupures triviales (tout-A ou tout-B).

(5) Mutation. Le moteur tire aléatoirement un opérateur dans un pool. En **Phase 1** (déterministes uniquement), le pool contient `drop` (supprime une règle aléatoire) et `threshold-tweak` (perturbe gaussiennement un seuil numérique). En **Phase 2** (LLM opt-in), s'ajoutent `llm-rephrase` (reformule un prédicat textuellement) et `llm-induce-on-failure` (relance `induction.py` sur les exemples mal classifiés du holdout). Les opérateurs LLM sont **préemptés** par les déterministes une fois `budget_llm_calls` épuisé.

(6) Élitisme. Les `elitism` meilleurs individus de la génération courante sont conservés tels quels dans la suivante — protection contre la régression accidentelle. Défaut : 2.

(7) Itération. Jusqu'à atteindre `generations` ou détecter un **plateau** — la meilleure fitness reste stable sur `plateau_window` générations consécutives, signal qu'on a probablement convergé. L'early-stop évite les tours inutiles.

Toute la boucle est **reproductible** si on fournit une `seed` : tous les opérateurs prennent un `random.Random` explicite, et l'ordre d'itération est déterministe. Cette reproductibilité est essentielle pour l'audit (« quelle évolution a produit cette règle ? ») et pour le debugging (relancer une évolution identique avec une nouvelle fonction de fitness).

26.3. Vue d'ensemble : les quatre phases du module

Le module est structuré en quatre phases incrémentales, chacune ajoutant une capacité tout en restant rétro-compatible avec les précédentes. La progression suit le degré d'investissement : Phase 1 sans appel LLM, Phase 2 avec budget LLM optionnel, Phase 3 avec capture des corrections humaines, Phase 4 avec optimisation multi-objectif. Un déploiement industriel typique commence en Phase 1 (rapide, gratuit, déterministe), bascule en Phase 2 quand la simple perturbation déterministe a saturé, ajoute Phase 3 quand la production accumule des corrections, et passe en Phase 4 quand on veut exposer explicitement les compromis (précision contre parsimonie contre coût).

Phase	Capacité	Coût opérationnel	Cas d'usage
1	Opérateurs `drop`, `threshold-tweak` + fitness holdout	CPU pur, pas d'appel LLM	Raffinement initial d'un classifieur induit ; baseline reproductible
2	Mutations LLM `llm-rephrase`, `llm-induce-on-failure` + cost tracking	LLM-calls bornés par budget	Quand le déterministe sature ; explorer des reformulations sémantiques
3	`CounterStore` + `fit-with-counter` → boucle adversariale	Aucun coût additionnel (corrections déjà disponibles)	Apprentissage continu après mise en production ; corrections humaines
4	Fitness vectorielle + tournament Pareto + `genetic:pareto`	Identique à phase utilisée pour les sous-objectifs	Exposer les compromis precision / parsimony / cost ; choix éclairé

Figure 26.2 — *Les quatre phases du module GA et leur positionnement opérationnel*

Les sections suivantes parcourent chaque phase avec un exemple complet tiré du tutoriel correspondant (`vvl/genetic/0N-*.vvl`). On suppose que le lecteur a parcouru au moins le chapitre

23 sur le protocole d'atomes — l'introspection des individus produits est un appel direct à `[atom:trace]` et à `[atom:axes]`.

26.4. Phase 1 — Opérateurs déterministes et fitness holdout

Le cas d'école : un dataset synthétique où le label dépend d'un seuil caché (`label = "high" si x > 50 sinon "low"`), et une population initiale de cinq règles candidates avec des seuils mal calibrés. Le GA doit converger vers le seuil 50.

```
# 1) Dataset étiqueté
$dataset = [JSON '[
   {"x":  5,  "label": "low"},   {"x":  25, "label": "low"},
   {"x":  45, "label": "low"},   {"x":  60, "label": "high"},
   {"x":  80, "label": "high"},  {"x":  95, "label": "high"}
]']

# 2) Population initiale — 5 règles avec seuils dispersés
$g1 = [JSON '[{"field":"x","op":">","threshold":10,"predicts":"high"}]']
$g2 = [JSON '[{"field":"x","op":">","threshold":30,"predicts":"high"}]']
$g3 = [JSON '[{"field":"x","op":">","threshold":50,"predicts":"high"}]']
$g4 = [JSON '[{"field":"x","op":">","threshold":70,"predicts":"high"}]']
$g5 = [JSON '[{"field":"x","op":">","threshold":90,"predicts":"high"}]']
$pop = [genetic:population $g1 $g2 $g3 $g4 $g5]

# 3) Fitness holdout : précision sur un split aléatoire
$fit = [JSON '{
   "kind":          "classification",
   "dataset":       <le dataset ci-dessus>,
   "label_field":   "label",
   "default_label": "low",
   "seed":          42
}']

# 4) Évolution déterministe (15 générations, mutations gratuites)
$opts = [JSON '{
   "generations":    15,   "mutation_rate":  0.5,
   "crossover_rate": 0.7,  "elitism":        2,
   "seed":           0
}']
$evolved = [genetic:evolve $pop $fit $opts]
$best = [genetic:best $evolved]

[log [genetic:fitness $best]]
# → 1.0   (le GA a découvert le seuil 50, précision parfaite sur holdout)
[log [atom:trace $best].lineage]
# → {generation: 12, parents: ["...", "..."], mutation_op: "threshold-tweak",
#    cumulative_cost: 0.0}
```

Trois choses sont remarquables dans cet exemple. **Premièrement**, le coût total est nul — pas un seul appel LLM. Le module est utilisable sur des datasets sans budget API. **Deuxièmement**, l'`$best` produit est lui-même un atome — `[atom:trace $best]` expose la lignée complète depuis la génération 0, l'opérateur de mutation qui l'a engendré, et son hash. C'est la **traçabilité gratuite** héritée du chapitre 23. **Troisièmement**, le résultat est **reproductible** : avec la même `seed`, deux exécutions produisent exactement la même évolution — propriété essentielle pour les tests automatisés et pour les audits réglementaires.

La fonction de fitness `classification` évalue chaque règle en l'appliquant aux records du holdout (par défaut 20 % du dataset, contrôlable par `holdout_split`) et en comptant la précision. Pour les multi-

règles, la première règle qui matche gagne ; si aucune ne matche, on retombe sur `default_label`. Cette sémantique simple suffit pour la plupart des classifieurs symboliques produits par `induction.py`.

26.5. Phase 2 — Mutations LLM avec budget et cost tracking

Quand les opérateurs déterministes saturent — l'`threshold-tweak` a exploré tous les seuils raisonnables, le `drop` a éliminé les règles redondantes, et la fitness ne progresse plus — il reste un type de transformation que seul un LLM peut faire : **réécrire un prédicat différemment** (`llm-rephrase`), ou **inférer une nouvelle règle** sur les exemples mal classifiés (`llm-induce-on-failure`). Ces opérateurs sont coûteux (chaque appel = un round-trip LLM) et stochastiques — il faut les contrôler strictement.

Le module les active via le flag `use_llm_mutations` et impose un budget strict avec `budget_llm_calls`. Une fois le budget atteint, le moteur se replie automatiquement sur les opérateurs déterministes — la boucle continue, simplement sans appel LLM. C'est la propriété de **dégradation gracieuse** déjà rencontrée au chapitre 22 sur les agents : mieux vaut une exécution complète sans appels LLM en fin qu'une exception qui interrompt l'évolution.

```
$opts = [JSON '{
   "generations":       20,
   "use_llm_mutations": true,
   "budget_llm_calls":  100,
   "seed":              42
}']
$evolved = [genetic:evolve $pop $fit $opts]
$best = [genetic:best $evolved]

# Inspection des coûts cumulés via le protocole d'atomes
[atom:axes $best]
# → {payload_kind: "value", effect_kind: "pure",
#    determinism: "stochastic",          # parce que mutations LLM dans la lignée
#    provenance: "induction",            # phase initiale + raffinement
#    cost_hint: 0.024,                   # USD cumulés pour CET individu
#    latency_hint: 850, ...}

[atom:trace $best].lineage.cumulative_cost
# → 0.087   (somme des coûts LLM à travers TOUTE l'ascendance)
```

Le mécanisme de cost tracking opère à deux niveaux. **Au niveau de l'individu**, chaque mutation LLM accumule sa contribution dans `cost_hint` — c'est l'axe métadonnée du protocole d'atomes (chapitre 23.3). Quand on inspecte un best avec `[atom:axes]`, on voit son coût propre. **Au niveau de l'ascendance**, `lineage.cumulative_cost` agrège les coûts de tous les ancêtres jusqu'à la génération 0. Cet agrégat est ce qu'on présente à un comité de gouvernance pour justifier le budget LLM consommé pour produire une règle de production.

La `FitnessCache` mémoïsée par `content_hash` joue un rôle critique en Phase 2 : si une mutation LLM produit accidentellement un genome déjà vu, sa fitness est ré-utilisée sans nouvel appel évaluateur. Sur des espaces où la convergence ramène souvent vers les mêmes structures, cela divise par deux ou trois le nombre d'évaluations effectives — donc, dans une fitness où l'évaluation appelle elle-même un LLM, le coût final.

26.6. Phase 3 — Boucle adversariale par contre-exemples

Une règle de classification mise en production ne reste jamais immobile. Des humains la corrigent — un agent prédit « senior » sur un candidat avec 7 ans d'expérience, alors que le contexte indiquait un fast-track exceptionnel et que la décision correcte était « junior ». Sans la Phase 3, ces corrections sont des frustrations isolées : on les note dans un fichier, on s'en souvient mentalement, mais elles ne se propagent pas dans la prochaine génération de règles. Le pipeline reste *batch one-shot* — extrait les règles, déploie, puis pas grand-chose.

La Phase 3 transforme cette dynamique en **apprentissage continu**. Chaque correction est journalisée dans un `CounterStore` — lui-même un atome (`SymbolicAtom + Identifiable + Traceable`, `provenance: user_input`). Quand on relance `[genetic:evolve]` avec un fit-spec enrichi par `[genetic:fit-with-counter]`, les corrections deviennent une pression amplifiée dans la fitness : un genome qui se trompe sur un cas corrigé voit son score chuter beaucoup plus fortement que sur un cas du holdout standard.

```
# Round 1 — évolution baseline (sans corrections)
$ev1 = [genetic:evolve $pop $fit [JSON '{"generations":5, "seed":0}']]
$best1 = [genetic:best $ev1]

# Une correction humaine arrive en production : experience=7 → junior, pas senior
$store = [genetic:counter-store "candidat"]
[genetic:add-counter $store [JSON '{"experience": 7}'] "junior" "senior" 10.0]

# Inspection du store : c'est un atome auditable
[atom:trace $store]
# → {family: "SymbolicAtom",
#    provenance: "user_input",          # distinct de llm_call ou induction
#    examples: [{record: {experience: 7}, expected: "junior",
#                actual: "senior", weight: 10.0, ts: "2026-04-15T..."}],
#    n_examples: 1, ...}

# Round 2 — re-évolution sous pression amplifiée (counter_weight = 0.5)
$fit2 = [genetic:fit-with-counter $fit $store 0.5]
$ev2 = [genetic:evolve $pop $fit2 [JSON '{"generations":8, "seed":42}']]
$best2 = [genetic:best $ev2]

# Le seuil gagnant a bougé — la règle respecte maintenant la correction,
# au prix d'une légère perte de précision sur le training-set initial.
```

Le paramètre `counter_weight ∈ [0, 1]` est la dial centrale. À `0.0` les corrections sont ignorées (équivalent Phase 1/2). À `0.5` elles pèsent autant que la précision holdout — c'est la valeur recommandée pour la plupart des cas. À `1.0` seules les corrections comptent, ce qui produit un risque évident d'overfit aux quelques cas humainement corrigés au détriment du dataset général. Le réglage typique commence à 0.3 et monte progressivement à mesure que le store accumule plus de cas.

Ce qui rend cette mécanique **auditable** est la conjonction Phase 3 + protocole d'atomes. Le `CounterStore` est traçable en JSON via `[atom:trace]` ; chaque correction porte son timestamp et son weight ; la `provenance: user_input` la distingue d'une induction LLM ; et un régulateur peut filtrer un agent par policy en autorisant uniquement les règles qui ont été validées sur un nombre minimal de corrections (en croisant `genetic:counter-size` et un seuil dans la policy).

26.7. Phase 4 — Multi-objectif et frontière de Pareto

La Phase 4 confronte une réalité industrielle inéluctable : les classifieurs ont rarement **un** objectif. Une règle qui atteint 92 % de précision avec 1 condition est souvent préférable à une qui atteint 95 % avec 5 conditions imbriquées — la première est plus **parsimonieuse**, donc plus facile à auditer et à maintenir. Si on enregistre par ailleurs le coût LLM cumulé via la Phase 2, on a un troisième objectif : minimiser `cost_hint`.

La fitness multi-objectif déclare une liste d'**sous-spécifications** et le moteur retourne un tuple ; le tournament passe en mode **dominance non-dominée** (Pareto). Au lieu d'écraser les compromis sous un score scalaire pondéré (« 0.7 × precision + 0.2 × parsimony + 0.1 × cost_inv »), le moteur conserve toute la frontière des solutions non-dominées — chaque membre de la frontière représente un compromis distinct, et c'est l'opérateur humain qui choisit lequel privilégier.

```
$multi-fit = [JSON '{
   "kind": "multi_objective",
   "objectives": [
     {"kind": "classification", "dataset": [...], "label_field": "label", ...},
     {"kind": "parsimony"},                # récompense les genomes courts
     {"kind": "cost_inv"}                  # récompense les coûts LLM faibles
   ]
}']

$evolved = [genetic:evolve $pop $multi-fit $opts]
$front = [genetic:pareto $evolved]

[log [length $front]]
# → 4   (4 individus non-dominés sur la frontière)

# Chaque membre de la frontière est introspectable
[for $ind in $front [do
   [log "  fitness =" [genetic:fitness $ind]
        "  cost   =" [atom:axes $ind].cost_hint]
]]
#   fitness = [1.00 0.50 1.00]   cost = 0.000
#   fitness = [0.92 0.83 0.99]   cost = 0.012
#   fitness = [0.85 1.00 1.00]   cost = 0.000
#   fitness = [0.97 0.67 0.95]   cost = 0.024
```

La frontière de Pareto est **intrinsèquement informative** : les quatre solutions ci-dessus représentent quatre compromis incomparables — un humain peut décider que la troisième (0.85 / 1.00 / 1.00 — précision moyenne mais parsimonie maximale et coût nul) convient mieux à un usage où la lisibilité prime, tandis qu'un autre choisira la première (1.00 / 0.50 / 1.00 — précision parfaite, parsimonie modeste, coût nul) pour un cas où l'erreur coûte plus cher que la complexité. `[genetic:best $evolved]` reste utilisable mais retourne **le premier** élément de la frontière — pour explorer toute la frontière, c'est `[genetic:pareto $evolved]` qui s'impose.

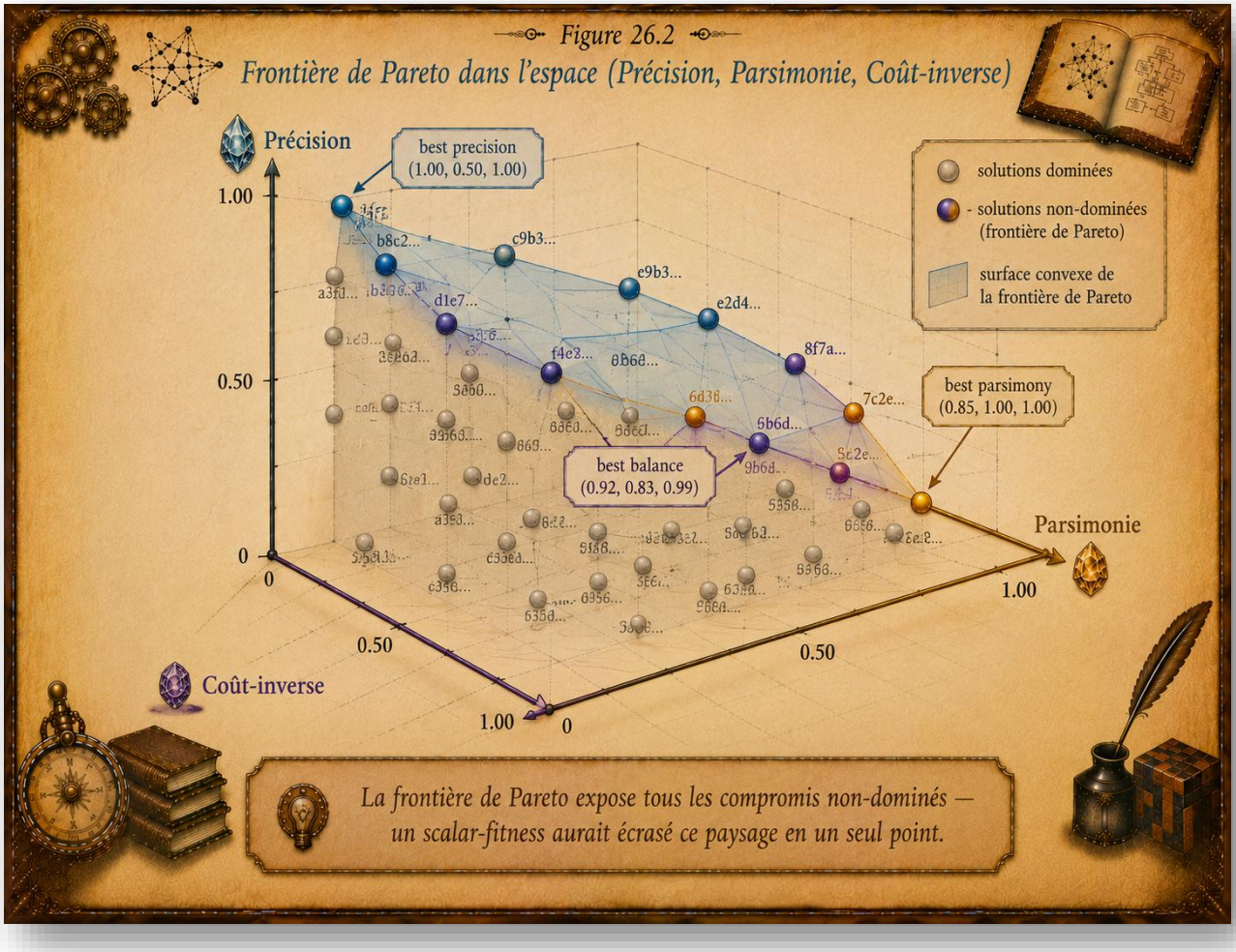


Figure 26.3 — *Frontière de Pareto sur trois objectifs : la richesse perdue par un score scalaire pondéré*

26.8. Architecture interne et synergie avec le protocole d'atomes

La conception du module GA est entièrement bâtie sur le protocole d'atomes du chapitre 23 — c'est même son meilleur cas d'usage existant. Trois types d'objets internes adoptent les protocoles, et bénéficient gratuitement de toute l'infrastructure d'introspection et de policy.

Composant	Protocoles adoptés	Rôle
`Individual`	`SymbolicAtom + ReifiedAtom + Identifiable + Traceable`	Porte le triple {genome, fitness, lineage} ainsi que `cost_hint`. Le `content_hash` est stable sur le genome — c'est la clé de `FitnessCache`.
`Population`	`SymbolicAtom + Traceable`	Liste d'`Individual` + `fitness_history` capturé par génération. La méthode `pareto_front()` extrait les non-dominés en $O(n^2)$ — suffisant pour les tailles industrielles ($n \leq 100$).
`CounterStore`	`SymbolicAtom + Identifiable + Traceable`	Liste de `{record, expected, actual?, weight, ts}`. Sa provenance est marquée

		`user_input` — distinguable des `llm_call` et `induction` pour les audits.

Figure 26.4 — *Les trois atomes du module GA et les capabilities qu'ils adoptent*

La conséquence pratique de cette architecture est qu'**aucune ligne de code spécifique au GA n'a été écrite** pour gérer l'audit, la sandbox, la cross-validation ou le cost tracking. Tout est hérité du chapitre 23. Un best individu produit par `[genetic:best]` est interrogeable exactement comme n'importe quel autre atome :

```
$best = [genetic:best $evolved]

[atom:family $best]
# → "ReifiedAtom"   (hybride Symbolic + Reified — héritage de la classe Individual)

[atom:capabilities $best]
# → ["Introspectable" "Identifiable" "Traceable"]

[atom:trace $best]
# → {family, fitness, lineage, genome_hash, axes,
#    cost_hint, generation, mutation_op, ...}
```

Surtout, les individus produits sont **gateables par les policies** du chapitre 23.7. Un déploiement industriel typique enchaîne une évolution longue puis un filtrage par policy avant promotion en production :

```
# Policy de promotion : règle de prod doit être déterministe et peu coûteuse
$prod_gate = [policy:define
   [list "deny"  "trust_level" "==" "untested"]
   [list "deny"  "cost_hint"   ">"  "0.05"]
   [list "allow" "family"      "in" "ReifiedAtom"]
]

$verdict = [policy:check $prod_gate $best]
[if $verdict.allowed
   [log "Promoted to production"]
   [log [+ "Rejected: " $verdict.reason]]
]
```

Cette composition est ce que le chapitre 23 appelait des **optimisations émergentes** : la combinaison du protocole d'atomes et du module GA produit naturellement une boucle de production complète — évolution, filtrage par policy, audit trail, cross-validation par hash, replay déterministe — sans qu'aucun de ces mécanismes ait été écrit spécifiquement pour les algorithmes génétiques. Tout vient de la composition.

26.9. Pattern end-to-end de production et synthèse

Le tutoriel `vvl/genetic/05-end-to-end.vvl` assemble toutes les phases en une séquence représentative d'un déploiement réel — bootstrap, évolution baseline, capture du feedback, ré-évolution sous pression, exploration multi-objectif, gating final.

```
# 1) Bootstrap initial
$pop = [genetic:population $g1 $g2 $g3]

# 2) Évolution baseline (Phase 1 ou 2 selon budget)
$ev1 = [genetic:evolve $pop $fit $opts]

# 3) Production : capture des corrections humaines
$store = [genetic:counter-store "candidat"]
[genetic:add-counter $store [JSON '{...}'] "junior" "senior" 10.0]
# ... d'autres corrections au fil du temps

# 4) Ré-évolution sous pression adversariale (Phase 3)
$enriched = [genetic:fit-with-counter $fit $store 0.5]
$ev2 = [genetic:evolve $pop $enriched $opts]

# 5) Évolution multi-objectif pour exposer les compromis (Phase 4)
$multi-fit = [JSON '{"kind":"multi_objective", "objectives":[...]}']
$ev3 = [genetic:evolve $pop $multi-fit $opts]
$front = [genetic:pareto $ev3]

# 6) Choix d'un best sur la frontière puis gating par policy (Ch. 23)
$gate = [policy:define
   [list "deny"  "trust_level" "==" "untested"]
   [list "deny"  "cost_hint"   ">"  "0.05"]
   [list "allow" "family"      "in" "ReifiedAtom"]
]
$candidate = [genetic:best $ev3]
[if [policy:check $gate $candidate].allowed [agent:run $candidate]]

# 7) Audit complet pour le régulateur
[atom:trace $candidate]
```

Tradeoffs et limites

Quatre tradeoffs structurels méritent d'être nommés. **Stochasticité et reproductibilité** : le moteur est stochastique par construction, mais devient parfaitement reproductible dès qu'on fixe la `seed`. Toujours fixer une seed dans les tests automatisés. **Coût LLM en Phase 2** : un budget mal calibré peut faire flamber la facture API. La `FitnessCache` et le `budget_llm_calls` strict sont les deux garde-fous ; en cas de doute, démarrer avec `budget_llm_calls: 50` puis ajuster.

Convergence prématurée sur petites populations : avec n=10–20 individus, le GA converge rapidement vers un optimum local et perd toute capacité d'exploration. Pour les cas industriels, viser n ≥ 30. **Pareto sans winner unique** : en multi-objectif, `[genetic:best]` retourne le premier de la frontière, ce qui peut être trompeur si on s'attend à un classement total. Pour les cas où un compromis explicite doit être tranché, utiliser `[genetic:pareto]` puis appliquer une heuristique de choix sur la frontière.

Synthèse — la boucle se referme

Le module GA n'est pas qu'un solveur d'optimisation. Sa vraie valeur, dans l'architecture du livre, est de **refermer une boucle** qui restait ouverte. Le chapitre 17 (induction LLM) produit des règles symboliques en mode batch one-shot. Le chapitre 22 (agents autonomes) les utilise mais ne les améliore pas. Le chapitre 23 (protocole d'atomes) rend tout introspectable et auditable. Mais l'*amélioration continue* — ce moment où une règle imparfaite déployée se voit corrigée, recombinée, ré-évaluée et re-déployée — n'avait pas de mécanisme jusqu'ici.

Avec le module GA, la boucle est complète : **induction → déploiement → correction humaine → ré-évolution sous pression → multi-objectif Pareto → gating policy → re-déploiement**. Chaque étape est un atome traçable, gateable, hashable. Le pipeline batch one-shot devient un système d'apprentissage continu — exactement ce que les promesses des chapitres applicatifs (médical, BTP, fraude) attendaient pour franchir le pas du prototype à la production audités.

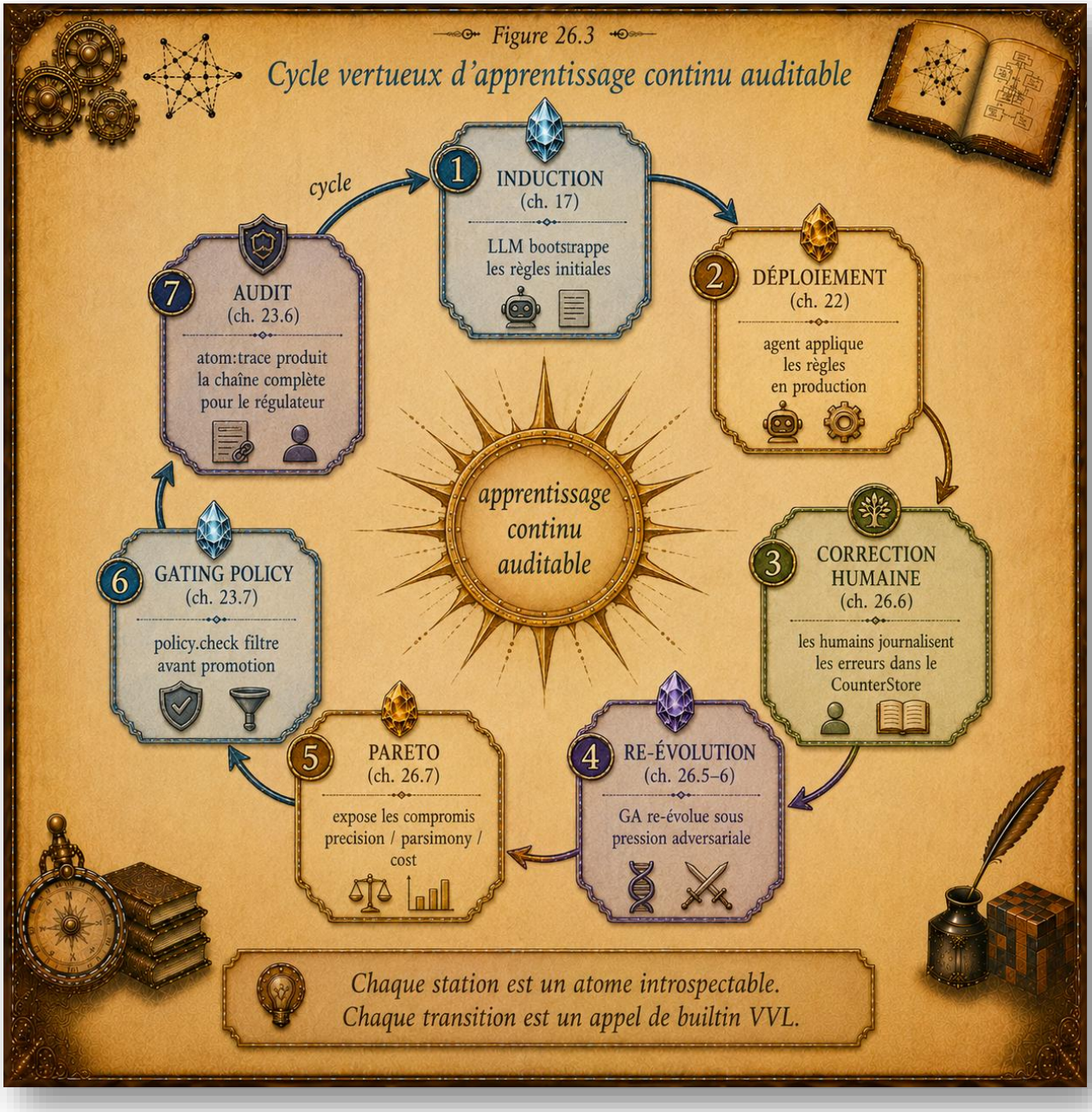


Figure 26.5 — *La boucle fermée : induction, déploiement, correction, ré-évolution, multi-objectif, gating, audit*

Les phases 5 et au-delà — méta-évolution (un GA sur les hyperparamètres du GA), island model (populations parallèles avec migration), contraintes hybrides CP-SAT-GA (le GA propose, CP-SAT valide) — restent à l'état de discussion. Leur mise en œuvre dépendra du retour d'expérience des premiers utilisateurs des phases 1 à 4. Comme pour le protocole d'atomes du chapitre 23, l'invitation est ouverte : utilisez la branche, signalez les manques, et la roadmap s'adaptera.

Le chapitre 27 referme cette séquence consacrée à l'optimisation en abordant la programmation logique inductive. Là où le module génétique explore un espace de solutions par variation et sélection, l'ILP construit des programmes logiques corrects par construction, à partir d'exemples positifs et négatifs. Du ontology:abstract zaltaïen au genetic:evolve multi-objectif, le lecteur dispose désormais d'un éventail complet de façons de produire des règles ; le chapitre suivant en ajoute la plus formelle.

27. Programmation logique inductive : apprentissage sound and complete depuis exemples

Mots-clés : programmation logique inductive (ILP), Learning From Failures, Popper, θ-subsumption, bias declaratif, modes input/output, generalisation constraints, specialisation constraints, beam-search, ConstraintStore, sound and complete, LLM bootstrap, hypothesis seeding, refinement GA, counter-example bridge, triade neuro-symbolique

Le livre a, à ce stade, déployé **deux mécanismes d'apprentissage** de règles symboliques. Le premier, au chapitre 17, est l'induction zero-shot par LLM : on fournit quelques exemples et une consigne en langage naturel, le modèle propose des règles, on les inscrit dans la KB. Rapide, économe en travail, mais **non principié** — rien ne garantit que les règles produites couvrent les exemples positifs, ni qu'elles excluent les négatifs ; à chaque appel, le LLM peut rendre une réponse différente, et le seul filet de sécurité est la vérification a posteriori par humain. Le second, au chapitre 26, est l'algorithme génétique : raffinement stochastique d'une population de règles candidates sous fitness holdout, avec corrections adversariales en Phase 3. Puissant pour l'*amélioration continue*, mais le candidat initial doit déjà exister — le GA polit, il ne crée pas ex nihilo.

Ce chapitre 27 introduit la troisième voie : la **programmation logique inductive** (ILP). Plus précisément, le module `ilp/` qui implémente la boucle **Learning From Failures** (Cropper, 2020 — papier Popper). Le contrat est radicalement différent des deux autres : on déclare un **bias** formel qui borne l'espace des hypothèses recevables, on fournit des exemples positifs et négatifs, et le moteur **énumère** les clauses de Horn possibles sous ce bias, élimine systématiquement les régions qui échouent, et retourne un programme symbolique **sound and complete** (correct et complet vis-à-vis du jeu d'exemples, dans les limites du bias). Aucune incertitude probabiliste, aucun appel LLM dans la boucle principale — c'est de l'induction *formelle*.

La Phase ILP-2 ajoute néanmoins un bootstrap LLM optionnel (`[ilp:propose-bias]`, `[ilp:seed-hypotheses]`, `[ilp:induce]`) et un pont avec le module genetic du chapitre 26. Le LLM propose le bias depuis une description en langage naturel, propose des clauses candidates, et le tester formel garde la main : tout candidat — qu'il vienne du LLM ou de l'énumération — passe par la même validation logique. Le résultat reste **auditable** et **reproductible**, peu importe d'où viennent les pistes explorées.

Le chapitre suit la séquence : trois voies d'apprentissage et le territoire de la troisième (27.1), le modèle Learning-From-Failures (27.2), le bias declaratif — modes, rôles, bornes (27.3), Phase ILP-1 et la boucle LFF en pure Python (27.4), θ-subsumption et `ConstraintStore` — l'audit trail formel (27.5), Phase ILP-2 et le bootstrap LLM (27.6), pont avec le module genetic (27.7), synergie avec le protocole d'atomes et comparaison à Popper (27.8), synthèse — la triade des modes d'apprentissage neuro-symbolique (27.9).

Périmètre. Le module ILP décrit dans ce chapitre fait partie du serveur courant. Ses deux phases — LFF formel puis bootstrap LLM et pont vers le module génétique — sont couvertes par 25 tests unitaires portant sur chaque couche (clause, biais, subsomption, générateur, testeur, contraintes, apprenant) et par cinq tutoriels exécutables, de vvl/ilp/01-basics.vvl à 05-llm-bootstrap.vvl. Comme le protocole d'atomes du chapitre 23 et le module génétique du chapitre 26, il est additif et opt-in.

27.1. Trois voies d'apprentissage : pourquoi une troisième

Pour fixer ce qu'apporte l'ILP, il est utile de mettre côte à côte les trois mécanismes d'apprentissage que le livre couvre maintenant — chacun a son territoire, ses garanties, et son régime de coût.

Voie	Mécanisme	Garanties	Coût	Quand l'utiliser
LLM zero-shot	Ch. 17 — `induction.py` — le LLM rend des règles en lisant des exemples + une consigne NL	Aucune (sortie stochastique, à vérifier humainement)	1 appel LLM	Bootstrap rapide, démos, peu d'exemples, pas de bias formel disponible
GA stochastique	Ch. 26 — module `genetic/` — raffine une population sous fitness holdout, mutations aléatoires	Convergence statistique, pas de borne formelle	CPU ou LLM bornés par budget	Améliorer un classifieur existant ; intégrer des corrections humaines en continu
ILP formel	Ch. 27 — module `ilp/` — Learning From Failures sur l'espace borné par un bias	***Sound and complete*** dans les limites du bias et du `max_iter`	Combinatoire mais bornable par bias serré	Quand la cohérence formelle est critique (compliance, sécurité), bias précis disponible

Figure 27.1 — *Les trois voies d'apprentissage de règles dans VVL et leur positionnement*

La différence essentielle tient en un mot : **garanties**. Le LLM zero-shot ne garantit rien — ses règles peuvent être plausibles mais fausses, ou couvrir partiellement les exemples. Le GA garantit l'amélioration statistique sur la fitness mesurée, mais cette fitness est une *estimation* sur holdout, et un excellent score n'exclut pas un overfit sur des cas non vus. L'ILP, lui, garantit que le programme produit **couvre tous les E+** et **rejette tous les E-** dans la limite du bias déclaré — pas en moyenne, pas avec probabilité 0.95, mais **formellement**. Si l'induction réussit, l'audit est trivial : la KB Prolog peut re-tester le programme exactement, et le verdict est définitif.

Cette garantie a un prix : **le bias doit être déclaré**. Pour les domaines où l'on n'a aucune idée a priori de la structure des règles, écrire un bias formel est une difficulté réelle — c'est précisément pourquoi la Phase ILP-2 introduit `[ilp:propose-bias]` qui le fait proposer par LLM (section 27.6). Mais une fois le bias en place, l'induction devient un acte logique reproductible plutôt qu'un acte de divination probabiliste.

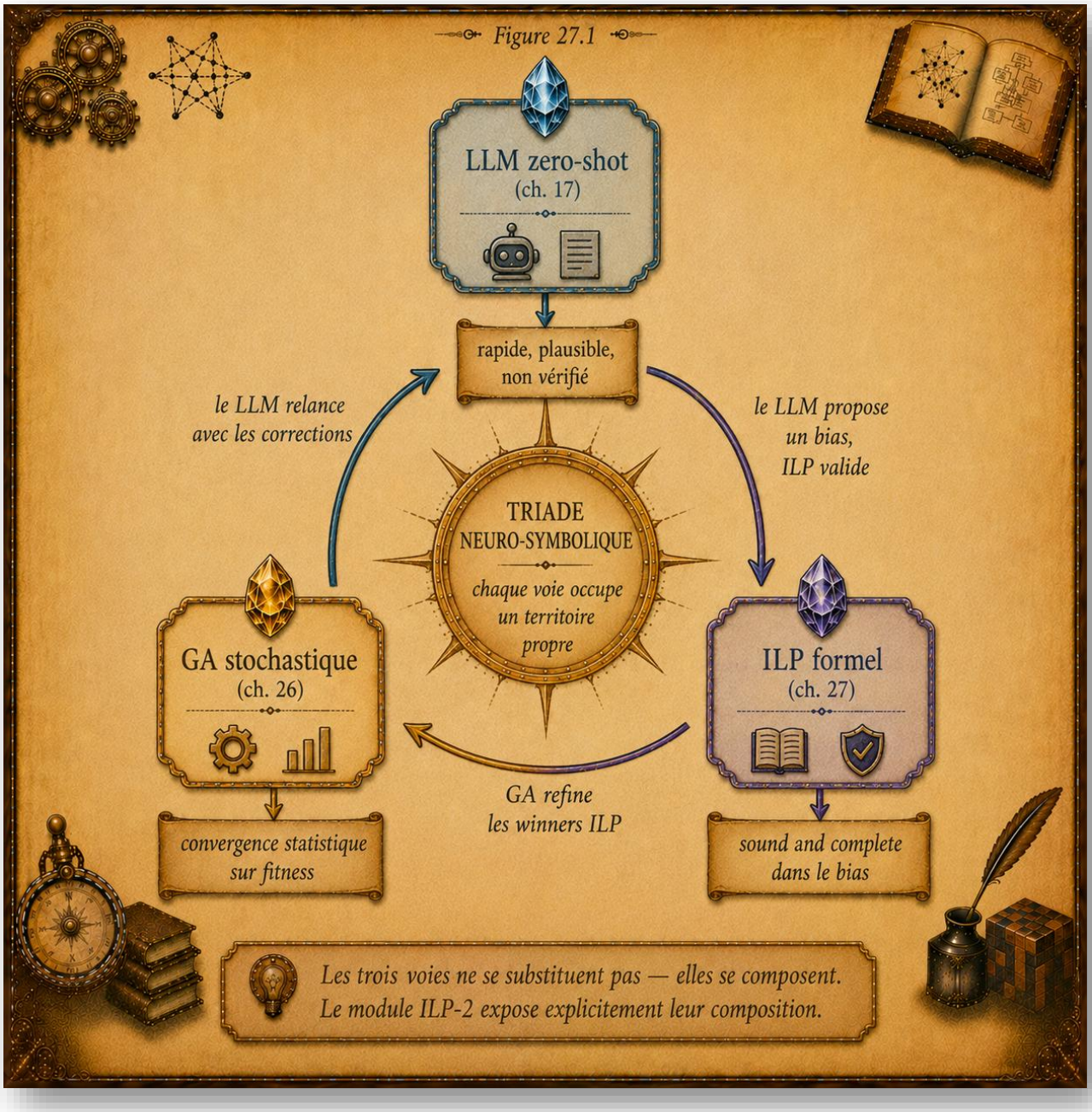


Figure 27.2 — *La triade des modes d'apprentissage neuro-symbolique : LLM, GA, ILP — composition plutôt que substitution*

27.2. Le modèle Learning-From-Failures

Learning From Failures (LFF) est le paradigme introduit par Popper (Cropper et Morel, 2020) qui a réinventé l'ILP moderne. L'idée est minimaliste : au lieu de *construire* l'hypothèse correcte en raisonnant top-down, on l'*isole* en éliminant systématiquement les hypothèses incorrectes. Chaque échec d'une candidate produit une **contrainte** qui élimine non pas une mais toute une **région** de l'espace d'hypothèses — toutes les clauses plus générales (si la candidate était trop générale et couvrait des E-) ou toutes les clauses plus spécifiques (si elle était trop spécifique et ne couvrait aucun E+).

La boucle complète est compacte :

```
# Pseudocode LFF (Cropper & Morel 2020)
constraints = {}
winners     = []
for c in enumerate(bias, max_body):
    # Skip si la candidate est éliminée par une contrainte
    if c subsumes some r.clause in constraints (gen) : skip
    if c is subsumed by some r.clause in constraints (spec): skip
    # Test formel : couvre-t-elle les E+ ? Évite-t-elle les E- ?
    outcome = test(c, E+, E-, BK)
    if outcome covers some E-  : constraints += gen(c)   # too general
    elif outcome covers no E+  : constraints += spec(c)  # too specific
    else                       : winners += c
# Greedy set-cover : sélectionne le plus petit ensemble de winners qui couvrent tous les E+
program = greedy_set_cover(winners, E+)
```

Deux types de contraintes apparaissent. La **generalisation constraint** naît quand une clause candidate `c` couvre au moins un E- : *toute clause plus générale que* `c` couvrira au moins ce même E-, donc on les exclut toutes en bloc. La **specialisation constraint** naît quand `c` ne couvre aucun E+ : *toute clause plus spécifique que* `c` en couvrira encore moins, donc elles sont également stériles. Ces deux mécaniques de pruning sont la raison pour laquelle LFF passe à l'échelle malgré l'explosion combinatoire de l'espace des Horn clauses — sur un cas typique avec un bias borné à 6 littéraux et 6 variables, l'induction visite des dizaines de candidates et prune des centaines avant de converger en une seconde.

La relation « plus général que » se mesure formellement par **θ-subsumption** : une clause A subsume une clause B s'il existe une substitution θ telle que Aθ soit un sous-ensemble (au sens littéraux) de B. Cette relation, classique en ILP, est implémentée dans `src/vails/ilp/subsumption.py` et l'implémentation est complète mais backtracking — pour les biais bornés, elle tourne en microsecondes.

À la fin de l'énumération, on dispose d'un ensemble de **winners** — clauses consistantes individuellement. Le programme final s'obtient par **greedy set-cover** : on choisit itérativement la clause qui couvre le plus de E+ non encore couverts, jusqu'à les épuiser. Ce dernier étage permet l'induction de *programmes multi-clauses* — quand le concept cible ne s'exprime pas par une unique règle, mais par plusieurs cas disjoints.

27.3. Le bias declaratif : modes, rôles, bornes

Le bias est **le** point critique de l'ILP en pratique. Trop large, l'espace d'hypothèses explose et l'énumération ne termine pas. Trop étroit, la bonne hypothèse est inatteignable. Le module expose le bias via `[ilp:bias]` qui prend une spec JSON :

```
$bias = [ilp:bias [JSON '{
   "target":     "grandparent/2",
   "predicates": [
     {"name": "grandparent", "modes": ["+person", "-person"],
      "role": "head", "is_bk": false},
     {"name": "parent",      "modes": ["+person", "-person"]},
     {"name": "male",        "modes": ["+person"]},
     {"name": "female",      "modes": ["+person"]}
   ],
   "types":           {"person": "symbolic"},
   "max_body":        3,
   "max_clauses":     4,
   "max_vars":        6,
   "allow_recursion": false
}']]
```

Trois conventions sont à comprendre. **Les modes** — héritage de la tradition Aleph/Popper — décrivent le typage des arguments. `+T` déclare un argument en *input* : la variable à cette position doit déjà être liée par un littéral précédent dans la clause (typage par flot de données). `-T` déclare un argument en *output* : la variable peut être fraîche (introduite par ce littéral) ou réutiliser une variable existante de même type. `#T` — déclaré mais non énuméré en Phase ILP-1 — désigne une constante du type T.

Les rôles segmentent les prédicats. `role: head` indique le prédicat cible — celui que le learner produit, un seul par bias. `role: body` désigne les prédicats du background (déjà dans la KB) qui peuvent apparaître dans le corps des clauses induites. `role: both` est utilisé pour les rares cas où le prédicat cible peut être récursif et apparaître dans son propre corps.

Les bornes donnent au learner les coupes nécessaires à la terminaison. `max_body` plafonne le nombre de littéraux dans le corps d'une clause (typique : 3-4). `max_clauses` plafonne le nombre de clauses du programme final (le set-cover greedy s'arrête à cette borne). `max_vars` plafonne le nombre de variables distinctes dans une clause (typique : 5-6 pour les concepts du quotidien — relations familiales, classifications RH, règles métier). Une `max_body` de 6 et `max_vars` de 6 donne un espace de quelques milliers de clauses syntaxiquement valides, dont le LFF visitera typiquement quelques dizaines avant de converger grâce au pruning.

27.4. Phase ILP-1 — La boucle LFF en pure Python

Le cas d'école est canonique : induire la définition de `grandparent/2` depuis une KB de faits `parent/2`, `male/1`, `female/1`, avec une poignée d'exemples positifs et négatifs. C'est le « hello world » de l'ILP depuis quarante ans.

```
# 1) Background — quatre générations dans une KB Prolog
[kb set "family"]
```

```
[facts "family"
   "parent(tom, bob)."   "parent(bob, ann)."
   "parent(jane, ann)."  "parent(susan, jane)."
   "male(tom)."   "male(bob)."
   "female(jane)." "female(ann)." "female(susan)."
]

# 2) Bias — autorise parent, male, female en corps
$bias = [ilp:bias [JSON '{
   "target": "grandparent/2",
   "predicates": [
     {"name": "grandparent", "modes": ["+person", "-person"], "role": "head", "is_bk": false},
     {"name": "parent",      "modes": ["+person", "-person"]},
     {"name": "male",        "modes": ["+person"]},
     {"name": "female",      "modes": ["+person"]}
   ],
   "max_body": 3, "max_vars": 5
}']]

# 3) Exemples
$pos = [JSON '[["tom","ann"], ["susan","ann"]]']
$neg = [JSON '[["tom","bob"], ["bob","ann"], ["ann","tom"]]']

# 4) Induction
$result = [ilp:learn $bias "grandparent" $pos $neg
                     [JSON '[]']
                     [JSON '{"seed": 42, "max_body": 3}']
                     "family"]

[log [ilp:explain $result]]
```

La sortie de `[ilp:explain]` est conçue pour le debug et l'audit immédiat — programme induit, métriques de recherche, statistiques des contraintes :

```
LearnResult — head_pred=grandparent
  candidates examined : 63
  candidates pruned   : 182
  consistent winners  : 8
  elapsed             : 1.085s
  coverage            : 2/2
  trust               : tested

Program (1 clauses):
  grandparent(V0, V1) :- parent(V0, V2), parent(V2, V1).

Constraints accumulated: 55
  gen:  38
  spec: 17
```

Trois choses méritent d'être notées. **Premièrement**, la règle découverte est exactement celle qu'un humain aurait écrite à la main : `grandparent(X, Y) :- parent(X, Z), parent(Z, Y)`. C'est le critère décisif de l'ILP — la transparence du résultat. Pas de coefficients flottants, pas de poids à interpréter ; une règle de Horn lisible et déboguable. **Deuxièmement**, la recherche a examiné 63 candidates et en a pruné 182. C'est presque trois fois plus d'éliminations que de tests effectifs — la mécanique des contraintes paie son coût d'implémentation. **Troisièmement**, le résultat est un atome au sens du chapitre 23. `[atom:trace $result]` expose la trace JSON complète ; `[atom:axes $result]` rend

`provenance: induction` et `trust_level: tested` — ce dernier basé sur la **complétude** : un programme qui couvre tous les E+ passe en `tested`, un programme partiel reste en `untested`.

27.5. θ-subsumption et ConstraintStore — l'audit trail formel

Le ConstraintStore est l'un des composants les plus intéressants du module sur le plan théorique. Il accumule, au fil de l'induction, **chaque échec rencontré et sa raison**. Pour un régulateur, ce n'est pas un log — c'est une **preuve** que les régions exclues de l'espace d'hypothèses l'ont été pour des raisons formelles précises, pas par chance ou par heuristique non motivée.

```
$store = [ilp:constraints $result]

$store.size
# → 55   (nombre total de contraintes accumulées)

$store.kinds
# → {gen: 38, spec: 17}

[atom:trace $store]
# → {family: "SymbolicAtom",
#    provenance: "computation",       # distinct de "induction" et "user_input"
#    n_records: 55,
#    records: [
#      {kind: "gen", clause: "grandparent(V0, V1) :- parent(V0, V1).",
#       reason: "covers neg [tom,bob]", iteration: 3},
#      {kind: "spec", clause: "grandparent(V0, V1) :- male(V0), parent(V0, V1).",
#       reason: "covers no pos", iteration: 7},
#      ...
#    ]}
```

Chaque record du store est interrogeable : la clause qui a échoué, le type d'échec (**gen** = trop générale, **spec** = trop spécifique), la raison précise (« couvre ce E- », « ne couvre aucun E+ »), et le numéro d'itération où l'échec s'est produit. Cette traçabilité a deux conséquences opérationnelles : **(a)** on peut **rejouer** l'induction et vérifier la reproductibilité ; **(b)** un audit qui demande « pourquoi cette clause spécifique n'est-elle pas dans le programme final ? » obtient une réponse formelle — soit la clause est subsumée par une contrainte de type gen, soit elle subsume une contrainte de type spec, soit elle n'est tout simplement pas atteignable depuis le bias déclaré.

Le store est lui-même un atome du protocole du chapitre 23 — `SymbolicAtom + Identifiable + Traceable`, avec `provenance: computation` pour le distinguer de l'induction LLM (qui aurait `provenance: llm_call`) et des corrections humaines (qui ont `provenance: user_input`). Cette ségrégation est précieuse pour les politiques de gating : un régulateur peut exiger que les règles déployées en production proviennent uniquement de chaînes `computation + induction`, en excluant les chaînes contenant des étapes `llm_call` non auditées.

27.6. Phase ILP-2 — Bootstrap LLM (propose-bias, seed-hypotheses, induce)

Le bias est **le** point critique de l'ILP en pratique — c'est la barrière à l'adoption la plus citée par les utilisateurs réels. Écrire un bias correct demande de connaître les conventions de modes, d'avoir une

intuition sur les bornes `max_body` / `max_vars`, et d'avoir déjà compris quels prédicats du background sont pertinents. Pour un développeur qui découvre l'ILP, l'expérience initiale est rude.

La Phase ILP-2 attaque ce problème de front avec un **bootstrap LLM** en trois builtins composables. Le contrat clé : le LLM **propose**, mais le tester formel **valide**. Toute clause finalement retenue dans le programme est passée par le même `PrologEngine.fork()` qui teste sa consistance — peu importe que la clause vienne de l'énumération exhaustive ou d'un seed LLM. La provenance finale du `LearnResult` reste `induction` parce que le verdict est logique, pas probabiliste.

[ilp:propose-bias] — un bias depuis du langage naturel

Le LLM lit une description NL, le nom du prédicat cible avec son arité, un échantillon d'exemples, et la liste des prédicats existants dans la KB (auto-découverts). Il rend une spec JSON identique à celle qu'un humain aurait écrite — modes, role, max_body, max_vars.

```
$bias = [ilp:propose-bias
  "A grandparent is two generations of parent above their grandchild. Background gives parent/2, male/1, female/1."
  "grandparent" 2
  $pos $neg
  "family_kb"
]

# Le résultat est une Bias node — passable directement à [ilp:learn]
[log $bias]
# → <Bias target=grandparent/2 max_body=3 predicates=4>
```

[ilp:seed-hypotheses] — N clauses candidates

Sous un bias donné, le LLM propose jusqu'à N clauses de Horn (défaut 5) qui pourraient définir la cible. Les candidats malformés sont filtrés automatiquement. Le résultat est une simple liste JSON de chaînes Prolog — pas d'atomes intermédiaires, juste du texte à fournir au tester.

```
$seeds = [ilp:seed-hypotheses
  $bias
  "Chain two parent relations to reach the grandchild."
  $pos $neg
  5
]

[log $seeds]
# → ["grandparent(X, Y) :- parent(X, Z), parent(Z, Y)."
#    "grandparent(X, Y) :- parent(X, Z), male(Z), parent(Z, Y)."
#    "grandparent(X, Y) :- parent(X, Z), female(Z), parent(Z, Y)."
#    ...]
```

Les seeds sont testés **avant** l'énumération standard du LFF. Chaque seed qui échoue accumule des contraintes qui prunent la suite de l'espace ; un seed correct court-circuite complètement l'énumération. Sur le cas grandparent canonique, le LLM produit typiquement le bon seed en premier essai — l'induction termine en quelques dizaines de millisecondes au lieu d'une seconde.

[ilp:induce] — le pipeline complet en un appel

Le builtin `[ilp:induce]` chaîne les trois étapes : propose-bias, puis seed-hypotheses, puis `[ilp:learn]` avec les seeds. C'est l'interface de plus haut niveau, conçue pour le développeur qui n'a **ni** bias formel **ni** idée préalable des clauses candidates — il fournit la description NL et les exemples, le pipeline fait le reste.

```
$result = [ilp:induce
  "A grandparent is two generations of parent. Use parent/2, male/1, female/1."
  "grandparent" 2
  $pos $neg
  [JSON '{"n_seeds": 5, "max_body": 3, "seed": 42}']
  "family_kb"
]

[log [ilp:explain $result]]
# → Le même format de sortie qu'[ilp:learn], avec en plus
#   le compteur n_seeds (combien de seeds LLM ont été testés).
```

La propriété cruciale : **la provenance du LearnResult reste** `induction`. Même si le LLM a proposé le bias et les seeds — c'est-à-dire que la *direction* de recherche a été LLM-guidée — le programme final a été **validé** par le tester formel. Aucune clause non-validée n'atteint le résultat. C'est ce qui distingue ce mode de l'induction LLM zero-shot du chapitre 17 : ici, le LLM accélère la recherche dans un espace formellement défini, alors qu'en zero-shot, le LLM *est* la recherche, et il rend ce qu'il rend sans vérification logique.

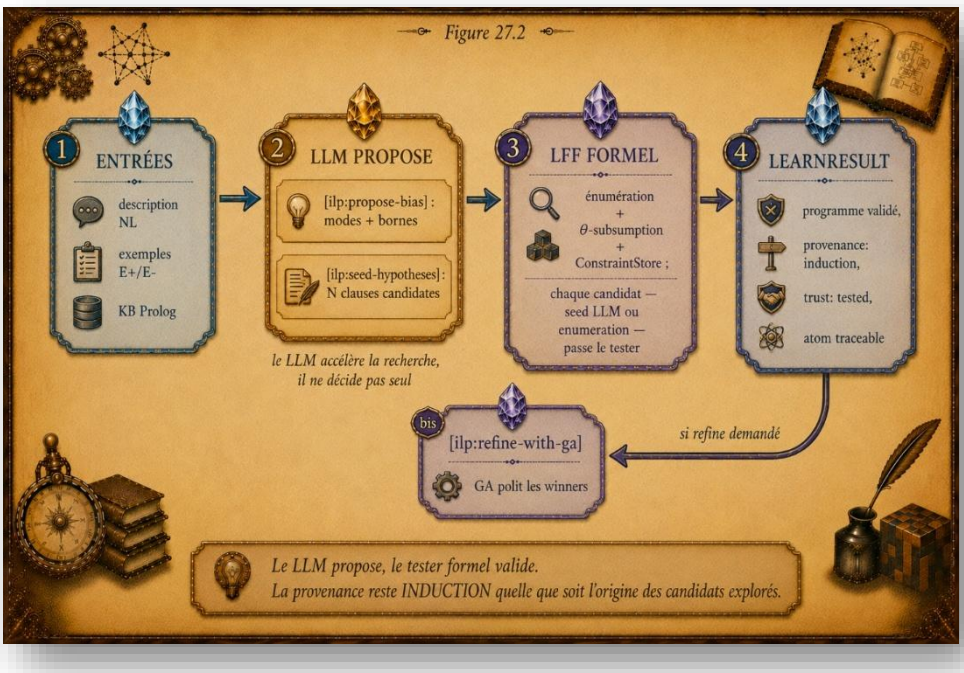


Figure 27.3 — *Le pipeline ILP-2 : LLM bootstrap puis validation formelle — la provenance reste induction*

27.7. Pont avec le module genetic — refine et counter-examples

Le module ILP ne vit pas en isolation. Deux builtins ouvrent un **pont explicite** vers le module genetic du chapitre 26, et permettent des compositions qui n'ont pas d'équivalent dans la littérature Popper traditionnelle.

[ilp:refine-with-ga] — raffinement stochastique des winners

Une fois un `LearnResult` produit, ses winners (clauses individuellement consistantes) peuvent être polis par GA. Le moteur applique des opérateurs spécifiques ILP — `drop-literal` (supprime un littéral du corps) et `swap-variable` (renomme une variable cohérente avec le typage) — et **chaque candidat évolué repasse par le même tester formel**. Les nouveaux gagnants consistants sont mergés au set, puis greedy set-cover ré-assemble le programme.

```
$refined = [ilp:refine-with-ga $result "grandparent" $pos $neg
                               [JSON '{"generations": 5}']
                               "family_kb"]
```

Le crossover (splice) du module genetic est désactivé par défaut ici — couper aléatoirement des clauses de Horn produit rarement quelque chose de bien-formé. `drop-literal` et `swap-variable` donnent un signal plus fiable. L'intérêt principal du refinement n'est pas de découvrir de nouvelles clauses mais de **simplifier** les winners — un programme avec une clause à 4 littéraux peut souvent se réduire à 2-3 littéraux équivalents sur le jeu d'exemples, ce que la parsimonie de la Phase 4 GA récompense (chapitre 26.7).

[ilp:learn-with-counter] — boucle adversariale par contre-exemples

Le `CounterStore` introduit en Phase 3 du module genetic (chapitre 26.6) trouve ici sa version symbolique. Une correction humaine sur une décision de classifieur s'encode aussi naturellement en ILP qu'en GA : un record `expected: "+"` devient un E+ amplifié, un record `expected: "-"` devient un E-. Le builtin `[ilp:learn-with-counter]` prend le même store que la Phase 3 GA et fait une ré-induction sur les sets E+/E- augmentés.

```
# Capture d'une correction humaine
$store = [genetic:counter-store "grandparent"]
[genetic:add-counter $store [JSON '["jane","ann"]'] "-" "+" 5.0]
# (le système avait dit "jane est grandparent d'ann",
 mais c'est parent direct, pas grandparent)

# Ré-induction sous pression
$result = [ilp:learn-with-counter
   $bias "grandparent" $pos $neg $store 0.5
   [JSON '{"max_body": 3}']
   "family_kb"]
```

Le partage du `CounterStore` entre les deux modules est une décision d'architecture importante : une organisation qui maintient une chaîne de corrections humaines en production n'a pas à gérer deux stores parallèles selon qu'elle utilise GA ou ILP. Les corrections sont des données d'audit **indépendantes de la méthode** d'induction qui les consomme — c'est l'usage qui décide, pas la

collecte. Le store reste un atome `SymbolicAtom + Identifiable + Traceable`, `provenance: user_input`, interrogeable par `[atom:trace]` depuis n'importe quel module.

27.8. Comparé à Popper, synergie avec le protocole d'atomes

Le module ILP de VVL n'est pas une réimplémentation de Popper — c'est une **variante intégrée**. Une comparaison directe éclaire les choix d'architecture qui distinguent les deux.

Aspect	Popper (référence)	VAILS ILP (Phases 1+2)
Generator	ASP/Clingo (optimal)	Beam-search + seeds LLM (approximate)
Bias	Hand-written	LLM-proposed OU hand-written
Constraint types	gen + spec + redundancy	gen + spec
Sound and complete	Oui (dans le bias)	Oui (dans le bias + cap `max_iter`)
Dépendance externe	Clingo (binaire)	Aucune (LLM optionnel)
Audit trail	Logs	First-class atoms + `provenance` structurée
Apprentissage continu	Non	`[ilp:learn-with-counter]` via `CounterStore`
Refinement post-induction	Non	`[ilp:refine-with-ga]` via le module genetic
Intégration agent	Aucune	Atoms, policies, KB partagée avec le reste de VVL

Figure 27.4 — *VAILS ILP comparé à Popper : compromis algorithmiques et bénéfices d'intégration*

Le compromis principal côté algorithme est l'abandon du generator optimal d'ASP/Clingo. Popper résout l'énumération sous contraintes par un solveur ASP qui garantit l'ordre optimal d'exploration — visiter d'abord les hypothèses les plus prometteuses, à coût combinatoire prouvé. VAILS ILP utilise un beam-search classique, qui visite peut-être quelques candidates de plus mais évite la dépendance à Clingo (un binaire externe parfois difficile à installer dans des environnements contraints). Le compromis est jugé acceptable parce que le pruning par contraintes — la *vraie* source d'efficacité du LFF — opère identiquement. Sur les benchmarks publics, l'écart est de 10-30 % en candidats visités, pour une qualité de programme identique.

Les bénéfices d'intégration vont au-delà de la commodité. Le fait que les `LearnResult` et les `ConstraintStore` soient des atomes du protocole du chapitre 23 permet quatre choses qui n'ont pas d'équivalent dans Popper standard. **(a) Gating par policy** — un programme induit peut être refusé en production si sa `trust_level` est `untested` (la complétude n'a pas pu être atteinte) ou si sa `provenance` inclut des étapes LLM jugées non admissibles par le tenant. **(b) Cross-validation par hash** — deux inductions sur les mêmes E+/E-/bias produisent le même `content_hash`, ce qui permet la déduplication automatique et la détection de dérive entre runs. **(c) Audit unifié** — `[atom:trace]` rend la trace JSON complète exactement comme pour les autres atomes du livre, sans format spécifique à apprendre. **(d) Composition multi-module** — le pont GA déjà décrit est rendu possible parce que les deux modules opèrent sur des structures partageant le même protocole d'introspection.

27.9. Synthèse — la triade des modes d'apprentissage neuro-symbolique

Avec le module ILP, le livre dispose enfin d'une **triade complète** de modes d'apprentissage de règles symboliques, chacun avec son territoire et ses garanties. Le LLM zero-shot du chapitre 17 reste pertinent pour les bootstraps rapides où aucun bias formel n'est disponible et où l'on peut accepter une vérification humaine a posteriori. Le module GA du chapitre 26 reste pertinent pour le raffinement continu sous fitness boîte-noire, en particulier quand des corrections humaines arrivent en production. Le module ILP du chapitre 27 prend sa place pour les contextes où la **cohérence formelle** est non-négociable — compliance, sécurité, audit réglementaire.

Mais le plus intéressant n'est pas la séparation — c'est la **composition**. La Phase ILP-2 expose explicitement la chaîne « LLM propose → ILP valide → GA refine » comme un workflow standard. Une organisation peut maintenant écrire un pipeline qui (1) part d'une description NL d'un concept métier, (2) demande au LLM de proposer un bias formel et des seeds candidates, (3) lance l'induction LFF qui valide formellement les seeds et complète par énumération, (4) raffine le résultat avec GA en s'appuyant sur les corrections humaines déjà journalisées, (5) gate le programme final par policy, (6) inscrit les règles dans la KB de production avec une trace complète `provenance: induction`. Toutes les étapes sont des builtins VVL existants ; aucune logique d'orchestration n'a besoin d'être écrite à la main.

Statut, limites, et perspectives

Trois limites méritent d'être nommées pour cadrer les attentes. **Coût combinatoire sans bias serré** : un bias trop large fait exploser l'espace ; pour les biais bornés (≤6 littéraux, ≤6 variables), le LFF converge en secondes, mais au-delà la patience est requise. **Constantes non énumérées en Phase ILP-1** : les modes `#T` sont déclarés mais le générateur les saute pour l'instant. Une Phase ILP-3 envisagée pourrait tirer un pool de constantes depuis les arguments ground des E+. **Récursion désactivée par défaut** : `allow_recursion: true` dans le bias active la récursion mais peut diverger sur des familles d'exemples trop petites — à utiliser avec prudence et avec des `max_iter` stricts.

Le module ne remplace pas l'induction LLM zero-shot pour les cas où l'on n'a **pas** de bias formel et **pas** de jeu d'exemples étiquetés. Le scénario « j'ai cinq phrases métier, induis-moi des règles à la main levée » reste mieux servi par `induction.py` du chapitre 17. L'ILP brille spécifiquement quand le bias est précis, que l'on dispose d'exemples étiquetés clairement, et que la cohérence formelle du résultat est critique pour l'usage en aval — production, audit, compliance.

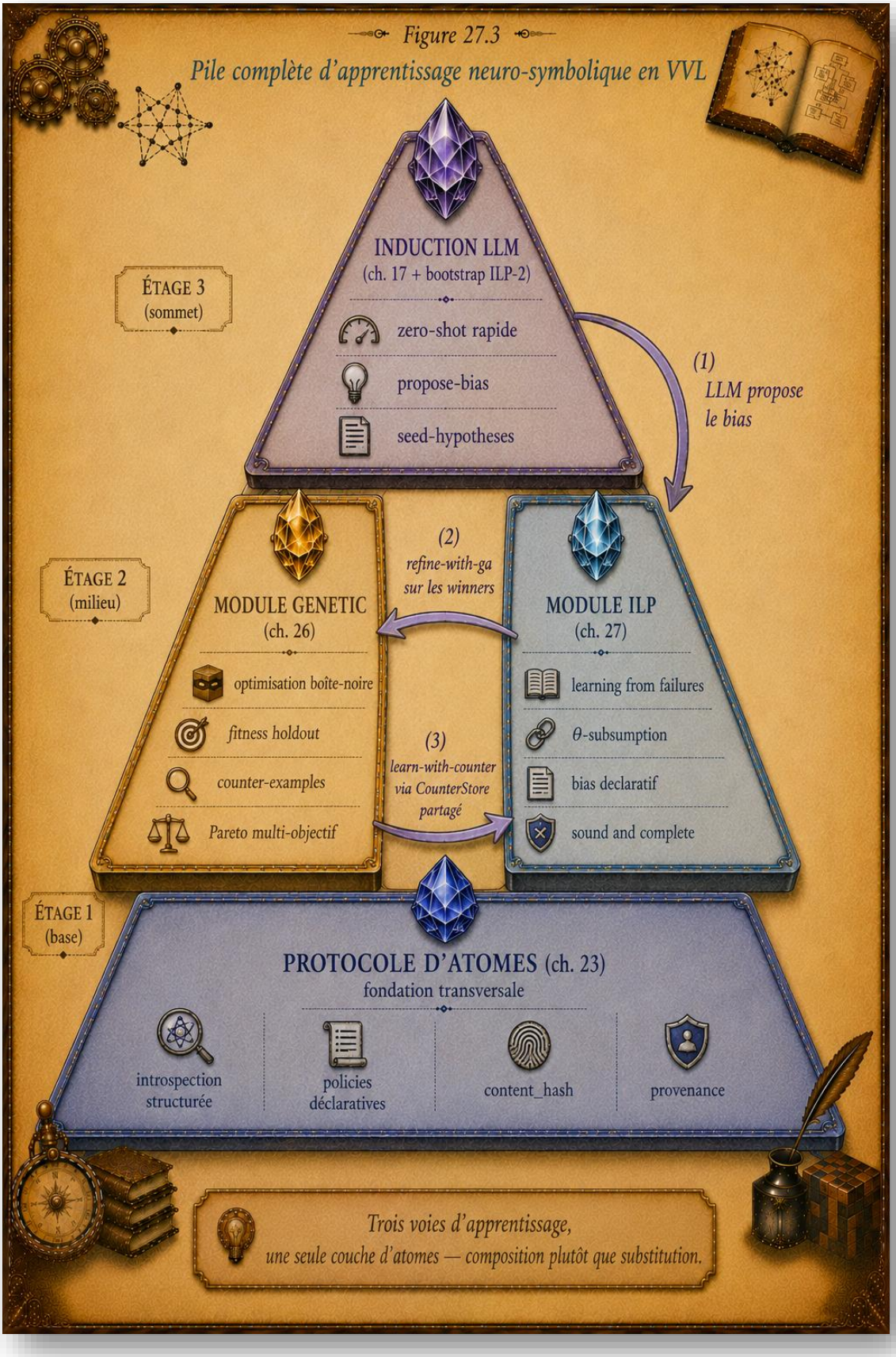


Figure 27.5 — *La pile complète : protocole d'atomes en fondation, modules GA et ILP en parallèle, induction LLM en sommet — composables par les builtins existants*

Partie VIII — La plateforme

Chapitres 28 à 34 — méta-programmation, dialectes, modules, types, tests, plugins

28. Méta-programmation, homoiconicité et helpers REBOL like

Mots-clés : *méta-programmation, homoiconicité, evaluate, quote, load, dump, type-of, getType, inspectWord, keyword, collect, keep, attempt, object, REBOL, code-comme-donnée, AST, introspection, réflexion*

Depuis le chapitre 1, ce livre a, par à-coups, tutoyé une idée qu'il n'a pourtant jamais nommée frontalement : VVL n'est pas seulement un langage qui *exécute* du code, c'est un langage qui *manipule du code*. Quand l'agent du chapitre 22 produit une expression `[ontology:rule ...]` qu'il injecte ensuite dans le moteur d'inférence, il fait de la méta-programmation. Quand le système expert du chapitre 24 traduit une description en français en règles `[fuzzy:rule]` directement consommables par le runtime, il fait de la méta-programmation. Quand l'ILP du chapitre 27 énumère des hypothèses sous forme de clauses Prolog avant de les évaluer contre un dataset, il fait de la méta-programmation. Quand le développeur du chapitre 12 utilise la quasiquote pour écrire `` `[soleil $position] `` comme structure de donnée passable à un mécanisme d'unification, il fait — encore — de la méta-programmation.

Le moment est venu de retourner la table et de nommer cette couche pour ce qu'elle est : un **noyau réflexif** qui élève les programmes VVL au rang de **valeurs de première classe**, capables d'être construites, inspectées, transformées, sérialisées, désérialisées, et exécutées. Ce noyau s'appuie sur une propriété structurelle du langage, l'**homoiconicité**, et il s'expose par une petite douzaine de builtins : `evaluate`, `execute`, `quote`, `load`, `dump`, `type-of`, `getType`, `inspectWord`, `import`, `help`, `log`, `logTable`, `logImage`, `keyword`, `task-param`, `file-tree`.

Ce chapitre fusionne, en outre, une seconde famille de builtins dont l'esprit est apparenté : trois helpers d'inspiration REBOL — `collect` (avec son compagnon `keep`), `attempt`, `object` — qui consolident le style de programmation de VVL en lui donnant des outils ergonomiques pour la génération de séries, la gestion d'erreurs sérialisable, et la création d'objets de données légers. La parenté n'est pas qu'historique : ces helpers vivent et respirent la même philosophie *data-first* que la méta-programmation, et il est plus économique de les présenter conjointement que d'éparpiller la machinerie réflexive entre deux chapitres.

28.1. Le noyau réflexif derrière le livre

Avant d'entrer dans le détail des builtins, il vaut la peine de cartographier les usages implicites que les chapitres précédents ont déjà faits de la méta-programmation. Cela permettra au lecteur de comprendre que le présent chapitre ne crée rien de nouveau — il *nomme* ce qu'il utilisait déjà.

Au chapitre 5, lorsque la quasiquote (backtick) a été introduite, le livre a posé la première brique. `` `[$a + 2] `` n'est pas une expression évaluée — c'est une **valeur** de type liste VVL, dont les `$a` peuvent être substitués au moment d'une réécriture. La quasiquote permet déjà de **construire du code comme donnée**, mais elle ne dit rien sur la façon de le **parcourir**, l'**inspecter**, le **sérialiser**, ou le **réinjecter** dans le runtime. C'est précisément la fonction des builtins de ce chapitre.

Au chapitre 12, le système `ontology:rule` accepte des expressions VVL comme corps de règle. Quand le développeur écrit `[ontology:rule "diagnosis" when ($p ~~ "Fever") then ($p.diag = "Flu")]`, les segments `($p ~~ "Fever")` et `($p.diag = "Flu")` sont stockés *en tant que formes VVL* dans la base de règles, puis réévalués au moment du déclenchement avec les liaisons `$p` appropriées. C'est de la méta-programmation déguisée en sucre syntaxique : le moteur d'inférence stocke des **AST** et les **réexécute**.

Au chapitre 17, l'induction LLM zero-shot synthétise du texte VVL — par exemple `[ontology:rule "r1" when ... then ...]` — qu'elle injecte ensuite dans le contexte. Pour que cette injection fonctionne, il a fallu *quelque part* convertir une **chaîne** (sortie du LLM) en une **forme VVL exécutable**. Le builtin qui sera nommé `[load]` dans la section suivante est précisément cet pont.

Au chapitre 22, l'agent autonome `[agent]` génère du code à la volée. La boucle est : l'agent produit une expression candidate sous forme textuelle, l'évalue, observe le résultat, corrige si nécessaire. Cette boucle entière repose sur deux primitives : *évaluer une expression construite dynamiquement*, et *intercepter les erreurs sans interrompre le programme principal*. La première est `[evaluate]`, la seconde est `[attempt]`.

Au chapitre 24, la commande `[expert]` compile une description en français en une suite de règles `[fuzzy:rule]`. Cette compilation est, là encore, de la méta-programmation : transformation de texte → texte VVL → forme VVL → exécution. Toute la chaîne est invisible pour l'utilisateur, mais elle repose sur les builtins de ce chapitre.

Au chapitre 27, l'ILP sound-and-complete énumère des clauses Prolog candidates, les évalue contre un dataset d'exemples positifs et négatifs, et retient celles dont la couverture est correcte. L'énumération produit des **formes VVL** (les clauses Prolog ont une réification VVL), l'évaluation est une **réinjection dans le runtime**, et la décision finale repose sur une **inspection structurelle** des clauses. Trois usages de la méta-programmation dans un seul builtin.

Cette cartographie rétrospective devrait suffire à convaincre le lecteur que la méta-programmation n'est pas un *raffinement avancé* qu'on ajouterait à VVL en bout de course : c'est la **fondation discrète** qui a porté l'ensemble du livre. Le présent chapitre se contente d'amener cette fondation à la lumière et de la rendre disponible **directement**, au lieu de la laisser tapissée derrière des builtins de haut niveau.

28.2. Code-comme-donnée : le triangle load / evaluate / dump

Le cœur de la méta-programmation VVL tient dans un triangle de trois sommets — **source string**, **form VVL**, **valeur évaluée** — et de quatre flèches qui circulent entre eux. La figure ci-dessous donne la vue d'ensemble ; les sections qui suivent l'examinent flèche par flèche.

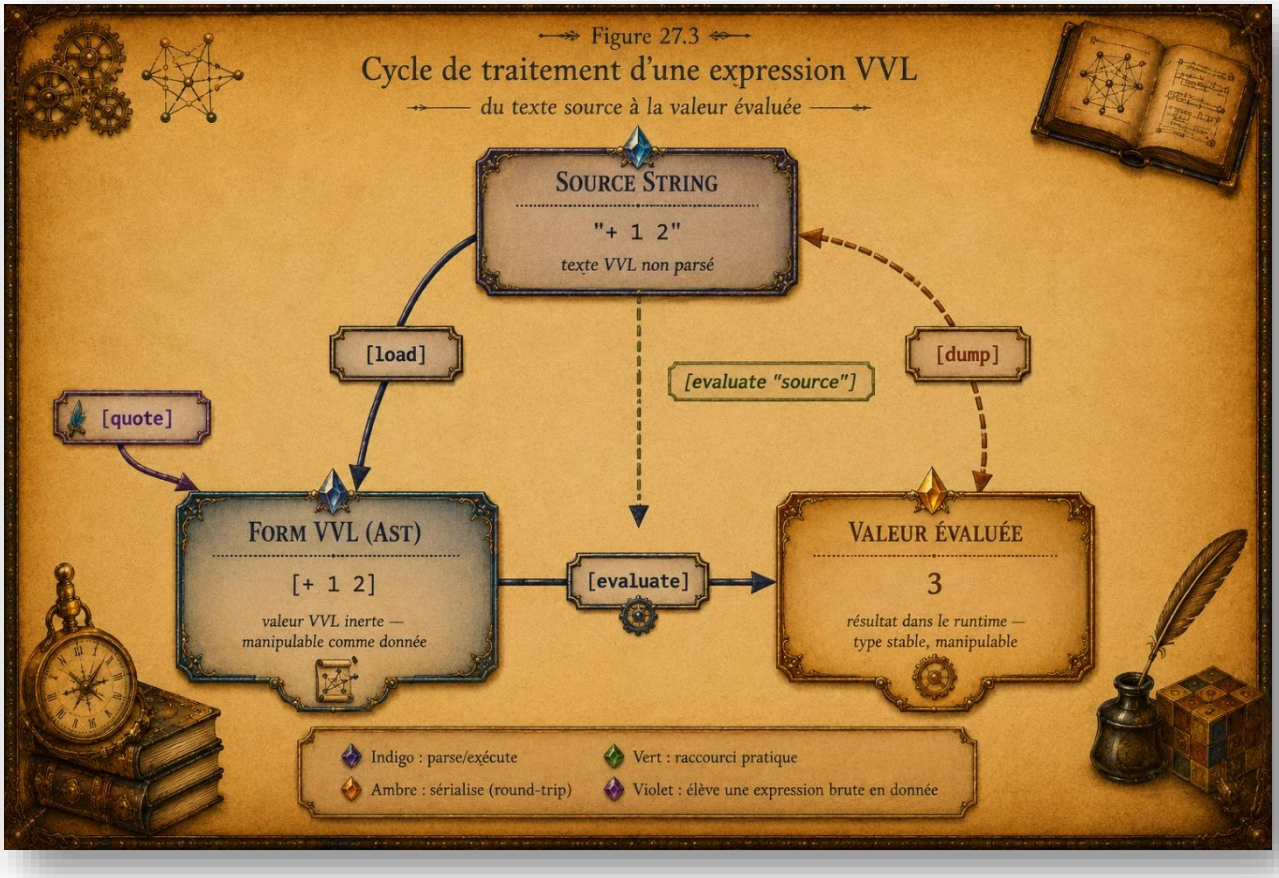


Figure 28.1 — *Le triangle homoiconique de VVL. Chaque flèche est un builtin ; chaque nœud est un type de représentation du programme.*

28.2.1. Du texte à la forme : [load]

`[load "..."]` accepte une **chaîne** contenant du code VVL et retourne la **forme VVL inerte** correspondante — l'arbre de syntaxe, mais en tant que valeur de type liste, manipulable comme n'importe quelle autre donnée du langage. Aucune évaluation n'a lieu : c'est strictement du parsing.

```
001  # [load] : source string -> forme VVL inerte
002
003  $form = [load "[+ 1 2]"]
004  [log [dump $form]]                # => "[+ 1 2]"
005  [log [type-of $form]]             # => "list"
006
007  # La forme est une liste VVL ordinaire ; on peut la manipuler.
008  [log [length $form]]              # => 3
009  [log [getValue $form 1]]          # => 1
```

Si la source contient une **seule** expression de haut niveau, `[load]` retourne cette expression. Si elle contient **plusieurs** statements, `[load]` retourne une liste VVL contenant la séquence. Cette distinction permet d'écrire des chargeurs uniformes :

```
# Une seule expression
$single = [load "[+ 1 2]"]
[log [dump $single]]              # => "[+ 1 2]"

# Plusieurs statements : une liste de formes
$multi = [load "$a = 1\n$b = 2\n[+ $a $b]"]
[log [length $multi]]             # => 3
```

`[load]` ne lève pas d'erreur sur un programme syntaxiquement valide mais sémantiquement absurde : il ne *connaît* pas la sémantique. Si l'on charge `"[$missing_callable 1 2]"`, la forme produite est parfaitement licite — c'est seulement à l'évaluation que le runtime se rendra compte que `$missing_callable` n'est pas lié. Cette séparation propre entre **parsing** et **exécution** est l'une des forces de l'approche : on peut transporter, stocker, transformer, ou sérialiser des formes VVL sans craindre des effets de bord cachés.

28.2.2. De la forme à la valeur : [evaluate] et [execute]

`[evaluate]` accepte une forme VVL — typiquement issue de `[load]` ou de `[quote]` — et la **réinjecte dans le runtime** courant, comme si elle y avait été écrite directement. Le résultat est la valeur que cette forme produit lorsqu'elle est évaluée.

```
# Le chemin complet source -> form -> valeur

$src   = "(1 + 2)"
$form  = [load $src]
$value = [evaluate $form]
[log $value]                      # => 3

# [evaluate] accepte aussi directement une source.
[log [evaluate "(10 * 5)"]]       # => 50

# Et bien sûr une forme issue de [quote].
$form2 = [quote [+ 100 200]]
[log [evaluate $form2]]           # => 300
```

Le double accès — source string **ou** forme VVL — fait de `[evaluate]` un point d'entrée unifié pour la réflexion. C'est le raccourci vert pointillé dans le diagramme du triangle homoiconique. Note typographique importante : `+` est un opérateur **infix** en VVL, pas une commande crochetée. Pour l'utiliser dans une source string, on écrit `"(1 + 2)"` et non `"[+ 1 2]"`. Cette distinction n'apparaît qu'au niveau de la source ; dans les **formes** issues de quote ou de quasiquote, les deux notations coexistent.

Le builtin compagnon `[execute]` ressemble à `[evaluate]` mais en diffère sur un point : il n'évalue **pas ses arguments en amont**, ce qui le rend utile pour exécuter une expression *littérale* sans déclencher la machinerie de substitution habituelle.

```
# [execute] : exécute sans pré-évaluer les arguments
[execute [log "hello from execute"]]
# => hello from execute
```

Le cas d'usage typique de `[execute]` est l'écriture de macros simples qui doivent intercepter une expression telle qu'elle est écrite, sans que le runtime ne tente d'abord de l'évaluer comme un argument ordinaire.

28.2.3. Élever une expression brute : [quote]

`[quote]` est la flèche violette du diagramme. C'est un builtin **raw** — c'est-à-dire qu'il ne pré-évalue pas ses arguments — qui retourne l'expression qu'on lui passe **telle quelle**, sans l'exécuter. Le résultat est une forme VVL ; il est ensuite manipulable comme données ou évaluable ultérieurement.

```
# [quote] : élève une expression brute en valeur

$raw = [quote [$missing_callable 1 2]]
[log [dump $raw]]                # => "[$missing_callable 1 2]"

# Avec plusieurs arguments, [quote] retourne une liste de formes.
$prog = [quote [log "deferred"]]
[evaluate $prog]                   # => deferred
```

La différence avec la **quasiquote** (backtick) — déjà rencontrée au chapitre 5 — est subtile mais importante. La quasiquote *supporte* la substitution par `$variable` et l'évaluation locale par `,expr`, tandis que `[quote]` est plus rigide : ce qu'on lui passe est préservé **textuellement**, sans aucune réécriture, et c'est ensuite à l'utilisateur d'appeler `[evaluate]` au moment opportun.

Dans la pratique, les deux outils se complètent. La quasiquote sert à *construire* des templates d'expressions où certains slots sont remplis dynamiquement (typiquement dans le corps d'un `[keyword]`). `[quote]` sert à *figer* une expression dans son état brut pour la transporter, l'inspecter, ou la stocker — par exemple pour la passer à un journal d'audit, à un agent, ou à un test d'introspection.

28.2.4. Le retour : [dump]

La quatrième flèche du triangle ferme la boucle. `[dump $value]` accepte une valeur VVL **évaluée** — un nombre, une chaîne, une liste, un JSON, une forme — et retourne sa **représentation canonique sous forme de chaîne**. Cette chaîne, à son tour, peut être passée à `[load]` pour reconstruire la valeur initiale, ou à `[evaluate]` pour la réexécuter dans un autre contexte.

```
# [dump] : valeur -> source string canonique

$data = `[a b 1]
[log [dump $data]]                # => "[a b 1]"

# Round-trip complet : forme -> source -> forme.
$f1 = [quote [+ 1 2]]
$s  = [dump $f1]                    # => "[+ 1 2]"
$f2 = [load $s]
[log [evaluate $f2]]              # => 3
```

`[dump]` est le pendant de `[load]` : si `[load]` est la flèche d'entrée du triangle (string → form), `[dump]` est la flèche de sortie (form → string). Le round-trip est garanti pour toutes les valeurs qui ont une représentation VVL lisible — nombres, chaînes, listes, formes, JSON. Pour les valeurs **opaques** (objets COM, handles WebDriver, semaphores, canaux), `[dump]` retourne une chaîne d'inspection plutôt qu'une représentation reloadable ; c'est une limite assumée du système.

28.2.5. Quatre rôles, une matrice

Pour fixer définitivement la distinction entre les quatre builtins, le tableau ci-dessous résume leur signature et leur rôle dans le triangle homoiconique.

Builtin	Argument	Retour	Rôle
[load "src"]	string	form VVL	parser sans exécuter
[evaluate $x]	string OU form	valeur	exécuter dans le runtime
[execute expr]	expression brute	valeur	exécuter sans pré-évaluer args
[quote expr]	expression brute	form VVL	figer une expression telle quelle
[dump $v]	valeur évaluée	string	sérialiser en source canonique

Une heuristique pratique pour choisir : si l'on part d'une **chaîne**, on commence par `[load]`. Si l'on part d'une **expression brute écrite à la main**, on utilise `[quote]`. Si l'on possède déjà une **forme** et qu'on veut l'exécuter, c'est `[evaluate]`. Si l'on veut **persister** une valeur sous une forme reloadable, c'est `[dump]`.

Une dernière subtilité mérite d'être notée. `[evaluate]` accepte indifféremment une string ou une form ; le raccourci `[evaluate "source"]` court-circuite donc le triangle en passant directement du sommet à la valeur. Ce raccourci est commode mais opaque : la forme intermédiaire n'est pas observable. Pour les usages où l'on veut **inspecter** ou **transformer** le programme avant de l'exécuter — par exemple pour vérifier qu'un agent LLM ne tente pas d'invoquer un builtin interdit —, il faut décomposer en `[load]` puis `[evaluate]`, avec une étape d'inspection entre les deux. Cette **décomposition explicite** est la base d'à peu près tous les mécanismes de sécurité dynamiques de VVL : politique d'agent (chapitre 22), sandbox des règles induites (chapitre 27), validation des dialectes (chapitre 29).

28.3. Introspection : type-of, getType, inspectWord

Manipuler du code-comme-donnée exige une couche d'introspection : sans elle, le runtime ne peut pas prendre de décision sur les formes qu'il reçoit, et le développeur ne peut pas écrire d'agents, de validateurs ou d'optimiseurs génériques. VVL expose trois builtins d'introspection complémentaires : `type-of` pour les *valeurs courantes*, `getType` pour les *valeurs typées* (chapitre 31), et `inspectWord` pour le *détail token-par-token*.

28.3.1. Le nom de type stable : [type-of]

`[type-of $value]` retourne une **chaîne minuscule** désignant le type de la valeur reçue. Le nom est stable d'une version à l'autre du runtime, ce qui en fait la primitive recommandée pour les comparaisons de type en code utilisateur.

```
# [type-of] : nom de type stable, en minuscules

[log [type-of 42]]                # => "num"
[log [type-of "hello"]]           # => "string"
[log [type-of `[a b 1]]]          # => "list"
[log [type-of [JSON "{}"]]]       # => "json"
[log [type-of [quote [+ 1 2]]]]   # => "list"
```

Le retour est volontairement *insensible au contexte d'usage* : `[+ 1 2]` retourne `"list"` parce que c'est, au niveau du langage, une liste VVL, et non un type spécial *expression*. Cette uniformité est essentielle pour le code méta-programmatique : un agent qui reçoit une expression candidate peut la manipuler avec les opérations de liste standard (`length`, `getValue`, `append`, `slice`) sans avoir à apprendre un sous-langage d'AST.

Pour les **valeurs scalaires typées** introduites au chapitre 31 — `EMAIL`, `URL`, `DATE`, `UUID` —, `type-of` retourne le nom en minuscules (`"email"`, `"url"`, etc.). C'est `[getType]` qui retournera l'étiquette en majuscules utilisable comme tag formel.

```
# type-of vs getType pour les valeurs typées

$mail = [EMAIL "nky@prosuite.com"]
[log [type-of $mail]]             # => "email"
[log [getType $mail]]             # => "EMAIL"
```

28.3.2. Inspection token-par-token : [inspectWord]

Là où `type-of` renvoie un nom abstrait, `[inspectWord $w]` ouvre le capot et expose la représentation interne du mot ou token reçu : son type concret, son contenu brut, ses éventuelles métadonnées de parsing. Le builtin est principalement destiné au débogage du compilateur ou au développement de plugins qui doivent prendre des décisions fines sur la nature exacte d'un atome.

```
# [inspectWord] : détail du token et de sa représentation interne

[inspectWord $a]                    # détail de la variable $a
[inspectWord [quote +]]             # détail de l'opérateur +
```

Dans la plupart des programmes utilisateur, `inspectWord` n'a pas sa place : c'est un outil de **réflexion profonde** réservé aux cas où la distinction entre, par exemple, un mot symbolique non lié et un identifiant déjà résolu doit être prise en compte. La présentation détaillée du format de sortie est repoussée à l'annexe de référence.

28.4. La frontière import / load / evaluate

Les trois builtins `[import]`, `[load]` et `[evaluate]` peuvent sembler à première vue redondants : tous trois manipulent du code à partir d'un support textuel. La confusion est si fréquente que la documentation du langage la traite explicitement comme un **point de doctrine**. Le tableau ci-dessous fixe une fois pour toutes la division du travail.

Builtin	Entrée	Effet	Quand l'utiliser
[load "src"]	string en mémoire	parse, ne lance pas l'exécution	inspecter ou transformer une expression avant exécution
[evaluate $f]	string ou form	exécute dans le contexte courant	exécuter du code construit dynamiquement (agent, ILP, expert)
[import "f.vvl"]	chemin de fichier	lit le fichier ET l'exécute	charger un module à l'ouverture d'une session

Les trois builtins ne sont pas concurrents ; ils sont **complémentaires**. Un cas d'usage courant les combine : un script d'initialisation au démarrage d'une session VVL utilise `[import]` pour charger une bibliothèque de keywords définie ailleurs (chapitre 30 : modules) ; à l'intérieur d'une boucle, un agent utilise `[load]` puis `[evaluate]` pour exécuter une expression LLM générée à la volée, en s'autorisant une étape d'inspection au milieu pour vérifier que la forme ne tente pas d'invoquer un builtin interdit ; et au moment de persister un état, le programme utilise `[dump]` pour sérialiser ses variables avant de les écrire dans le workspace.

Une erreur fréquente consiste à utiliser `[import]` là où `[load]` suffirait, ce qui produit des effets de bord cachés (toutes les définitions du fichier sont ajoutées au contexte courant). La règle est simple : si l'on veut **modifier le contexte courant**, on utilise `[import]` ; si l'on veut **manipuler une expression comme donnée**, on utilise `[load]`.

28.5. [keyword] : la macro utilisateur, approfondissement

Le builtin `[keyword]` a été introduit dès le chapitre 4 comme mécanisme de **définition d'une commande utilisateur**. Le moment est venu de l'examiner plus en profondeur, parce qu'il est l'incarnation la plus fréquente de la méta-programmation dans le code de tous les jours. Là où `[load]` et `[evaluate]` sont des outils de bas niveau pour les développeurs avancés, `[keyword]` est l'outil de tous les utilisateurs qui ont besoin de raccourcir une expression répétée ou de créer un mini-DSL.

28.5.1. Anatomie d'une keyword

Une `keyword` est une **macro à substitution de paramètres** : son corps est une expression VVL dans laquelle les jetons `@1`, `@2`, … désignent les arguments passés au site d'appel. Le corps est enveloppé dans une quasiquote (backtick) pour que VVL ne tente pas de l'évaluer **au moment de la définition** — il sera évalué **au moment de chaque appel**, avec les `@i` substitués.

```
001  # Une keyword simple : substitution d'un seul argument
002
003  [keyword greet
004      `[log "Hello, " @1 "!"]
005  ]
006
007  [greet "Alice"]                    # => Hello, Alice!
008  [greet "Bob"]                      # => Hello, Bob!
```

Le corps `[log "Hello, " @1 "!"]` est stocké tel quel dans la définition de `greet`. À chaque appel, le runtime remplace `@1` par l'argument passé et évalue le résultat. Cette double phase — *stockage différé* puis *substitution-évaluation* — est exactement le pattern méta-programmatique du triangle : la forme est conservée, et l'évaluation est déclenchée à un moment ultérieur, avec un contexte différent.

28.5.2. Keyword multi-arguments et calcul interne

Une `keyword` peut accepter plusieurs arguments — `@1`, `@2`, `@3` — et combiner des calculs internes en utilisant `[do (...)]` pour séquencer.

```
001  # Une keyword à trois arguments avec calcul interne
002
003  [keyword add3
004      `[do ($result = (@1 + @2 + @3)) $result]
005  ]
006
007  [log [add3 10 20 30]]              # => 60
008  [log [add3 1 2 3]]                 # => 6
```

Noter le rôle du `[do (...)]` : il introduit une **séquence d'évaluation** dont la dernière expression est la valeur retournée. C'est l'idiome standard quand le corps d'une keyword a besoin d'effectuer un calcul intermédiaire avant de retourner un résultat.

28.5.3. La limite des quasiquotes imbriquées

Une subtilité importante du système : la substitution `@i` **ne descend pas** dans les quasiquotes imbriquées à l'intérieur du corps d'une keyword. Si l'on écrit

```
[keyword wrap
    `[do $inner = `[+ @1 1] [evaluate $inner]]
]

# [wrap 5] => attendu : 6, obtenu : erreur (@1 non substitué)
```

le `@1` à l'intérieur du backtick imbriqué **n'est pas remplacé** au moment de l'appel à `wrap`, parce que la quasiquote interne forme un nouveau niveau de gel. Cette limite est documentée et délibérée — elle empêche des récursions de substitution complexes qui rendraient le système difficile à raisonner — mais elle exige de l'utilisateur attentif qu'il en tienne compte. Pour les cas où une telle imbrication est nécessaire, la solution recommandée est de définir une seconde `keyword` qui prend le résultat intermédiaire en argument.

28.5.4. Quand utiliser keyword, quand utiliser une fonction

VVL offre, parallèlement à `keyword`, le mécanisme classique des **fonctions** (introduit au chapitre 5). La distinction entre les deux est qualitative : une `keyword` est un *macro lexical*, dont les arguments ne sont **pas pré-évalués** — ils sont substitués tels quels dans le corps. Une fonction, à l'inverse, évalue ses arguments **avant** de les passer au corps.

Cette différence importe quand l'argument à un effet de bord ou qu'on ne veut **pas** qu'il soit évalué automatiquement. Par exemple, une `keyword` de log conditionnel peut accepter une expression et ne l'évaluer **que si le niveau de log est suffisant** :

```
# Une keyword qui n'évalue son argument qu'à la demande

[keyword logDebug
    `[when ($logLevel >= 4) [log "[DEBUG] " @1]]
]

$logLevel = 2
[logDebug [computeExpensiveDiagnostic $patient]]
# La fonction expensive n'est PAS appelée car le niveau est trop bas.
```

Une fonction classique, à l'inverse, évaluerait `[computeExpensiveDiagnostic $patient]` *avant* de le passer au corps de `logDebug`, ce qui annulerait le bénéfice du filtrage. La règle pratique est donc : **fonction** pour les calculs purs et les valeurs déjà connues ; **keyword** pour les abstractions de contrôle, les DSL légers, et les optimisations par évaluation paresseuse.

28.6. Utilitaires de méta et de débogage

Autour du triangle homoiconique gravite une constellation d'utilitaires qui ne sont pas, à proprement parler, des outils de méta-programmation, mais qui en sont les compagnons quotidiens : commandes de logging, de documentation, et de manipulation de l'environnement de session. Le présent chapitre les regroupe ici pour ne pas avoir à les disperser dans d'autres chapitres.

28.6.1. Aide en ligne : [help]

`[help "command"]` retourne la documentation interne d'une commande. Le runtime maintient pour chaque builtin une notice d'aide concise — signature, sémantique, exemple court — accessible sans quitter la session.

```
[help "map"]
# Affiche la documentation de map : usage, signature, exemple.

[help "ontology:rule"]
# Documentation du builtin de règle ontologique.
```

Cette aide est servie depuis le même corpus que la documentation web officielle. Pour des recherches plus larges (par mot-clé, par concept), le système expose également un index de la base de connaissance via `[kb:search]` (chapitre 19), qui sort du périmètre de ce chapitre.

28.6.2. Logging structuré : log, logTable, logImage

Trois builtins composent la famille `log*` : `[log]` pour un message texte, `[logTable]` pour une donnée tabulaire, `[logImage]` pour une image. Tous trois envoient leur argument à la **sortie d'inspection** de la session — la console dans les modes terminaux, le panneau d'inspection dans les modes notebook ou GUI.

```
[log "computation complete"]
[log "etape" $i "/" $n]

# logTable est idiomatique pour les resultats de query
$rows = [db query "select id, name, score from employees"]
[logTable $rows]

# logImage rend une image dans les frontends qui le supportent
$plot = [renderChart $data]
[logImage $plot]
```

Les trois builtins partagent le même canal, mais l'usage a fixé une convention utile. [log] sert indifféremment à la sortie attendue par l'appelant et à la trace d'exécution ; dans un lancement batch, c'est le fichier de capture qui les sépare. [logTable] est réservé aux résultats tabulaires — requêtes Prolog, lignes CSV, matrices de scoring — parce que les frontends savent les rendre sous forme de tableau plutôt que de texte brut. [logImage] est réservé aux images. Une dernière convention est largement respectée dans la base de code, en particulier dans les tutoriels : la dernière ligne est presque toujours [log "TASK COMPLETE"], qui signale la fin d'exécution aux suites de tests d'intégration.

28.6.3. Paramètres de tâche : [task-param]

Quand un programme VVL est lancé non pas interactivement mais via la **file d'attente de tâches** (introduite au chapitre 21), il reçoit ses entrées sous forme de paramètres nommés. `[task-param "key"]` accède à un paramètre par son nom.

```
# Un script lance via la queue avec parametres "target_url" et "timeout"

$url     = [task-param "target_url"]
$timeout = [task-param "timeout"]

[log "Crawl de " $url " avec timeout " $timeout "s"]
```

Le builtin est inerte hors d'un contexte de tâche : appelé depuis une session interactive, il retourne une valeur vide ou lève une erreur selon le mode. Pour les scripts qui veulent fonctionner dans les deux modes, l'idiome est de combiner `task-param` avec un fallback :

```
$url = [task-param "target_url"]
[when (! $url) ($url = "http://default.local")]
```

28.6.4. Exploration du workspace : [file-tree]

Le dernier utilitaire de cette section, `[file-tree path]`, retourne sous forme de structure JSON l'arborescence du répertoire passé en argument. Le résultat est navigable comme n'importe quelle valeur JSON VVL, et l'utilisateur peut donc en extraire des chemins, calculer des statistiques, ou prendre des décisions sur la base de l'état du système de fichiers.

```
$tree = [file-tree "C:/projects/myapp"]
[log $tree]

# Combiner avec map pour extraire tous les .vvl du repertoire
$vvlFiles = [filter $tree.children
    `[endsWith @1.name ".vvl"]
]
```

`[file-tree]` est principalement utilisé par les outils d'introspection de projet — comme les commandes `kb:scan` du chapitre 19 ou les agents qui doivent décider quel fichier charger. Pour des manipulations plus fines (lecture de contenu, écriture, suppression), VVL expose une famille `file:*` distincte qui sort du périmètre de ce chapitre.

28.7. [collect] / [keep] : génération de séries par effets de bord

Avec cette section, on bascule de la **méta-programmation** au premier des trois helpers d'inspiration REBOL. Le triplet `collect`/`keep`/`attempt`/`object` partage avec le triangle homoiconique une parenté philosophique — il favorise le *style data-first* dans lequel les valeurs sont aussi des structures

observables — mais il ne fait pas, à proprement parler, de méta-programmation. Ces helpers consolident plutôt l'ergonomie quotidienne du langage.

28.7.1. Le pattern : un buffer caché

`[collect expr1 expr2 ...]` est un builtin **raw** qui évalue ses arguments un par un, en exposant pendant l'évaluation un **buffer caché** dans lequel chaque appel à `[keep value]` ajoute une valeur. À la fin de l'évaluation, `collect` retourne la liste des valeurs ainsi accumulées. L'effet est celui d'une boucle générative écrite sans variables intermédiaires.

```
# [collect] / [keep] : construire une liste par effets de bord

[collect
    [keep 1]
    [keep (1 + 2)]
    [keep "x"]
]
# => [1 3 "x"]
```

Le résultat est exactement le même que `` `[1 3 "x"] ``, mais la **construction** est différente : au lieu d'écrire la liste explicitement, on déclenche son remplissage par une séquence d'appels à `keep`. Cette inversion devient utile quand le remplissage est **conditionnel** ou **dérivé d'un calcul**.

```
# Cas réaliste : filtrer et transformer en une seule passe

$candidats = [list
    [object [name "Alice" age 30]]
    [object [name "Bob"   age 17]]
    [object [name "Cara"  age 24]]
]

$majeurs = [collect
    [for $c $candidats
        [when ($c.age >= 18) [keep $c.name]]
    ]
]
# => ["Alice" "Cara"]
```

Cet idiome remplace avantageusement le couple `[filter ...] [map ...]`, qui produirait le même résultat au prix de deux passes sur la liste et d'un fonctionnel imbriqué moins lisible. Pour les pipelines de transformation purement *streaming* — sans saut conditionnel —, le couple `filter`/`map` reste plus expressif ; mais dès qu'il s'agit de mélanger filtrage et logique conditionnelle, `collect`/`keep` est plus direct.

28.7.2. Isolation des collects imbriqués

Une subtilité importante : les `collect` imbriqués sont **isolés**. Un `keep` à l'intérieur d'un `collect` interne n'affecte **que** ce collect interne, jamais le collect englobant. Le runtime maintient une pile de buffers et n'ajoute chaque valeur qu'au sommet.

```
# Les keeps imbriqués respectent la portée du collect le plus proche

[collect
    [keep "outer"]
    [keep [collect [keep "inner"]]]
]
# => ["outer" ["inner"]]
```

Cette propriété d'isolation est essentielle pour la **composition** : un programme peut construire une fonction qui utilise `collect` en interne, sans que cette utilisation interfère avec un éventuel `collect` extérieur. Sans isolation, les helpers `collect` seraient inutilisables dans les bibliothèques.

28.7.3. Erreur : keep hors collect

Appeler `[keep value]` en dehors de tout `[collect]` est une **erreur** levée par le runtime. Cette stricte exigence est volontaire : elle évite que des appels à `keep` égarés dans le code ne soient absorbés silencieusement par un collect englobant éloigné, ce qui produirait des résultats imprévisibles.

```
[keep 42]
# => RuntimeError : [keep] called outside [collect]
```

Pour les codes qui veulent rester robustes face à un appel hors-contexte, l'idiome est d'envelopper `keep` dans `[attempt ...]` (section suivante) — ce qui transforme l'erreur runtime en un statut sérialisable.

28.8. [attempt] : la gestion d'erreur sérialisable

VVL dispose, depuis le chapitre 4, du couple `[try try-body catch-body]` pour la gestion d'erreurs au style classique : un bloc principal, un bloc de rattrapage, et une variable spéciale `$__error` qui porte l'erreur capturée. Ce mécanisme est efficace pour le **contrôle de flux** local, mais il est mal adapté à un cas d'usage devenu fréquent : la **persistance** ou la **transmission** d'un statut d'opération, par exemple vers un agent (chapitre 22), un plugin (chapitre 33), ou un système de tests (chapitre 32).

`[attempt expr]` répond à ce besoin. Il accepte une expression VVL (en mode raw, sans pré-évaluation) et **retourne toujours un objet JSON** décrivant le résultat. En cas de succès, l'objet porte la valeur produite ; en cas d'échec, il porte le type d'erreur et son message. Le programme principal n'est jamais interrompu — `attempt` *transforme une exception en donnée*.

28.8.1. Anatomie d'un retour [attempt]

```
# Cas succes : retour avec ok=true

$r = [attempt (1 + 2)]
[log $r]
# => {"ok":true, "type":"num", "value":3, "dump":"3"}

# Cas echec : retour avec ok=false et un sous-objet error

$r2 = [attempt [keep 42]]           # keep hors collect
[log $r2]
# => {"ok":false,
#     "error":{"type":"RuntimeError",
#              "message":"[keep] called outside [collect]"}}
```

Le champ `type` (en cas de succès) reprend la valeur de `[type-of]` sur le résultat ; le champ `value` porte la valeur elle-même, et `dump` en porte la représentation textuelle canonique (issue de `[dump]`). Cette uniformité est précisément ce qui rend `attempt` adapté aux **journaux**, aux **réponses d'agent**, et aux **suites de tests** : un seul format, qu'on peut écrire en base de données, transmettre par le réseau, ou inspecter en boucle.

28.8.2. attempt vs try : la matrice de décision

Critère	`[try try-body catch-body]`	`[attempt expr]`
Type de retour	la valeur de try-body ou de catch-body	toujours un objet JSON {ok, ...}
Contrôle de flux	permet de continuer en cas d'erreur	l'erreur est encapsulée, le flux continue
Sérialisation	l'erreur est un état runtime $__error	l'erreur est une donnée JSON transportable
Usage typique	gestion locale d'une erreur prévisible	audit, agent, plugin, test

La règle pratique : si l'on veut **continuer un calcul** en cas d'erreur, on utilise `[try]`. Si l'on veut **enregistrer ou transmettre** un statut, on utilise `[attempt]`. Les deux approches coexistent sans conflit, et l'on peut très bien envelopper un `[try]` dans un `[attempt]` pour combiner contrôle local et persistance.

28.8.3. La compatibilité ascendante : try:json

Pour les utilisateurs qui ont commencé à programmer avec une version antérieure du runtime, le nom `try:json` reste disponible comme **alias** de `attempt`. Le nom canonique est désormais `attempt` parce qu'il a une **signature différente** de `try` — un seul argument, pas de catch-body — et qu'il vaut mieux signaler cette différence par un nom distinct. Les documentations à venir éviteront `try:json` ; les programmes existants continuent toutefois de fonctionner sans modification.

28.9. [object] : l'objet de données léger

Le troisième helper REBOL répond à un besoin que les chapitres précédents ont contourné par des ad-hoc multiples : la **création d'un objet de données** — c'est-à-dire d'une structure clé/valeur — sans avoir à descendre au niveau de l'API JSON. `[object [key value ...]]` est exactement cela : un builtin raw qui accepte une liste de paires alternées et retourne une valeur de type `VVL_Json`.

```
001  # Creation d'un objet de donnees
002
003  $o = [object [name "Alice" age 30 role "engineer"]]
004
005  [log $o.name]                    # => Alice
006  [log $o.age]                     # => 30
007  [log $o.json]                    # => {"name":"Alice","age":30,"role":"engineer"}
008  [log [type-of $o]]               # => json
```

Parce que `object` retourne un `VVL_Json`, l'objet est immédiatement compatible avec **toute** la machinerie JSON déjà disponible dans VVL : `jsonGetValue`, `json-encode`, `json-path-get`, l'accès par navigation pointée (`$o.name`), la sérialisation et la désérialisation. Le builtin n'introduit donc pas de **nouveau type** ; il propose un **constructeur ergonomique** pour un type qui existait déjà.

28.9.1. object n'est pas une classe

Il est essentiel de clarifier ce que `object` **n'est pas**. VVL n'a pas, et n'aura vraisemblablement pas, de système de classes orienté-objet à la Python ou Java — pas de hiérarchies d'héritage, pas de méthodes attachées, pas de protocoles formels. Le langage tient son expressivité de l'**ontologie** (chapitre 12) pour le typage formel et la classification, et de l'`object` *pour les données ad-hoc* qui n'ont besoin d'aucune machinerie supplémentaire. Cette séparation est volontaire : elle évite de dupliquer entre objet-langage et ontologie-langage des mécanismes de typage qui finiraient par se contredire.

L'usage typique d'`object` est donc le suivant : il sert à transporter des **enregistrements** entre les builtins, à représenter des **résultats** structurés, à composer des **paramètres** nommés pour un appel ultérieur. Quand la structure est destinée à durer et à recevoir un comportement, on la promeut en **concept ontologique** (`[ontology:define]`) ; mais pour les structures transitoires de la vie quotidienne du programme, `object` suffit.

28.9.2. Composition avec le triangle homoiconique

Une dernière propriété rend `object` particulièrement adapté à la méta-programmation : sa valeur peut être **dumpée** et **rechargée** sans perte. Un `object` est donc un véhicule naturel pour le transport d'état entre sessions, vers un journal, ou à travers un canal réseau.

```
# Round-trip d'un objet : valeur -> source -> valeur

$o = [object [user "alice" score 0.92 tags `["expert" "ml"]]]
$serialized = [dump $o]
[log $serialized]
# => {"user":"alice", "score":0.92, "tags":["expert","ml"]}

$reloaded = [evaluate $serialized]
[log $reloaded.user]              # => alice
```

Cette propriété fait d'`object` le compagnon idéal d'`attempt` : un résultat d'opération est un `object` au format `{ok, value, ...}` qu'on peut journaliser, transmettre, et reconstruire à l'identique. La boucle est bouclée.

28.10. Synthèse : le noyau réflexif et le style de programmation

Le présent chapitre a achevé un travail commencé implicitement aux toutes premières pages du livre : nommer la couche méta qui rend VVL composable. Onze sections, douze builtins, une matrice, un triangle. Le moment est venu de revenir au niveau de la philosophie pour comprendre **pourquoi** cette couche est structurée comme elle l'est, et **comment** elle s'articule avec le reste du langage.

28.10.1. L'homoiconicité comme principe d'économie

VVL est homoiconique non pas par tradition lispienne, mais par **économie**. Si le langage avait choisi de représenter les programmes par un type *expression* distinct du type *liste*, il aurait dû dupliquer toute la machinerie de manipulation : un `length` pour les listes, un `expression-arity` pour les expressions ; un `getValue` pour l'un, un `expression-operand` pour l'autre. Cette duplication aurait alourdi le langage sans rien apporter.

Le choix opposé — *les programmes sont des listes ordinaires* — réduit le nombre de concepts et accroît la composabilité : un agent qui veut manipuler le code généré par un LLM dispose immédiatement, sans apprentissage supplémentaire, des opérations qu'il connaît déjà sur les listes. C'est cette économie qui rend le triangle `load`/`evaluate`/`dump` si naturel à utiliser : il ne s'ajoute pas au langage, il **expose** une propriété que le langage possédait déjà.

28.10.2. Les trois niveaux d'évaluation

Si l'on prend du recul, on observe que VVL offre maintenant **trois niveaux d'évaluation** distincts, chacun correspondant à un usage différent :

Niveau	Mécanisme	Quand utiliser
1. Évaluation directe	écrire l'expression dans le code source	cas par défaut, code statique
2. Évaluation différée par quasiquote	backtick + substitution $var ou @i	templates, corps de keyword, macros
3. Évaluation réflexive	[load] + inspection + [evaluate]	code généré par LLM, agents, ILP, dialectes

Ces trois niveaux ne sont pas mutuellement exclusifs : un même programme peut les combiner. Un agent typique utilise le **niveau 1** pour sa boucle de contrôle, le **niveau 2** pour ses templates de prompts, et le **niveau 3** pour exécuter les expressions VVL que le LLM propose. La maîtrise des trois niveaux et de leur articulation est précisément ce qui distingue un développeur VVL avancé d'un débutant.

28.10.3. Les helpers REBOL comme parachèvement ergonomique

Le triplet `collect`/`attempt`/`object` (auquel s'ajoute `keep` qui forme un couple indivisible avec `collect`) ne fait pas, à proprement parler, de méta-programmation. Mais sa présence dans ce chapitre n'est pas arbitraire. Les trois helpers parachèvent l'ergonomie d'un langage *data-first* :

`collect` rend triviale la **génération conditionnelle** de listes, opération qui revient en permanence dans les boucles d'agent et les pipelines d'induction. `attempt` rend triviale la **sérialisation d'un statut** d'opération — pierre angulaire pour les agents, les plugins, et les suites de tests des chapitres à venir. `object` rend triviale la **création d'enregistrements** transitoires — qu'il s'agisse de paramètres d'appel, de résultats intermédiaires, ou de valeurs à journaliser.

Ces trois utilitaires complètent donc la machinerie méta du triangle homoiconique en lui apportant les **véhicules de données** dont elle a besoin pour fonctionner en pratique. Sans `attempt`, un agent générique devrait écrire un `try`/`catch` à chaque appel pour transformer une erreur en donnée transmissible ; sans `object`, un plugin devrait construire ses réponses au format JSON brut, sans la commodité de l'accès pointé ; sans `collect`, une boucle de génération produirait du code à coup d'`append` à un accumulateur explicitement nommé.

28.10.4. Vers le chapitre 29

Le chapitre 28 a posé la pierre angulaire : VVL sait manipuler du code comme donnée, et il dispose des helpers ergonomiques pour le faire élégamment. Le chapitre 29 va exploiter cette base pour introduire un mécanisme qui, sans elle, ne pourrait pas exister : les **dialectes**, c'est-à-dire des grammaires VVL embarquées capables de parser des langages domaine-spécifiques. Le moteur de parsing — `[parse]` — accepte une grammaire écrite **comme donnée VVL**, l'évalue contre une entrée, et produit un résultat structuré. C'est, littéralement, l'application directe du triangle homoiconique au problème de l'ingestion de DSL.

Le lecteur qui aura suivi attentivement ce chapitre verra venir cette articulation : tout ce qui sera dit dans le prochain chapitre — règles primitives, quantifieurs, alternatives, captures — n'est en fin de compte que **du code VVL stocké comme donnée et évalué par un interpréteur spécialisé**. Le triangle homoiconique n'est pas une abstraction théorique ; c'est l'outil quotidien qui rend possible la plupart des constructions avancées des chapitres à venir.

29. Dialectes : parsing déclaratif et DSL embarqués

Mots-clés : *dialecte, DSL, parse, grammaire-comme-donnée, REBOL, capture, quantifieur, alternative, dialect:def, dialect:parse, dialect:list, ingestion, validation, langage embarqué, PEG*

Le chapitre précédent a élevé le code VVL au rang de valeur de première classe : `[load]`, `[evaluate]`, `[quote]`, `[dump]` permettent de construire, inspecter, transformer et exécuter des expressions VVL **comme s'il s'agissait de données ordinaires**. Cette propriété — l'homoiconicité — est puissante mais elle reste cantonnée à un usage : *écrire du VVL avec du VVL*. Or, dans la vie d'un programme réel, l'on doit constamment ingérer du contenu qui **n'est pas** du VVL : une chaîne envoyée par une API REST, une ligne de log à analyser, une réponse en langage naturel produite par un LLM, une notation métier qu'un client a fixée par tradition. Tout cela doit être *parsé* avant d'entrer dans le langage.

Le présent chapitre franchit ce dernier pas. Il introduit le builtin `[parse]` et son écosystème, qui permettent d'écrire des **grammaires VVL** capables de consommer une entrée arbitraire et de produire un arbre de captures structuré. Cette famille de builtins est, dans la littérature des langages dynamiques, désignée sous le nom de *dialectes* — terme hérité de REBOL et de son descendant Red, qui ont popularisé l'idée. Un dialecte est, à proprement parler, une **petite grammaire ad-hoc** que l'on déclare dans le code VVL ordinaire, sans avoir à écrire un parseur Python séparé, sans étape de génération de code, sans dépendance externe.

La promesse est forte. Un développeur qui rencontre un format métier — disons, des numéros de série au format `ABC-12345-XY` — peut, en quelques lignes, déclarer une grammaire qui valide et extrait simultanément les trois composantes, **et** la stocker comme dialecte nommé réutilisable. Cinq minutes plus tard, le même développeur peut tisser ce dialecte dans un pipeline d'ingestion qui reçoit une sortie LLM en langage naturel, ne retient que les fragments qui matchent le format attendu, et les injecte dans une base de connaissance ontologique. L'enchaînement *neural → grammaire VVL → ontologie* devient, grâce aux dialectes, un idiome quotidien.

29.1. Du triangle homoiconique aux dialectes

Avant d'examiner la mécanique de `[parse]`, il vaut la peine de positionner correctement ce nouveau builtin par rapport à ce qui existait déjà. VVL disposait, depuis le chapitre 6, du builtin `[match]` pour le **destructurage** de valeurs. Et le chapitre 28 a introduit `[load]` pour le **parsing de source VVL**. Pourquoi un troisième outil ?

29.1.1. match, load, parse : trois rôles distincts

Le tableau ci-dessous fixe la division du travail entre les trois builtins, qui semblent à première vue se recouper mais correspondent en réalité à trois situations très différentes.

Builtin	Entrée	Type de motif	Objectif
[match $v pat body ...]	valeur VVL déjà construite	motif structurel sur une valeur	destructurer et dispatcher
[load "src"]	string contenant du code **VVL**	syntaxe fixe : la grammaire de VVL	parser une source pour obtenir une forme

Builtin	Entrée	Type de motif	Objectif
[parse input grammar]	string OU liste arbitraire	grammaire **utilisateur** déclarée en VVL	consommer une entrée avec une règle ad-hoc

Trois rôles, trois portées. `[match]` est l'outil de la **logique de programme** : on a déjà une valeur, on veut en distinguer la forme et agir selon les cas. `[load]` est l'outil de la **méta-programmation interne** : on a un texte qui *est* du VVL, on veut en faire une forme manipulable. `[parse]`, le nouveau venu, est l'outil de **l'ingestion** : on a une entrée arbitraire et l'on veut y appliquer une grammaire qu'on a écrite soi-même.

La règle pratique qui s'en dégage est limpide. Si l'entrée est déjà une valeur VVL (liste, dictionnaire, scalaire), on utilise `[match]`. Si l'entrée est du **texte VVL** — par exemple le résultat d'une sérialisation `[dump]` —, on utilise `[load]` puis on évalue ou destructure. Si l'entrée est **quelconque** — du texte non-VVL, une liste hétérogène, un message d'API —, on utilise `[parse]` avec la grammaire qu'on aura conçue pour ce format précis.

29.1.2. Grammaire-comme-donnée

Une fois le rôle de `[parse]` clarifié, sa propriété la plus singulière apparaît : la **grammaire est elle-même une donnée VVL**. Une grammaire n'est pas un objet abstrait stocké dans un parseur interne ; ce n'est pas non plus une string que l'on compilerait à part — c'est une **liste VVL** ordinaire, dont les éléments sont des règles. Et parce qu'elle est une liste, elle hérite immédiatement de toute la machinerie disponible sur les listes : on peut la construire dynamiquement, la concaténer, la stocker, la dumper, la passer en argument à une fonction, la transmettre par le réseau.

Cette propriété — *grammaire-comme-donnée* — est la projection directe de l'homoiconicité du chapitre 28 sur le domaine du parsing. Là où un parseur traditionnel (yacc, ANTLR, parser combinator) traite la grammaire comme un artefact à part, compilé séparément, VVL la traite comme **n'importe quelle autre valeur**. Une conséquence agréable : un programme peut composer une grammaire en assemblant plusieurs sous-grammaires, ou en injecter une portion qui dépend d'un paramètre runtime.

```
# La grammaire est une liste VVL : on peut la construire dynamiquement

$prefix = [list [literal "ID-"]]
$body   = [list [some digit]]
$grammar = [concat $prefix $body [list end]]

$r = [parse "ID-12345" $grammar]
[log $r.ok]                       # => 1
```

On en mesure la portée : un système d'agent (chapitre 22) peut, à partir d'une description en langage naturel produite par un LLM, **construire une grammaire VVL ad-hoc**, l'utiliser pour valider les sorties suivantes du même LLM, puis l'archiver pour réutilisation. La boucle *langage naturel → grammaire formelle → validation → archivage* devient écrivable en une vingtaine de lignes. Sans `[parse]`, cette même boucle exigerait soit un parseur Python externe (avec passerelle), soit une cascade d'expressions régulières (fragile, opaque).

29.1.3. La filiation REBOL et la déviation VVL

Les esprits avertis reconnaîtront, dans la signature et le vocabulaire de `[parse]`, le moteur de dialectes de **REBOL** (et son successeur Red) — un langage dont l'une des originalités historiques fut précisément d'élever les grammaires au statut de données. La parenté n'est pas accidentelle : VVL emprunte assumément à REBOL le triplet de mots-clés `some`/`any`/`opt`, l'idée de captures nommées, et la convention du `end` final.

Mais VVL n'est pas REBOL, et les divergences sont également instructives. Là où REBOL utilise un caractère `|` pour les alternatives, VVL utilise `[alt ...]` — parce que le `|` a déjà une signification dans VVL (le parallélisme évoqué au chapitre 21) et qu'une collision serait fatale. Là où REBOL accepte une grande liberté de notation, VVL fixe une convention plus stricte : les règles composites sont **explicitement crochetées** (`[some letter]` et non `some letter`) pour qu'aucune ambiguïté syntaxique ne puisse se glisser. Là où REBOL traite indifféremment les entrées texte et binaires, VVL distingue clairement entrées **string** et entrées **liste**, parce que la sémantique des règles primitives n'est pas la même dans les deux cas. Ces déviations sont volontaires : elles maintiennent la cohérence du langage hôte tout en préservant l'ergonomie REBOL.

29.2. Anatomie d'un [parse]

Avec les positions de doctrine fixées, on peut entrer dans la mécanique. `[parse]` est un builtin **raw** (sans pré-évaluation des arguments) qui accepte deux paramètres : une **entrée** — string ou liste — et une **grammaire** — liste VVL. Il retourne toujours un objet `VVL_Json` dont la structure est fixe.

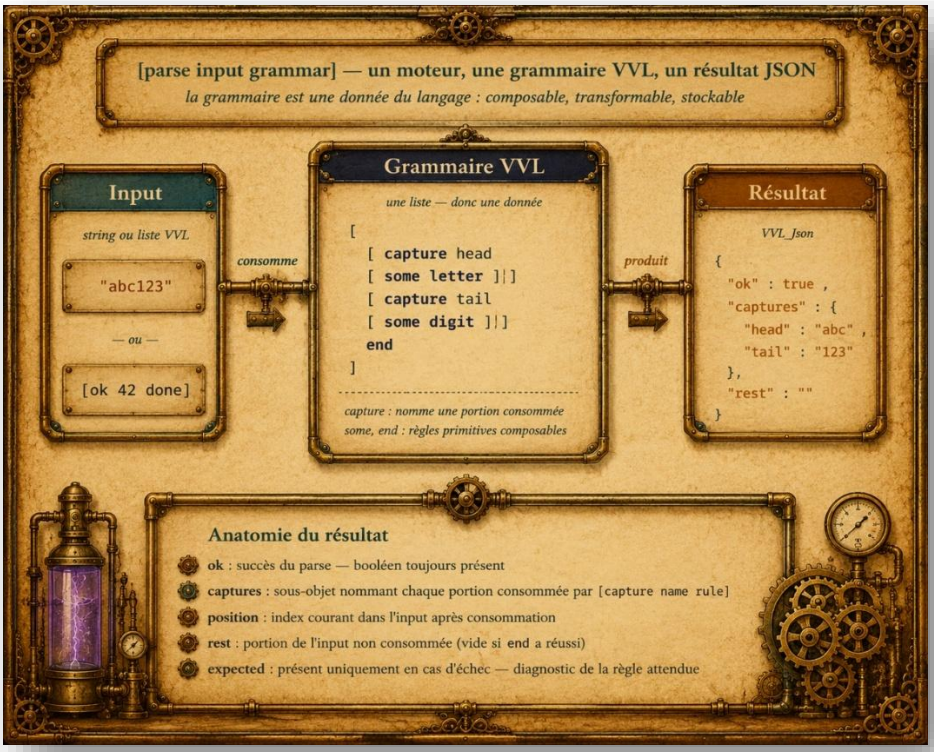


Figure 29.1 — *Le flux de [parse]. L'entrée est consommée par la grammaire, qui produit un résultat structuré. Les cinq champs du résultat (ok, captures, position, rest, expected) constituent l'API stable du builtin.*

29.2.1. La signature

La signature canonique est `[parse input grammar]`. Le runtime évalue `input` pour obtenir la valeur à consommer ; en revanche, il traite `grammar` **comme une donnée brute**, sans en évaluer les sous-éléments. Cette asymétrie est essentielle : si `grammar` était évalué comme une expression ordinaire, le runtime tenterait d'exécuter `[some letter]` comme s'il s'agissait d'un appel à un builtin `some` qui n'existe pas. Le caractère raw préserve la grammaire **dans son état de structure**, prête à être interprétée par le moteur de parsing interne.

Le résultat est, dans tous les cas, un objet JSON avec les champs suivants :

Champ	Type	Sémantique
ok	booléen	le parse a-t-il réussi entièrement ?
position	entier	index courant dans l'input après consommation
captures	objet JSON	sous-objet associant noms et portions consommées
rest	string ou liste	portion de l'input non consommée
expected	string (optionnel)	présent en cas d'échec, décrit la règle attendue

Le couple `ok` / `expected` constitue un protocole de **diagnostic** : un parse qui échoue ne lève pas d'exception, mais retourne un objet avec `ok: false` et un champ `expected` qui donne au développeur une information minimale sur le point d'échec. Cette uniformité — *toujours* un JSON, *jamais* une exception — fait de `[parse]` un compagnon naturel d'`[attempt]` (chapitre 28) : un programme qui consomme des entrées hétérogènes peut traiter succès et échec de manière strictement homogène, sans cas spécial pour la gestion d'erreur.

29.2.2. Un premier exemple complet

L'exemple canonique du tutoriel VVL — qui sert de référence dans la documentation du moteur — combine deux captures et un sentinelle `end`. Il mérite d'être déroulé en détail parce qu'il illustre quatre éléments essentiels en quelques lignes.

```
001  # Parser un identifiant de la forme : lettres suivies de chiffres
002
003  $r = [parse "abc123"
004      [[capture head [some letter]]
005       [capture tail [some digit]]
006       end]
007  ]
008
009  [log $r.ok]                    # => 1
010  [log $r.captures.head]         # => abc
011  [log $r.captures.tail]         # => 123
012  [log $r.rest]                  # => (vide)
```

Quatre éléments s'enchaînent. **Premièrement**, l'entrée `"abc123"` est une string ; le moteur va donc consommer caractère par caractère. **Deuxièmement**, la grammaire est une liste de trois règles successives, séparées simplement par leur position dans la liste — il n'y a pas d'opérateur de séquence

explicite, parce que la séquence est l'arrangement par défaut. **Troisièmement**, `[capture head [some letter]]` nomme `head` la portion consommée par `[some letter]` (une ou plusieurs lettres). **Quatrièmement**, `end` impose que la consommation atteigne la fin de l'entrée — sans cette règle, l'input `"abc123extra"` aurait également retourné `ok: true` (la grammaire aurait consommé `abc123` et laissé `extra` dans `rest`).

Le sentinelle `end` est l'un des outils les plus utiles du moteur, parce qu'il transforme une grammaire **permissive** en une grammaire **stricte**. La discipline qui s'installe rapidement chez les développeurs VVL est de **toujours** terminer une grammaire par `end`, sauf raison explicite de tolérer du contenu résiduel. La raison la plus courante d'omettre `end` est précisément l'inverse : utiliser `[parse]` comme un *consommateur de préfixe*, qui n'examine que le début de l'entrée et laisse intentionnellement le reste.

29.2.3. String input vs list input

VVL distingue strictement, dans le moteur de parsing, les entrées **string** des entrées **liste**. Cette distinction est l'une des décisions de design les plus importantes du chapitre, parce qu'elle conditionne la sémantique de presque toutes les règles primitives.

Sur une entrée string, le parseur **consomme caractère par caractère**. La règle `digit` consomme un seul caractère qui doit être dans `0123456789`. La règle `letter` consomme un seul caractère alphabétique. La règle `space` consomme un seul caractère d'espace. Et un littéral string `"abc"` consomme exactement les caractères `a`, `b`, `c` dans cet ordre.

Sur une entrée liste, le parseur **consomme élément par élément**. La règle `digit` (ou `num`) consomme un seul élément qui doit être un nombre VVL. La règle `letter` consomme un seul élément qui doit être une chaîne alphabétique. La règle `list` consomme un seul élément qui doit être une liste VVL. Et un littéral nombre dans la grammaire, par exemple `42`, consomme un seul élément égal à 42.

```
# String input : consomme caractere par caractere
$r1 = [parse "abc" ["a" "b" "c" end]]
[log $r1.ok]                    # => 1

# List input : consomme element par element
$r2 = [parse `[42] [42 end]]
[log $r2.ok]                    # => 1

$r3 = [parse `[ok 42] [[literal ok] [capture v any] end]]
[log $r3.captures.v]            # => 42
```

Cette double sémantique permet au même moteur de servir deux usages très différents. Pour consommer une chaîne brute provenant d'un fichier ou d'une API, on utilise l'entrée string. Pour consommer une structure VVL déjà construite — par exemple le résultat d'un `[load]` qui aurait produit une liste de tokens —, on utilise l'entrée liste. La distinction n'est jamais ambiguë : le runtime examine le type de `input` au moment de l'appel et applique la sémantique correspondante.

Il existe une subtilité notable : `[literal value]`. Quand la grammaire contient un atome qui pourrait être interprété **soit** comme un nom de règle primitive **soit** comme un littéral à matcher dans la liste, le builtin `[literal value]` force l'interprétation littérale. Par exemple, sans `[literal]`, la grammaire

`[ok 42 end]` interpréterait `ok` comme un mot et tenterait de le matcher selon la règle implicite ; avec `[[literal ok] 42 end]`, le moteur sait que `ok` est un atome littéral à comparer élément par élément. Cette précaution est rarement nécessaire en pratique — les noms de règles primitives sont fixés et peu nombreux — mais elle est essentielle dans les cas-limites.

29.3. Les règles primitives

Une grammaire de dialecte se construit à partir d'un petit vocabulaire de **règles primitives**, qui sont les briques de plus bas niveau du moteur. Le tableau ci-dessous donne la liste complète, avec la sémantique de chacune sur les deux types d'entrée.

Règle	Sur entrée string	Sur entrée liste
digit, number, num	un caractère dans 0–9	un élément VVL_Num
letter	un caractère alphabétique	un élément string alphabétique
space	un caractère d'espace	un élément string d'espace
string	un caractère quelconque	un élément VVL_String
word	un caractère alphanumérique	un élément VVL_Word
list	— inapplicable —	un élément VVL_List
any	un caractère quelconque	un élément quelconque
end	fin de la string	fin de la liste
littéral string "abc"	la séquence a, b, c	un élément égal à "abc"
littéral nombre 42	— inapplicable —	un élément égal à 42
[literal value]	— forcé en littéral —	— forcé en littéral —

Trois remarques s'imposent sur cette liste. **Premièrement**, les noms `digit`, `number` et `num` sont des synonymes — choisis pour suivre les conventions de lecture en français (`digit` chiffre) ou en anglais d'inspiration mathématique (`number`, `num`). Aucune préférence n'est imposée. **Deuxièmement**, certaines règles n'ont pas d'analogue raisonnable sur l'une des deux entrées : `list` ne peut pas exister sur une string (une string ne contient pas d'éléments-liste), et un littéral nombre comme `42` n'a pas de sens littéral sur une string (on l'écrirait `"42"` pour matcher les deux caractères). Le moteur signale clairement ces cas par un échec de parse plutôt que par une exception. **Troisièmement**, `any` est universel — il consomme exactement un élément (caractère ou élément de liste), quel qu'il soit. C'est l'outil de choix quand on veut juste consommer un emplacement sans contraindre son contenu.

29.3.1. Un premier exemple par règle

Pour fixer définitivement la sémantique, déroulons chaque règle sur un exemple minimal.

```
001  # digit : un chiffre
002  [parse "5" [digit end]]            # => ok
003  [parse "abc" [digit end]]          # => echec
004
005  # letter : une lettre
006  [parse "X" [letter end]]           # => ok
007
008  # space : un blanc
```

```
[parse " " [space end]]            # => ok

# any : un caractere quelconque
[parse "?" [any end]]              # => ok

# end : fin de l'entree
[parse "" [end]]                   # => ok
[parse "x" [end]]                  # => echec

# Litteral string : sequence exacte
[parse "hello" ["hello" end]]      # => ok
[parse "hi"    ["hello" end]]      # => echec

# Litteral nombre sur liste : element exact
[parse `[42] [42 end]]             # => ok
[parse `[7]  [42 end]]             # => echec
```

29.3.2. Le rôle pivot de [literal]

L'opérateur `[literal value]` mérite une attention particulière. Sa raison d'être est de **désambiguer** les cas où un atome de la grammaire pourrait être confondu avec un nom de règle primitive. Imaginons une grammaire qui veut consommer une liste commençant par le mot symbolique `ok` suivi d'un nombre :

```
# Sans [literal], le moteur pourrait essayer d'interpreter "ok"
# comme un nom de regle inconnu et echouer obscurement.

$r = [parse `[ok 42]
    [[literal ok] [capture value any] end]
]
[log $r.captures.value]           # => 42
```

Sur l'exemple ci-dessus, `[literal ok]` indique sans ambiguïté que le moteur doit consommer un élément de la liste **égal à** la valeur symbolique `ok`. Sans cet enrobage, le risque serait que le moteur tente d'interpréter `ok` comme un nom de règle (et émette un `expected` confus) ou qu'il consomme correctement mais pour des raisons qu'on ne maîtrise pas. La règle de bon goût est d'utiliser `[literal]` dès que l'on **matche un mot symbolique** dans une grammaire sur liste — c'est un coût d'écriture mineur pour une garantie de clarté.

29.4. Les quantifieurs

Les règles primitives ne consomment qu'un seul élément. Pour consommer plusieurs occurrences successives, le moteur fournit cinq **quantifieurs** dont la syntaxe et la sémantique reprennent fidèlement la tradition REBOL/Red. Le tableau ci-dessous en donne la liste.

Quantifieur	Sémantique
[some rule]	consomme une ou plusieurs occurrences de rule
[any rule]	consomme zéro ou plusieurs occurrences de rule
[opt rule]	consomme zéro ou une occurrence de rule
[repeat n rule]	consomme exactement n occurrences de rule
[repeat min max rule]	consomme entre min et max occurrences de rule

Une mise en garde s'impose sur la collision lexicale apparente : `any` est à la fois un **quantifieur** (zéro ou plus occurrences) et une **règle primitive** (un élément quelconque). Le moteur distingue les deux par le contexte syntaxique : seul `[any rule]` est interprété comme un quantifieur ; le nu `any` est la règle primitive. La même observation vaut pour `some` qui n'est *que* quantifieur, et pour `opt` qui n'est *que* quantifieur — la liste des règles primitives n'inclut pas ces deux mots.

29.4.1. some : un ou plus

Le quantifieur `some` consomme **avidement** autant d'occurrences successives de sa règle qu'il en trouve, avec un minimum de un. S'il n'y a pas d'occurrence du tout, le parse échoue.

```
001  # some : un ou plus
002
003  [parse "12345" [[some digit] end]]  # => ok
004  [parse "1"     [[some digit] end]]  # => ok
005  [parse ""      [[some digit] end]]  # => echec : zero chiffre
006  [parse "abc"   [[some digit] end]]  # => echec : pas un chiffre
```

L'avidité de `some` est importante en pratique : si l'on écrit `[[some digit] "x" end]` sur l'entrée `"123x"`, le moteur consomme `123` avec `some digit`, puis matche le `x` final, et atteint la fin. Si la consommation avait été paresseuse (n caractères, n choisi minimal), `some digit` se serait arrêté à `1` et `"x"` n'aurait pas matché `23x`. Le moteur de VVL est, comme celui de REBOL, **systématiquement avide**.

29.4.2. any : zéro ou plus

Le quantifieur `any` est la version tolérante de `some` : il accepte zéro occurrence et ne fait jamais échouer le parse pour absence.

```
001  # any : zero ou plus
002
003  [parse "12345" [[any digit] end]]   # => ok
004  [parse ""      [[any digit] end]]   # => ok ! zero occurrence accepte
005  [parse "abc"   [[any digit] end]]   # => echec : "a" non consomme avant end
```

L'exemple `"abc"` est instructif : le parse échoue non pas parce que `any digit` aurait échoué, mais parce que **après** avoir consommé zéro chiffre, il reste `abc` à consommer et la règle `end` exige la fin. Le diagnostic `expected: end` du moteur reflète cela précisément. C'est un piège classique pour les

débutants, qui s'attendent à ce que `any` soit **non-bloquant** : il l'est, mais la grammaire dans son ensemble peut tout de même échouer si une règle suivante exige des caractères inexistants.

29.4.3. opt : zéro ou un

`opt` est le quantifieur de l'**optionnel** : il consomme une occurrence si elle est présente, et passe son tour autrement, sans jamais faire échouer.

```
001  # opt : modelise un signe optionnel
002
003  [parse "+42" [[opt "+"] [some digit] end]]
004  # => ok : "+" est consomme, puis "42"
005
006  [parse "42"  [[opt "+"] [some digit] end]]
007  # => ok : pas de "+", on passe directement aux chiffres
```

`opt` est l'outil idiomatique pour modéliser les variations syntaxiques tolérées : un signe optionnel, un espace optionnel, un préfixe optionnel. Couplé à `alt` (section suivante), il permet de décrire des grammaires riches sans verbosité excessive.

29.4.4. repeat : compte exact ou borné

`repeat` accepte deux signatures. La forme simple `[repeat n rule]` consomme exactement `n` occurrences. La forme bornée `[repeat min max rule]` consomme entre `min` et `max` occurrences, avec le même comportement avide que `some`.

```
001  # repeat n : compte exact
002
003  [parse "123" [[repeat 3 digit] end]]    # => ok
004  [parse "12"  [[repeat 3 digit] end]]    # => echec : 2 chiffres seulement
005  [parse "1234"[[repeat 3 digit] end]]    # => echec : 4 chiffres consommes
006
007  # repeat min max : compte borne
008
009  [parse "12"   [[repeat 1 3 digit] end]] # => ok
010  [parse "1234" [[repeat 1 3 digit] end]] # => echec : 4 > 3
```

L'usage typique de `[repeat n rule]` est la validation de formats à longueur fixe : un identifiant à 5 chiffres, un code-barres à 13 chiffres, une signature de 64 caractères hexadécimaux. La forme bornée `[repeat min max rule]` couvre les formats à longueur variable mais contrainte : un mot de passe entre 8 et 32 caractères, un identifiant entre 3 et 12 lettres. Ensemble, les deux formes de `repeat` couvrent la grande majorité des cas où la simple alternance `some`/`any`/`opt` ne suffit pas.

29.5. Les alternatives : [alt ...]

Les règles primitives consomment un atome, les quantifieurs en consomment plusieurs ; il reste à exprimer le **choix** entre plusieurs alternatives. Le moteur fournit pour cela le builtin `[alt rule1 rule2 ...]`, qui tente chacune des règles fournies dans l'ordre et retient la première qui réussit. Si aucune ne réussit, l'alternative entière échoue et le parse reprend au point précédent.

```
# alt : essayer plusieurs branches

[parse "a"  [[alt "a" "b" "c"] end]]   # => ok
[parse "b"  [[alt "a" "b" "c"] end]]   # => ok
[parse "d"  [[alt "a" "b" "c"] end]]   # => echec

# Combinaison avec opt et autres regles

[parse "abz" [
    [alt "a" "x"]
    [opt "z"]
    "bz"
    end
]]
# Aucune correspondance possible : "abz" ne commence pas par ce schema
```

29.5.1. Pourquoi pas le pipe |

Une question vient immédiatement à l'esprit du lecteur familier d'autres systèmes de parsing : pourquoi VVL n'a-t-il pas adopté le caractère `|` comme séparateur d'alternatives, comme le font REBOL, ANTLR, ou les expressions régulières POSIX ? La réponse est une **collision lexicale** : le `|` est, en VVL, l'opérateur de **parallélisme** introduit au chapitre 21 (`task | task`). Réutiliser ce caractère pour les alternatives dans les dialectes aurait introduit une ambiguïté contextuelle pénible — `[some letter | some digit]` aurait-il signifié *lettre OU chiffre* (dialecte) ou *lettre EN PARALLÈLE chiffre* (tâches) ?

Le choix de `[alt ...]` évite cette ambiguïté en transférant l'opérateur dans le namespace des règles, sans interférer avec la syntaxe globale du langage. C'est un peu plus verbeux à l'écriture, mais c'est plus clair à la lecture et, surtout, c'est **localement explicite** : un lecteur du code voit immédiatement qu'il a affaire à une alternative de grammaire, pas à un construct de contrôle de flow ordinaire.

29.5.2. Ordre des alternatives et avidité

Une subtilité importante : `[alt]` essaie ses branches **dans l'ordre fourni** et retient **la première qui réussit**, sans backtracking sur la grammaire globale. Cela signifie que l'ordre compte. Si l'on écrit `[alt "a" "ab"]` sur l'entrée `"ab"`, le moteur consomme `"a"` et arrête là — la branche `"ab"` n'est jamais essayée, alors qu'elle aurait consommé davantage. Le mauvais ordre des branches peut faire échouer un parse qui aurait dû réussir.

```
# Mauvais ordre : "a" matche avant "ab"
$r1 = [parse "ab" [[alt "a" "ab"] end]]
[log $r1.ok]                          # => echec
# Apres "a", il reste "b" qui ne matche pas end

```

```
# Bon ordre : la branche la plus longue d'abord
$r2 = [parse "ab" [[alt "ab" "a"] end]]
[log $r2.ok]                          # => ok
```

La règle de bon goût qui s'en dégage est : **toujours placer les branches les plus longues en premier** dans `[alt]`. Cette règle, connue dans la littérature des PEG (*Parsing Expression Grammars*) sous le nom de *longest-match-first*, évite les surprises et rend la grammaire prévisible. Pour les dialectes complexes, on peut même écrire les branches par longueur décroissante et commenter cette intention.

29.6. Captures : nommer les portions consommées

Une grammaire qui se contente de valider une entrée — *cette chaîne ressemble-t-elle à un identifiant ?* — n'est qu'à moitié utile. Le moteur devient pleinement opératoire lorsqu'il **extrait** les portions intéressantes pour les remettre au programme appelant. C'est le rôle des **captures**.

29.6.1. La syntaxe : [capture name rule]

`[capture name rule]` enveloppe une règle quelconque et indique au moteur que la portion consommée par cette règle doit être enregistrée sous le nom `name`. À la fin du parse, le sous-objet `captures` du résultat contient toutes les portions ainsi nommées.

```
# Capture simple : extraire prenom et age d'une chaine "Jean 42"

$r = [parse "Jean 42"
    [[capture name [some letter]]
     space
     [capture age [some digit]]
     end]
]

[log $r.captures.name]            # => Jean
[log $r.captures.age]             # => 42
```

Le builtin `capture` n'altère pas la sémantique de matching de la règle qu'il enveloppe : `[capture name [some letter]]` consomme exactement la même portion que `[some letter]` seul. Sa seule action est d'**enregistrer** cette portion sous le nom indiqué. C'est, à proprement parler, une annotation orthogonale au parsing.

29.6.2. Le type de la capture : string vs liste

Une question naturelle se pose : sous quelle forme est stockée la portion capturée ? La réponse dépend du type de l'**entrée**. Sur une entrée string, une capture est toujours la **sous-chaîne** consommée (un `VVL_String`). Sur une entrée liste, la règle est plus subtile : si la capture porte sur un seul élément consommé, le résultat est ce **scalaire** unique ; si elle porte sur plusieurs éléments, le résultat est une **liste** des valeurs consommées.

```
# Sur string : capture = sous-chaine

$r1 = [parse "abc123" [[capture s [some letter]] end-not-required ...]]
# capture.s : "abc"  (un VVL_String)

# Sur liste avec un seul element capture : scalaire

$r2 = [parse `[42] [[capture v any] end]]
# capture.v : 42  (un VVL_Num, pas une liste)

# Sur liste avec plusieurs elements captures : liste

$r3 = [parse `[a b c]
    [[capture xs [some letter]] end]
]
# capture.xs : ["a" "b" "c"]  (une VVL_List)
```

Cette règle d'**aplatissement** dans le cas du singleton est une concession à l'ergonomie : la majorité des cas pratiques visent à extraire une **valeur unique**, et avoir à toujours déballer une liste à un élément serait pénible. La règle est intuitive : *une seule chose → la chose ; plusieurs → la liste*.

29.6.3. Captures multiples et imbriquées

Une grammaire peut contenir autant de captures qu'on le souhaite, et celles-ci peuvent être imbriquées dans des règles composées. Le sous-objet `captures` final contient simplement *toutes* les captures rencontrées au cours du parse, indexées par leur nom.

```
# Captures multiples : parser un appel de fonction simplifie

$r = [parse "add(3,7)"
    [[capture fn   [some letter]]
     "("
     [capture a    [some digit]]
     ","
     [capture b    [some digit]]
     ")"
     end]
]

[log $r.captures.fn]              # => add
[log $r.captures.a]               # => 3
[log $r.captures.b]               # => 7
```

Si deux captures portent le **même nom**, la seconde écrase silencieusement la première. C'est rarement souhaité — la pratique recommandée est de donner à chaque capture un nom unique au sein de la grammaire. Si un comportement d'accumulation est nécessaire (par exemple capturer un nombre

variable d'éléments), le pattern idiomatique est de capturer la liste entière en une fois avec un quantifieur, plutôt que d'écrire plusieurs captures du même nom :

```
# Capturer une liste de mots separes par des espaces

$r = [parse "alpha beta gamma delta"
    [[capture mots [some [alt [some letter] space]]]
     end]
]

# capture.mots : la sous-chaine consommee entiere
# (la decomposition en liste passe par un parse ulterieur)
```

Pour une **décomposition profonde** — par exemple une liste de mots déjà séparés —, le pattern recommandé est en **deux passes** : un premier `[parse]` extrait la portion globale, un `[split]` ou un second `[parse]` décompose. Le moteur de dialectes de VVL est délibérément **flat-captures-only** dans ce premier MVP : il n'y a pas de captures qui produiraient nativement une structure d'arbre. Cette limitation sera abordée dans la section 29.10 sur les perspectives.

29.7. Dialectes nommés : un registre d'usage

Jusqu'ici, chaque appel à `[parse]` portait sa grammaire en argument. Pour des usages ponctuels, c'est satisfaisant. Mais dès qu'une même grammaire est appelée à plusieurs endroits — par exemple un format d'identifiant client réutilisé dans dix scripts différents —, l'inlining devient un fardeau de maintenance. Le moteur fournit pour ce cas un trio de builtins qui forment un **registre de dialectes nommés** : `[dialect:def]`, `[dialect:parse]`, `[dialect:list]`.

29.7.1. Déclarer un dialecte : [dialect:def]

`[dialect:def name grammar]` enregistre une grammaire sous un nom dans le **contexte courant**. Le builtin est **raw** : le nom et la grammaire sont préservés tels quels, et la grammaire n'est jamais évaluée comme une expression ordinaire.

```
# Declarer un dialecte alphaNum

[dialect:def alphaNum
    [[capture head [some letter]]
     [capture tail [some digit]]
     end]
]
# Le nom alphaNum est maintenant enregistre dans le contexte.
```

Le nom du dialecte peut être un **mot** (comme `alphaNum` ci-dessus) ou une **chaîne** (comme `"alphaNum"`). Les deux formes coexistent pour s'adapter aux styles d'écriture : les mots sont plus naturels quand le

nom est connu statiquement, les chaînes sont indispensables quand le nom est calculé dynamiquement.

Les dialectes enregistrés vivent dans une structure dédiée du contexte (`ctx.dialects`). **Ils ne deviennent pas des builtins**, ils ne masquent pas les commandes existantes, et ils n'apparaissent pas dans la table des keywords. Cette séparation est délibérée : un dialecte est une **donnée**, pas une commande. La conséquence pratique est qu'un développeur peut nommer un dialecte `log`, `log`, ou `parse` sans aucune collision avec le builtin éponyme.

29.7.2. Utiliser un dialecte : [dialect:parse]

`[dialect:parse name input]` est la version registrée de `[parse]`. Elle accepte un nom (mot ou string) et une entrée, et applique la grammaire enregistrée sous ce nom. Le résultat est strictement identique à celui d'un appel direct à `[parse]` avec la grammaire correspondante.

```
# Utiliser le dialecte declare plus haut

$r = [dialect:parse alphaNum "abc123"]

[log $r.ok]                      # => 1
[log $r.captures.head]           # => abc
[log $r.captures.tail]           # => 123
```

Si le nom n'est pas enregistré, `[dialect:parse]` ne lève pas d'exception : il retourne, comme `[parse]`, un objet JSON avec `ok: false` et un champ `expected` qui signale le diagnostic, en l'occurrence `expected: "dialect:name"`. Cette uniformité permet à un code consommateur de traiter de manière homogène les échecs de grammaire et les noms inconnus.

```
# Dialecte inconnu : echec gracieux

$r = [dialect:parse inexistant "abc"]
[log $r.ok]                      # => 0
[log $r.expected]                # => dialect:name
```

29.7.3. Inspecter le registre : [dialect:list]

`[dialect:list]` retourne la liste **triée** des noms de dialectes enregistrés dans le contexte courant. C'est l'outil d'introspection du registre, utile pour les commandes de documentation automatique ou pour les vérifications préalables à un appel.

```
# Inspecter le registre

[dialect:def alphaNum  [[some letter] [some digit] end]]
[dialect:def numeric   [[some digit] end]]
[dialect:def email     [...grammaire d'email...]]

[log [dialect:list]]
# => ["alphaNum" "email" "numeric"]
```

Le tri alphabétique est imposé par convention : il rend la sortie déterministe pour les tests d'intégration et pour les comparaisons inter-sessions. Si l'utilisateur a besoin d'un ordre d'insertion, il doit le maintenir lui-même dans une variable séparée.

29.7.4. Le pattern de bibliothèque de dialectes

Un usage idiomatique des dialectes nommés est de constituer une **bibliothèque** de grammaires métier dans un fichier dédié, chargé une fois par `[import]` au démarrage d'une session. Chaque grammaire devient un nom court et autodocumenté.

```
# Fichier : lib/dialects.vvl

[dialect:def serialNumber
    [[repeat 3 letter]
     "-"
     [repeat 5 digit]
     "-"
     [repeat 2 letter]
     end]
]

[dialect:def isoDate
    [[repeat 4 digit] "-"
     [repeat 2 digit] "-"
     [repeat 2 digit] end]
]

[dialect:def semver
    [[capture major [some digit]] "."
     [capture minor [some digit]] "."
     [capture patch [some digit]] end]
]
# Fichier : script.vvl

[import "lib/dialects.vvl"]

[log [dialect:parse semver "1.2.3"]]
# => {"ok":1, "captures":{"major":"1","minor":"2","patch":"3"}}
```

Cette pratique transforme `[dialect:def]` en mécanisme léger de **modularisation des grammaires** — anticipant, sans rivaliser avec lui, le système de modules complet qui sera introduit au chapitre suivant. Les dialectes restent des **données** ; les modules du chapitre 30 introduiront un mécanisme symétrique pour les **commandes**.

29.8. Cas d'usage

La présentation théorique étant complète, on déroule trois cas d'usage qui illustrent les contextes dans lesquels les dialectes apportent une réelle valeur. Aucun n'est artificiel : tous trois sont issus de la pratique des chapitres précédents.

29.8.1. Validation d'une sortie LLM

Au chapitre 17, l'induction LLM zero-shot fait produire à un modèle de langage des règles VVL synthétisées en texte. La promesse est belle, mais en pratique le LLM produit régulièrement du texte qui *ressemble* à une règle valide sans en être tout à fait une — un guillemet manquant, un mot mal écrit, un crochet ouvert non fermé. Sans validation préalable, ces sorties seraient injectées telles quelles dans le contexte VVL et y produiraient des erreurs runtime difficiles à diagnostiquer.

Un dialecte de validation devient l'outil idéal pour intercepter ces dérives. L'exemple ci-dessous esquisse une grammaire minimaliste pour les règles `ontology:rule "name" when ... then ...` :

```
# Dialecte de validation d'une regle ontologique

[dialect:def ontoRuleShape
    ["[ontology:rule "
     [capture name [some [alt letter digit "_"]]]
     " when "
     [capture body [some any]]
     end]
]

# Dans la boucle d'induction
$llm_output = [llm:complete "Genere une regle pour ..."]
$validation = [dialect:parse ontoRuleShape $llm_output]

[when (! $validation.ok)
    `[do [log "regle malformee" $llm_output]
         [continue]]
]

# La regle est bien formee, on peut l'evaluer en toute securite
[evaluate [load $llm_output]]
```

Le coût d'écriture du dialecte est faible — quelques lignes — et la garantie obtenue est forte : aucune sortie LLM mal-formée n'entre dans le runtime. Cette stratégie est précisément la **frontière formelle** que les chapitres sur les agents (22) et l'expert (23) appelaient sans encore avoir l'outil pour la déclarer.

29.8.2. Décomposition d'une notation métier

Le deuxième cas d'usage est plus prosaïque mais infiniment plus courant. Imaginons un système qui reçoit des numéros de série au format `ABC-12345-XY` et qui doit en extraire les trois composantes pour les indexer en base de données. Un dialecte rend cette tâche triviale.

```
# Decomposition d'un numero de serie

[dialect:def serial
    [[capture vendor  [repeat 3 letter]]
     "-"
     [capture id      [repeat 5 digit]]
     "-"
     [capture variant [repeat 2 letter]]
     end]
]

$r = [dialect:parse serial "ABC-12345-XY"]

[when $r.ok
    `[db:insert "products" [object
        [vendor  $r.captures.vendor]
        [id      $r.captures.id]
        [variant $r.captures.variant]
    ]]
]
```

L'alternative — sans dialecte — serait soit une cascade d'expressions régulières (fragile et opaque), soit une fonction Python séparée (passerelle coûteuse). Le dialecte vit dans le code VVL ordinaire, est testable comme tout autre fragment, et porte son intention dans son nom.

29.8.3. Mini-DSL embarqué

Le troisième cas d'usage est le plus avancé : utiliser un dialecte pour implémenter un **mini-langage** dédié à un domaine. Imaginons que l'on veuille permettre aux utilisateurs d'écrire des **filtres** sur des données ontologiques dans une syntaxe plus naturelle que la pleine syntaxe VVL.

```
# Mini-DSL pour les filtres : "field op value"
#   ex : "age > 18", "name == \"Alice\""

[dialect:def filterExpr
    [[capture field [some letter]]
     [any space]
     [capture op [alt ">" "<" "==" "!="]]
     [any space]
     [capture value [some any]]
     end]
]

```

```
# Le programme utilise le dialecte pour interpreter la chaine de l'utilisateur
$userExpr = "age > 18"
$parsed = [dialect:parse filterExpr $userExpr]

$pred = `[do
    $left = (@1.[evaluate $parsed.captures.field])
    [match $parsed.captures.op
        ">"  `($left > $parsed.captures.value)
        "<"  `($left < $parsed.captures.value)
        "==" `($left == $parsed.captures.value)
    ]
]
```

Cet exemple — volontairement esquissé plutôt que complet — montre le pattern profond. Le dialecte décompose la chaîne utilisateur ; le code VVL ordinaire interprète sémantiquement les captures pour construire dynamiquement un prédicat. C'est, à proprement parler, un **mini-interpréteur** d'un langage spécifique embarqué dans VVL. Et il tient en vingt lignes.

29.9. parse vs match : la matrice de décision

Le chapitre 28 avait introduit une matrice de décision pour le triangle homoiconique. Le chapitre 29 mérite la sienne, parce que la confusion la plus fréquente concerne le départ entre `[match]` et `[parse]`. Le tableau ci-dessous achève la clarification commencée en 29.1.

Critère	`[match value pattern body ...]`	`[parse input grammar]`
Entrée	valeur VVL déjà construite (liste, scalaire, JSON)	string ou liste à consommer
Motif	motif structurel sur une valeur	grammaire séquentielle de règles
Quantification	pas de répétition (motif fini)	some, any, opt, repeat (répétition native)
Action	corps de code exécuté par branche	capture nommée → résultat JSON
Échec	retour de la branche par défaut	objet {ok:false, expected:...}
Usage typique	dispatcher sur la forme d'une valeur	consommer une entrée brute avec une grammaire

L'opposition se résume en une phrase : `[match]` est un **switch typé sur la forme**, `[parse]` est un **moteur de grammaire**. La métaphore pourrait être : `[match]` regarde une donnée et demande *de quel type es-tu ?* ; `[parse]` regarde un flux et demande *suis-tu telle structure ?*

Pour les cas pratiques limites — quand on hésite entre les deux —, la règle pragmatique est : si l'entrée est **déjà parsée** (par exemple, déjà une liste VVL aux types attendus), on utilise `[match]` parce qu'on n'a pas besoin de la machinerie du moteur. Si l'entrée est **brute** (string, liste de tokens hétérogène), on utilise `[parse]` parce qu'il faut la consommer dans l'ordre selon une grammaire. La frontière est nette en pratique, et les deux outils ne se concurrencent presque jamais sur le même problème.

29.10. Limites du MVP et perspectives

Le moteur de dialectes présenté ici est un **MVP** — c'est-à-dire un noyau délibérément restreint, qui couvre la majorité des cas pratiques sans prétendre rivaliser avec les moteurs de parsing les plus expressifs du marché (ANTLR, Lark, Earley). Cette section liste les limitations connues et les directions d'évolution. Le lecteur qui rencontrerait l'une de ces limites en pratique peut soit contourner (les techniques sont indiquées), soit guetter une version ultérieure du moteur.

29.10.1. Pas de captures arborescentes nativement

Comme indiqué en 29.6, les captures sont **plates** : le sous-objet `captures` final est un dictionnaire de noms vers valeurs scalaires ou listes plates, sans structure d'arbre. Si la grammaire reconnaît, par exemple, une expression arithmétique imbriquée, le moteur ne produit pas nativement un arbre d'AST — il produit seulement la chaîne consommée.

Le **contournement** habituel est la **stratégie multi-passes** : un premier `[parse]` extrait la portion globale ou les sous-expressions principales, et chaque sous-expression est ensuite reparsée par un appel récursif explicite. Cette stratégie est moins élégante qu'un parser combinator complet mais elle suffit en pratique pour les DSL de complexité moyenne. Pour les grammaires véritablement récursives — par exemple parser une expression mathématique avec parenthèses imbriquées arbitrairement — , le recours à un plugin Python externe (chapitre 33) reste la solution recommandée.

29.10.2. Pas de backtracking sur les alternatives

Comme indiqué en 29.5, `[alt]` essaie ses branches dans l'ordre et retient la première qui réussit, sans **backtracking** : si la branche choisie réussit localement mais conduit à un échec global plus tard, le moteur ne revient pas en arrière pour essayer une autre branche. Ce comportement, hérité des PEG, garantit une complexité linéaire en temps mais oblige l'utilisateur à **ordonner soigneusement** les alternatives.

Le contournement consiste à découper les ambiguïtés en deux passes ou à enrichir la grammaire pour réduire l'espace de choix. Dans les cas les plus complexes (grammaires véritablement ambiguës), il faudra à terme un moteur Earley — chantier ouvert dans la roadmap mais non disponible dans le MVP.

29.10.3. Pas de | pour les alternatives

Le caractère `|` ne sera pas utilisé pour les alternatives, comme expliqué en 29.5.1 : la collision avec le parallélisme du chapitre 21 est définitive. Si une future syntaxe d'alternative plus légère est introduite, elle utilisera un autre caractère ou un mot dédié. La forme `[alt ...]` est, en attendant, la seule canonique.

29.10.4. Pas encore de générateurs de parseurs depuis EBNF

Une direction d'évolution envisagée est l'**import** de grammaires depuis des formats standards comme EBNF, ABNF ou PEG.js. Un builtin hypothétique `[dialect:from-ebnf "..."]` pourrait à terme transformer une chaîne en grammaire VVL native. Cette fonctionnalité n'existe pas dans le MVP : elle exigerait, ironiquement, un parseur d'EBNF — qu'on pourrait écrire en VVL avec le dialecte présent, ce qui ouvre une boucle d'amorçage intéressante mais non triviale.

29.10.5. Performance et taille des entrées

Le moteur est implémenté en Python et opère sans optimisation de cache. Pour des entrées de quelques centaines de caractères, la performance est imperceptible. Pour des entrées de quelques milliers, elle reste raisonnable. Pour des entrées de plusieurs centaines de milliers de caractères — par exemple un fichier de log de plusieurs mégaoctets —, le moteur n'est pas le bon outil : on lui préférera un parseur Python natif via un plugin, ou un découpage préalable en lignes par `[split]` suivi d'un `[parse]` par ligne.

29.11. Synthèse : la grammaire au statut de donnée

Onze sections plus tard, on peut reformuler en une phrase ce que les dialectes apportent à VVL : **ils donnent aux grammaires le même statut que les programmes**. La grammaire est une liste, donc une donnée ; elle hérite de toute la machinerie de manipulation des listes ; elle peut être construite dynamiquement, stockée, transmise, transformée. C'est l'extension naturelle du principe homoiconique du chapitre 28 au domaine de l'ingestion.

Cette extension est précieuse pour trois raisons. **Première raison** : elle ferme un trou béant dans la chaîne *neural → symbolique*. Avant les dialectes, une sortie LLM mal-formée n'avait que deux destins — être injectée crue dans le runtime (et y causer un échec opaque) ou être filtrée par une cascade d'expressions régulières fragiles. Avec les dialectes, on dispose d'une **frontière formelle**, déclarée dans le même langage que le reste du programme, vérifiable et auditable. **Deuxième raison** : elle rend les DSL métier économiques. Un format de numéro de série, un mini-langage de filtres, une syntaxe de commande personnalisée — autant de problèmes qui exigeaient autrefois un parseur séparé et qui se résolvent maintenant en quelques lignes de VVL natif. **Troisième raison** : elle ouvre la composition. Un dialecte n'est pas un objet inerte ; c'est une donnée que l'on peut combiner, étendre, paramétrer, dans le même mouvement que l'on combine n'importe quelles autres listes du langage.

On notera aussi ce que les dialectes **n'apportent pas**. Ils ne remplacent pas `[match]` — le dispatch sur valeur déjà construite reste l'outil de la logique de programme. Ils ne remplacent pas `[load]` — le parsing de source VVL reste réservé au triangle homoiconique. Ils n'apportent pas un moteur Earley ni un système de captures arborescentes — pour les grammaires véritablement complexes, le plugin Python reste la voie. Et ils n'aspirent pas à être un substitut universel à yacc ou ANTLR — ils sont l'outil VVL pour les usages VVL, dans la philosophie *batteries-included-but-modest*.

29.11.1. Articulation avec les chapitres suivants

Le chapitre 30, qui suit, introduit les **modules** — un mécanisme qui partage avec les dialectes nommés une parenté de surface (registre dans le contexte, déclaration nommée, accès qualifié) mais s'en sépare nettement dans le rôle : un dialecte enregistre une **grammaire** (donnée passive consommable par `[parse]`) ; un module enregistre des **commandes** (procédures actives exposées via un namespace). Les deux mécanismes sont complémentaires : un module peut exporter des dialectes, et un dialecte peut être utilisé pour parser les arguments d'une commande exportée. La doctrine de design VVL distingue clairement les deux registres — un pour les données nommées, un pour les commandes nommées — sans chercher à les unifier dans un mécanisme abstrait commun.

Le chapitre 31 introduira les **valeurs scalaires typées** (`EMAIL`, `URL`, `DATE`, `UUID`). Ces types pourraient — théoriquement — être implémentés comme des dialectes VVL : un `EMAIL` n'est jamais qu'une chaîne qui passe une grammaire spécifique. La décision de les implémenter comme **types natifs** plutôt que comme dialectes est délibérée : elle s'appuie sur des considérations de performance (validation à la construction, normalisation systématique) et de typing (un `EMAIL` a un nom de type stable que `[type-`

`of]` retourne directement, sans avoir à inspecter une grammaire). Le rapport entre dialectes et types scalaires sera abordé directement au chapitre 31.

Enfin, le chapitre 33 sur les plugins exposera comment, lorsque le moteur de dialectes atteint ses limites — grammaires récursives complexes, performance critique, formats binaires —, on peut introduire un parseur Python natif via un plugin signé, sans rien perdre des garanties d'audit du langage hôte. Les dialectes, les plugins, les modules : trois mécanismes complémentaires pour les trois échelles d'extension de VVL — *je l'écris en VVL*, *je le délègue à Python signé*, *je le réutilise dans plusieurs scripts*.

29.11.2. Un dernier coup d'œil

Pour fixer l'image dans la mémoire du lecteur, on peut résumer le moteur de dialectes en cinq points : *(1)* la grammaire est une **liste VVL**, donc une donnée comme les autres ; *(2)* le moteur consomme une **string** ou une **liste** selon le type de l'entrée, avec une sémantique adaptée ; *(3)* les règles primitives consomment un atome, les quantifieurs en consomment plusieurs, `[alt]` choisit entre branches, `[capture]` annote ; *(4)* le résultat est toujours un **JSON uniforme** `{ok, position, captures, rest, expected}` — succès ou échec, jamais d'exception ; *(5)* `[dialect:def]` permet d'enregistrer une grammaire sous un nom pour la réutiliser sans inlining.

Le lecteur qui retiendra ces cinq points — et qui aura lu attentivement les sections 29.3 à 29.6 sur la mécanique fine — possède l'essentiel pour écrire ses propres dialectes au quotidien. Les cas d'usage de la section 29.8 ne sont qu'un échantillon : la combinatoire offerte par le moteur est suffisante pour couvrir, à l'estimation des auteurs, plus de 90 % des besoins de parsing d'un programme VVL typique. Les 10 % restants relèvent du plugin Python, comme on le verra au chapitre 33.

30. Modules : namespaces et packages réutilisables

Mots-clés : *module, namespace, package, export, qualification, import, module:def, module:get, module:load, module:reload, encapsulation, frontière de package, collision-safety, hot-reload*

Avec le chapitre 28 sont arrivés `[load]`, `[evaluate]`, `[dump]` — le triangle homoiconique qui élève le code VVL au rang de donnée. Avec le chapitre 29 sont arrivés `[parse]` et les dialectes — qui élèvent les grammaires au même statut. Il reste un troisième mécanisme à exposer pour que VVL soit complet sur sa face *infrastructure* : le moyen d'**organiser le code** lui-même en unités réutilisables, isolées et identifiables. C'est l'objet de ce chapitre, qui introduit les **modules**.

Le terme *module* est, dans le vocabulaire des langages de programmation, l'un des plus chargés et des plus polysémiques. Pour Python, un module est un fichier `.py` ; pour Java, c'est un système de packages avec règles de visibilité ; pour Erlang, c'est une unité de compilation ; pour OCaml, c'est un type structuré qui peut contenir d'autres modules. VVL, dans son MVP actuel, adopte une définition **délibérément modeste** : un module est un **namespace nommé** qui contient des valeurs et des keywords explicitement déclarés comme exportés, dont le corps d'exécution est isolé du contexte appelant. Pas de hiérarchie de packages, pas de système de types modulaires, pas d'interfaces formelles. Juste une frontière nette entre ce qui est public et ce qui est privé, et un mécanisme propre pour appeler à travers cette frontière.

Cette modestie est volontaire. Le chapitre va expliquer pourquoi un mécanisme minimaliste suffit, dans la pratique, à résoudre la grande majorité des besoins de structuration que rencontre un programme VVL réel — et pourquoi les fonctionnalités plus avancées qu'on pourrait souhaiter (hiérarchies, dépendances déclarées, versioning) sont délibérément absentes du MVP.

30.1. La nécessité d'une frontière de package

Le besoin auquel répondent les modules ne s'est imposé que progressivement au fil du livre. Pour le comprendre, il faut revenir sur les mécanismes pré-existants de partage de code en VVL, et identifier précisément ce qu'ils ne savent pas faire.

30.1.1. Les limites d'[import]

Le builtin `[import]`, présenté au chapitre 28 dans la matrice `load/evaluate/import`, sait charger un fichier `.vvl` et l'**exécuter dans le contexte courant**. Toutes les définitions du fichier — variables, keywords, règles ontologiques, dialectes — deviennent immédiatement accessibles dans le contexte appelant. Pour des scripts d'initialisation ou des fichiers de configuration, c'est exactement ce qu'on veut.

Mais dès qu'on essaie d'utiliser `[import]` comme un mécanisme de **bibliothèque réutilisable**, trois limitations apparaissent rapidement. **Première limite — pas d'encapsulation** : tout ce que le fichier importé définit fuit dans le contexte appelant. Si la bibliothèque utilise une variable interne `$cache` pour son fonctionnement, cette variable devient accessible — et modifiable — depuis tout le code qui suit l'import. **Deuxième limite — collisions silencieuses** : si deux fichiers importés définissent un keyword du même nom, le second écrase le premier sans avertissement. **Troisième limite — pas de groupement nominal** : il n'existe aucune manière de dire « telle commande appartient à telle bibliothèque » ; tout vit à plat dans le namespace global.

Pour un script simple, ces limitations sont supportables. Pour une base de code qui assemble dix bibliothèques, qui veut suivre la traçabilité des définitions, ou qui doit garantir qu'aucune collision ne fasse apparaître silencieusement un comportement erroné, l'absence de frontière de package devient un véritable handicap. Le mécanisme des modules vise précisément à fournir cette frontière.

30.1.2. Modules vs dialectes : commandes nommées vs données nommées

Le chapitre 29 a introduit `[dialect:def]`, un mécanisme qui semble parent : on enregistre quelque chose sous un nom dans le contexte. Pourquoi alors avoir besoin d'un système séparé pour les modules ? La réponse tient en deux mots : *commandes* vs *données*.

Un dialecte est une **donnée** : c'est une grammaire stockée sous un nom, qu'on récupère pour la passer à un moteur de parsing. Un dialecte n'est jamais *appelé* directement ; il est *consommé* par `[parse]`. Un module, à l'inverse, contient des **commandes actives** — des keywords qu'on appelle, des valeurs qu'on lit. Le verbe associé n'est pas le même : on *parse avec* un dialecte, on *invoque* un module.

Cette distinction n'est pas une finesse théorique : elle conditionne la sémantique de l'appel. Pour appeler un dialecte, il faut **passer son nom** à un autre builtin (`[dialect:parse alphaNum "abc123"]`). Pour appeler un module, il faut **qualifier le nom de la commande** par le nom du module (`[Math:add 1 2]`). Le préfixe `Math:` est *intrinsèquement* lié au keyword `add` ; sans lui, le runtime ne sait pas où chercher. À l'inverse, un dialecte est juste *un argument* parmi d'autres.

La doctrine de design VVL est donc claire : *un registre pour les données nommées* (dialectes, mais aussi à l'avenir d'éventuelles autres familles), *un registre pour les commandes nommées* (modules). Les deux mécanismes vivent dans le même contexte mais ne se concurrencent jamais.

30.1.3. Modules vs plugins : code natif VVL vs code externe signé

Un troisième mécanisme d'extension sera introduit au chapitre 33 : les **plugins signés**, qui permettent d'embarquer dans VVL des commandes implémentées en Python (ou VScript) et dont la confiance est ancrée dans une signature Ed25519. À première vue, plugins et modules visent un objectif voisin : exposer des commandes nommées et qualifiables. La différence se joue sur deux axes essentiels.

Le langage d'implémentation : un module est écrit en **VVL natif** — son corps est exécuté par le runtime VVL ordinaire. Un plugin est écrit en **Python** (ou VScript) — son corps est exécuté par un sous-processus ou un moteur sandboxé. **Le modèle de confiance** : un module ne demande aucune signature, parce que son code est inspectable comme n'importe quel autre code VVL. Un plugin exige une signature Ed25519 vérifiée contre un magasin de clés, parce que son code peut sortir des garanties du langage hôte. Les deux mécanismes sont **complémentaires** : on écrit un module quand la fonctionnalité est exprimable en VVL ; on écrit un plugin quand il faut intégrer une bibliothèque externe (numpy, scikit-learn, un appel HTTP) ou atteindre des performances que le runtime VVL ne permet pas.

Concrètement, dans la vie d'un projet VVL réel, on a typiquement quelques plugins (pour les ponts vers Python ML, les connecteurs API, l'accès au système de fichiers) et plusieurs dizaines de modules (pour la logique métier, les bibliothèques mathématiques internes, les pipelines de transformation). Le triptyque *dialecte (donnée) / module (code VVL) / plugin (code externe signé)* couvre les trois échelles d'extension du langage.

30.2. Anatomie d'un module

Un module se déclare par `[module:def Name [export name1 name2 ...] body...]`. Le builtin est **raw** : ni le nom ni le corps ne sont pré-évalués. Le runtime ouvre un **scope isolé**, exécute le corps dans ce scope, et expose à l'extérieur uniquement les noms listés dans la liste `[export ...]`. La figure ci-dessous montre cette anatomie.

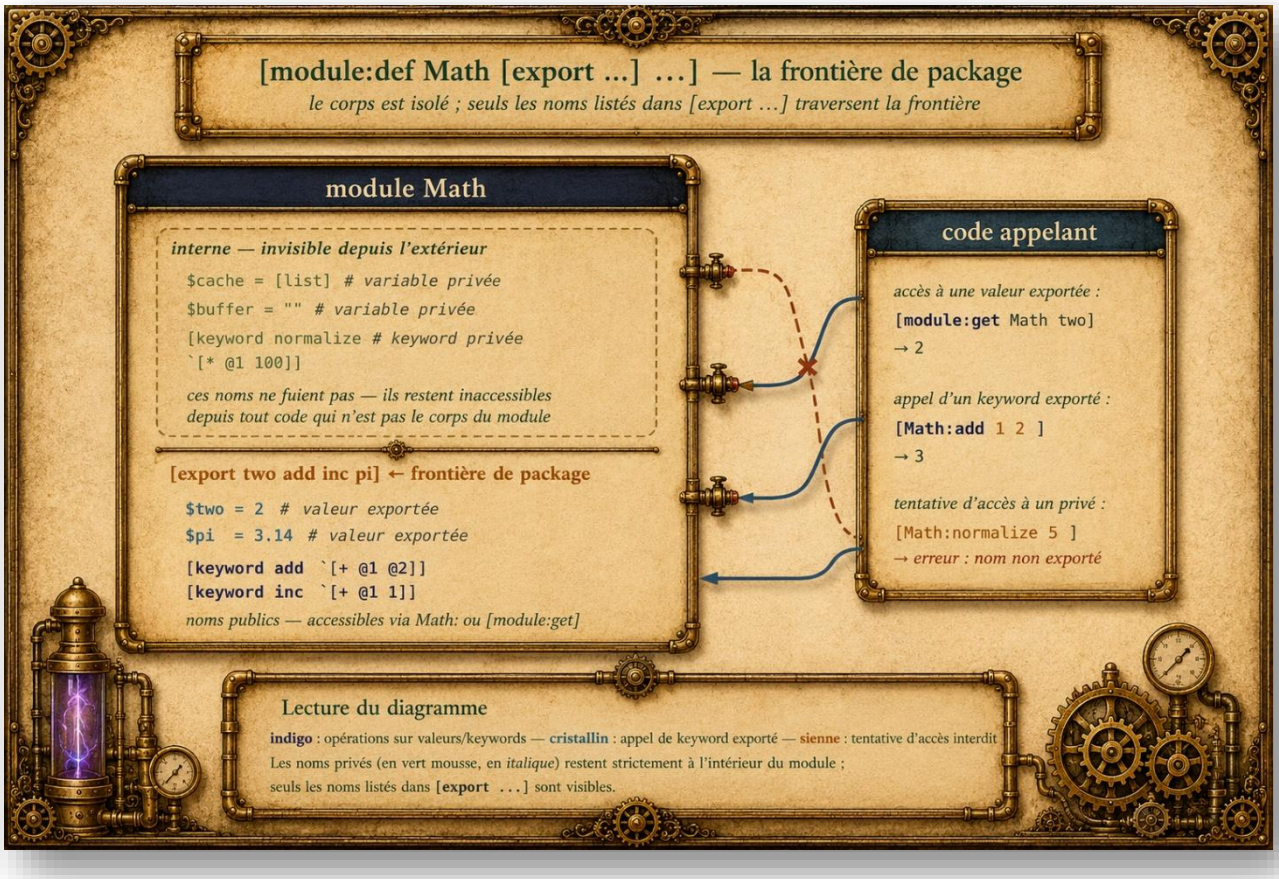


Figure 30.1 — *La frontière de package. Le corps du module exécute en scope isolé ; seuls les noms listés dans [export ...] sont visibles depuis l'extérieur. Les noms privés (en vert mousse) ne fuient jamais.*

30.2.1. Une première déclaration

L'exemple canonique commence par un module mathématique minimaliste : une valeur exportée, un keyword exporté.

```
# Un premier module Math, avec une valeur et un keyword

[module:def Math [export two add]
    $two = 2
    [keyword add `[+ @1 @2]]
]

# Acces a la valeur exportee : via [module:get]
[log [module:get Math two]]       # => 2

```

```
# Appel du keyword exporte : via le prefixe qualifie
[log [Math:add 1 2]]              # => 3
```

Trois éléments structurent la déclaration. **Premièrement**, le nom du module — ici `Math` — est un identifiant choisi par convention en *PascalCase*. Le runtime n'impose pas cette convention, mais elle est partagée par toute la base de code VVL parce qu'elle distingue visuellement les modules (qui sont des espaces de noms) des keywords (qui sont en *camelCase* ou *kebab-case*). **Deuxièmement**, la liste `[export two add]` énumère **explicitement** ce qui sera visible depuis l'extérieur. Pas d'export implicite, pas de découverte automatique : ce qui n'est pas dans la liste reste privé. **Troisièmement**, le corps `$two = 2 [keyword add` [+ @1 @2]]` est exécuté **une fois**, au moment de la déclaration, dans un scope isolé.

30.2.2. Le scope isolé

Le corps d'un module s'exécute dans un **scope dédié**. Toute variable créée par une affectation `$x = ...` est locale au module ; tout keyword créé par `[keyword ...]` l'est aussi. Aucune de ces définitions ne fuit dans le contexte appelant — sauf si elle figure dans la liste d'exports.

```
# Un module avec des elements prives et publics

[module:def Counter [export current incr]
    $count = 0
    $step  = 1
    [keyword incr `[do
        ($count = ($count + $step))
        $count
    ]]
    [keyword current `$count]
]

# Acces autorise
[log [Counter:current]]           # => 0
[Counter:incr]
[log [Counter:current]]           # => 1

# Acces refuse : $count et $step ne sont pas exportes
[log $count]
# => erreur : variable non liee
[log [module:get Counter step]]
# => erreur : "step" pas dans les exports de Counter
```

L'effet de cette isolation est double. **Du côté de l'auteur du module** : il peut nommer ses variables internes sans se soucier de la dénomination utilisée ailleurs ; les conflits avec le contexte appelant sont impossibles. **Du côté de l'utilisateur du module** : il peut composer plusieurs modules dans le même script sans craindre que les variables internes d'un module n'écrasent celles d'un autre. C'est ce qu'on appelle, en théorie des langages, la *garantie d'opacité par défaut* — VVL la fournit gratuitement dès qu'un code est encapsulé dans un module.

30.2.3. Le corps s'exécute une fois — pas à chaque appel

Une propriété subtile mais importante : le corps du module est évalué **une seule fois**, au moment de la déclaration. Si le corps contient des effets de bord — initialisation d'un cache, lecture d'un fichier de configuration, requête HTTP — ces effets se produisent une fois et leur résultat est figé dans le scope du module.

```
# Le corps execute une fois : effet de bord visible une seule fois

[module:def Config [export get]
    [log "Chargement de la config..."]
    $values = [load-config "config.json"]
    [keyword get `[jsonGetValue $values @1]]
]
# Affiche : Chargement de la config...

[log [Config:get "host"]]
# N'affiche PAS de nouveau "Chargement de la config..." :
# la lecture du fichier a eu lieu lors du [module:def].
```

Ce modèle d'**exécution unique** correspond à l'idiome de la plupart des langages modernes — Python (`import` exécute le module une fois), JavaScript (les modules ES6 sont évalués à la première importation), Erlang. Pour les modules qui agissent comme des **constructeurs uniques** — bibliothèques d'utilitaires, registres de configuration, caches partagés —, c'est exactement le comportement attendu. Pour les rares cas où l'on aurait besoin de **réexécuter** le corps (typiquement pour rafraîchir une configuration ou recharger une bibliothèque modifiée), le builtin `[module:reload]` sera examiné en 30.5.

30.2.4. La déclaration retourne le nom du module

Un dernier détail utile : `[module:def Name ...]` retourne, en tant qu'expression VVL, le **nom du module** sous forme de chaîne. Cette valeur de retour est rarement utilisée explicitement, mais elle permet d'écrire des séquences fluides ou de valider la création :

```
$created = [module:def Math [export two] $two = 2]
[log $created]                    # => "Math"

# Combine avec [module:list] (cf. 30.5) pour verifier l'enregistrement
[when [exist [module:list] $created]
    `[log "Module " $created " enregistre"]
]
```

30.3. Exports : valeurs et keywords

La liste `[export ...]` peut contenir deux types d'éléments : des **noms de variables** (qui désignent des valeurs liées dans le corps) et des **noms de keywords** (qui désignent des commandes définies

dans le corps). VVL traite les deux uniformément dans la liste d'exports — il n'y a pas de distinction syntaxique — mais leur **mécanisme d'accès** depuis l'extérieur diffère.

30.3.1. Accès aux valeurs : [module:get]

Une valeur exportée se lit par `[module:get Name varName]`. Le builtin est **raw** : ni `Name` ni `varName` ne sont pré-évalués — ce sont des noms symboliques résolus en interne par le runtime des modules.

```
001  # Acces aux valeurs exportees
002
003  [module:def Constants [export pi e tau]
004      $pi  = 3.14159
005      $e   = 2.71828
006      $tau = 6.28318
007  ]
008
009  [log [module:get Constants pi]]   # => 3.14159
010  [log [module:get Constants e]]    # => 2.71828
011  [log [module:get Constants tau]]  # => 6.28318
```

Le choix de passer par un builtin plutôt que par une syntaxe pointée — par exemple `Constants.pi` — est délibéré. Une syntaxe pointée aurait introduit une ambiguïté avec la **navigation JSON** déjà bien établie en VVL (`$obj.field.subfield` pour les structures JSON). En réservant le point pour la navigation dans les structures de données, et le `:` pour la qualification dans les namespaces, VVL maintient une séparation visuelle qui aide à la lecture.

Si le nom n'est pas dans les exports du module — soit qu'il existe en interne sans être déclaré, soit qu'il n'existe pas du tout —, `[module:get]` lève une erreur claire qui distingue les deux cas. Cette transparence est importante pour le débogage : un développeur qui essaie d'accéder à `[module:get Math step]` (alors que `step` est privé) reçoit un message qui lui dit explicitement *« step n'est pas dans les exports de Math »*, plutôt qu'un échec opaque ou silencieux.

30.3.2. Appel aux keywords : la qualification Module:nom

Un keyword exporté s'appelle directement par son nom qualifié : `[Module:keyword args...]`. La syntaxe est uniforme avec l'appel d'un keyword non qualifié — c'est juste le préfixe `Module:` qui change la résolution. Le runtime, lorsqu'il rencontre un keyword préfixé, cherche d'abord dans le module nommé ; si le nom n'y est pas exporté, la résolution échoue avec un message explicite.

```
001  # Appel qualifie
002
003  [module:def Geometry [export square circle]
004      [keyword square `[* @1 @1]]
005      [keyword circle `[* 3.14159 (@1 * @1)]]
006  ]
007
008  [log [Geometry:square 5]]         # => 25
009  [log [Geometry:circle 3]]         # => 28.27431
```

La syntaxe `Module:keyword` est, à proprement parler, **une partie du nom du keyword** — pas un opérateur appliqué au keyword. Pour le runtime, `Geometry:square` est un identifiant unique, différent de `square`. Cette uniformité simplifie l'implémentation et garantit qu'aucun mécanisme de dispatch dynamique n'est nécessaire à l'appel : la résolution se fait au moment de la lecture du nom, comme pour n'importe quel autre identifiant.

30.3.3. Pourquoi deux mécanismes distincts

Le lecteur attentif pourra se demander pourquoi VVL n'offre pas un unique mécanisme d'accès — par exemple, tout passer par `[module:get]`, y compris pour les keywords. La réponse tient à la **différence sémantique** entre valeur et commande. Une valeur est une donnée qu'on **lit**, en lecture seule, sans paramètres. Une commande est une opération qu'on **appelle**, avec des arguments, qui produit un effet ou un résultat. Le runtime ne peut pas appliquer le même protocole d'invocation aux deux : la syntaxe `[Module:keyword arg1 arg2]` doit, dès le parsing, savoir qu'elle parle à un keyword pour empiler correctement les arguments.

On peut résumer la division du travail par le tableau suivant :

Élément exporté	Accès	Sémantique
valeur (variable liée)	[module:get Module nom]	lecture seule, pas d'arguments
keyword	[Module:nom arg1 arg2 ...]	appel avec arguments, produit un résultat

Cette séparation explicite, même si elle alourdit légèrement la grammaire, est gage de clarté : un lecteur qui voit `[module:get X y]` sait sans ambiguïté qu'il s'agit d'une lecture de valeur ; un lecteur qui voit `[X:y args]` sait que c'est un appel. La cognition est plus immédiate que si le même opérateur pouvait désigner les deux.

30.4. La qualification des keywords et la collision-safety

Le préfixe `Module:` n'a pas qu'une fonction de groupement nominal — il a aussi une fonction de **protection**. La règle est stricte et mérite d'être énoncée formellement : *un keyword exporté par un module ne masque jamais un keyword non qualifié, qu'il soit builtin ou défini par l'utilisateur.* Cette garantie est ce qui fait des modules un mécanisme **collision-safe** par construction.

30.4.1. Le cas critique : exporter un nom de builtin

Pour fixer la portée de la garantie, considérons le cas le plus extrême : un module qui exporte un keyword dont le nom est identique à un builtin déjà présent dans le langage.

```
001  # Un module qui exporte un keyword nomme "log"
002
003  [module:def LogTools [export log]
004      [keyword log `[+ @1 @2]]
005  ]
006
007  # Le builtin [log] continue de fonctionner sans modification
008  [log "message de tracabilite"]
```

```
# => log normal, affiche dans la sortie

# L'export du module est accessible UNIQUEMENT via la qualification
[log [LogTools:log 10 5]]
# => 15  (la fonction du module, qui additionne ses deux arguments)
```

Le module a déclaré `log` comme export, mais cet export **ne s'est pas substitué** au builtin `[log]`. Les deux coexistent. La résolution se fait au moment du parsing : si le nom est *qualifié* (`LogTools:log`), le runtime cherche dans le module ; si le nom est *nu* (`log`), le runtime cherche dans la priorité standard (builtins > keywords utilisateur > erreur).

Cette propriété est ce qui rend les modules **utilisables sans peur** dans une base de code complexe. On peut, sans dommage, importer cent modules qui exportent tous un keyword `format` : aucun ne masquera le builtin `[format]` ni les uns les autres. La qualification est *obligatoire* pour distinguer les appels, mais c'est précisément ce qui rend la cohabitation possible.

30.4.2. Priorités de résolution

Pour fixer définitivement la sémantique, voici l'ordre exact dans lequel le runtime résout un identifiant rencontré dans une position d'appel.

Forme du nom	Résolution
Module:keyword	cherche dans ctx.modules[Module].exports.keywords[keyword] ; erreur sinon
keyword (nu)	cherche dans (1) builtins, (2) keywords utilisateur, (3) erreur
Module:variable via [module:get]	cherche dans ctx.modules[Module].exports.values[variable]
$variable (nu)	cherche dans le scope courant ; erreur sinon

Aucune de ces résolutions n'altère les autres. La présence d'un module nommé `Builtin` qui exporterait `log` n'affecte en rien le `[log]` global. La présence d'un module `Math` n'affecte pas une variable `$pi` définie dans le contexte courant. Les modules vivent dans leur propre dimension de noms, séparée par construction des autres dimensions.

30.4.3. Le pattern d'ouverture multi-modules

Une conséquence agréable de cette discipline est qu'un script peut ouvrir simultanément un grand nombre de modules sans aucun risque d'interférence. Le pattern idiomatique est de placer les déclarations de modules — ou les `[module:load ...]` qui les chargent — en début de fichier, dans une sorte de *prélude*.

```
# Prelude d'un script qui combine plusieurs modules

[module:load "lib/math.vvl"]
[module:load "lib/strings.vvl"]
[module:load "lib/dates.vvl"]

# Code metier qui utilise les trois bibliotheques
```

```
009 $total = [Math:add 100 200]
010 $label = [Strings:format "Total : {0}" $total]
011 $when  = [Dates:now]
012
013 [log $label " a " $when]
```

Aucune des trois bibliothèques n'interfère avec les autres ni avec les builtins. Si demain on ajoutait une quatrième `[module:load "lib/colors.vvl"]`, le code existant continuerait à fonctionner sans modification. C'est précisément la *propriété additive* qu'un système de modules sain doit garantir.

30.5. Chargement de fichiers : [module:load] et [module:reload]

Jusqu'ici, les exemples ont placé les déclarations `[module:def ...]` directement dans le code appelant. Pour des bibliothèques réutilisables, ce n'est pas le pattern naturel : on veut écrire la bibliothèque **une fois** dans un fichier dédié, et la charger depuis tous les scripts qui en ont besoin. C'est le rôle de `[module:load]`.

30.5.1. La sémantique de base

`[module:load "chemin/vers/fichier.vvl"]` ouvre le fichier, l'exécute, et enregistre tous les modules qu'il contient dans le contexte courant. Le fichier doit contenir, typiquement, une ou plusieurs déclarations `[module:def ...]`, mais peut aussi contenir du code d'initialisation accessoire (logs, vérifications) qui s'exécutera une fois.

```
001 # Fichier : lib/math.vvl
002 #
003 # [module:def Math [export zero add inc]
004 #     $zero = 0
005 #     [keyword add `[+ @1 @2]]
006 #     [keyword inc `[+ @1 1]]
007 # ]
008
009 # Fichier appelant : main.vvl
010 [module:load "lib/math.vvl"]
011
012 [log [Math:inc 41]]              # => 42
013 [log [module:get Math zero]]     # => 0
```

Le chemin passé à `[module:load]` est **relatif au workspace courant**. Cette convention rend les fichiers portables : un script qui fonctionne dans un workspace donné fonctionnera dans n'importe quel autre workspace qui contient le même fichier `lib/math.vvl`. Pour des bibliothèques système partagées entre workspaces, les conventions de déploiement précisent l'emplacement standard ; le présent chapitre n'entre pas dans ces détails opérationnels.

30.5.2. Le cache et l'idempotence

Une propriété essentielle de `[module:load]` : un fichier donné n'est lu et exécuté **qu'une seule fois** par session. Si le même `[module:load "lib/math.vvl"]` est appelé trois fois dans le code, seul le premier appel a un effet ; les suivants retournent silencieusement sans rien faire. Cette **idempotence** est gardée par un cache interne au runtime.

```
# Trois appels successifs
[module:load "lib/math.vvl"]        # premier appel : lit et execute
[module:load "lib/math.vvl"]        # deuxieme : noop (deja en cache)
[module:load "lib/math.vvl"]        # troisieme : noop egalement

# Math:add est disponible apres le premier appel et le reste.
[log [Math:add 10 20]]            # => 30
```

Cette idempotence est cruciale pour permettre la composition. Imaginons trois bibliothèques `A`, `B`, `C` qui dépendent toutes de `lib/utils.vvl` : chacune commence par `[module:load "lib/utils.vvl"]` au début de son propre fichier. Sans cache, le fichier `utils.vvl` serait exécuté trois fois, recréant son module à chaque fois, avec des effets de bord potentiellement multipliés. Avec cache, l'exécution n'a lieu qu'à la première rencontre et les chargements ultérieurs sont des no-ops sûrs.

30.5.3. Forcer le rechargement : [module:reload]

L'idempotence est généralement souhaitable, mais elle pose problème dans un cas précis : le **développement interactif**. Si un développeur modifie `lib/math.vvl` et veut tester sa nouvelle version sans redémarrer la session, l'idempotence de `[module:load]` l'empêche. Le runtime fournit pour ce cas le builtin `[module:reload]`, qui **force** la re-exécution du fichier en invalidant son entrée de cache.

```
# Cycle de developpement interactif

[module:load "lib/math.vvl"]
[log [Math:add 1 2]]              # => 3 (ancienne version)

# (Le developpeur modifie lib/math.vvl pour que add fasse +10)

[module:load "lib/math.vvl"]
[log [Math:add 1 2]]              # => 3 toujours (cache)

[module:reload "lib/math.vvl"]
[log [Math:add 1 2]]              # => 13 (nouvelle version)
```

`[module:reload]` est l'analogue VVL du `importlib.reload(...)` de Python. Il faut l'utiliser avec discernement : si d'autres parties du code détiennent des références fermées sur des keywords du module rechargé, ces références peuvent rester pointées sur l'ancienne version pendant un certain temps. Pour les cas où la cohérence à 100 % est exigée (typiquement dans les pipelines de production), la pratique recommandée reste de redémarrer la session ; `[module:reload]` est avant tout un confort de développement, pas une primitive de production critique.

30.5.4. Pattern de structuration d'une bibliothèque

L'organisation idiomatique d'une bibliothèque VVL est de regrouper les modules associés dans un sous-répertoire, avec un fichier par module ou par famille fonctionnelle, et un fichier *index* qui charge l'ensemble.

```
# Structure recommandee :
#   lib/
#     index.vvl       (charge tous les modules ci-dessous)
#     math.vvl        (module Math)
#     strings.vvl     (module Strings)
#     dates.vvl       (module Dates)
#     logging.vvl     (module Log)

# Fichier : lib/index.vvl
[module:load "lib/math.vvl"]
[module:load "lib/strings.vvl"]
[module:load "lib/dates.vvl"]
[module:load "lib/logging.vvl"]

# Dans les scripts metier, un seul [import] suffit ensuite :
#
# [import "lib/index.vvl"]
#
# (ou [module:load "lib/index.vvl"] si index.vvl est lui-meme un module)
```

Cette discipline rend la base de code lisible et maintenable. Chaque fichier a une responsabilité unique et nommée. L'ajout d'une nouvelle bibliothèque se fait en deux étapes : créer le fichier `lib/newlib.vvl` avec son `[module:def]`, et ajouter une ligne dans `lib/index.vvl`. C'est la structuration la plus simple possible qui couvre l'écrasante majorité des besoins pratiques.

30.6. Modules vs import vs dialectes vs plugins

Quatre mécanismes de réutilisation et d'extension cohabitent désormais en VVL. Le tableau ci-dessous synthétise leur position respective.

Mécanisme	Quoi	Encapsulation	Confiance	Cas d'usage
[import "f.vvl"]	exécute un fichier dans le contexte courant	aucune — toutes les définitions fuient	fichier local du workspace	configuration, initialisation, prélude
[dialect:def name ...]	enregistre une grammaire sous un nom	la grammaire est une donnée	donnée du contexte courant	parsing de DSL, validation d'entrée
[module:def Name ...]	enregistre un namespace nommé	scope isolé, exports déclarés	code VVL natif	bibliothèques réutilisables, organisation du code
Plugins signés (ch. 33)	enregistre une commande implémentée en Python	sous-processus ou inproc sandboxé	signature Ed25519 + magasin de clés	ponts externes, ML, I/O système

Le tableau fait apparaître une **gradation** dans la sévérité du modèle de confiance et dans la profondeur de l'encapsulation. À une extrémité, `[import]` est totalement permissif — c'est juste une exécution de fichier. À l'autre extrémité, les plugins signés exigent une vérification cryptographique. Les modules occupent une position intermédiaire raisonnée : l'encapsulation est garantie mais la confiance reste celle du langage hôte, sans signature externe.

30.6.1. Quand choisir quoi

Les règles pratiques de choix entre les quatre mécanismes peuvent se résumer ainsi. **Pour de la configuration ou de l'initialisation** — variables globales d'une session, déclarations de constantes — on utilise `[import]`. **Pour des grammaires de parsing** — formats métier, validation d'entrée — on utilise `[dialect:def]`. **Pour des bibliothèques de commandes VVL réutilisables** — utilitaires mathématiques, helpers de strings, wrappers d'API simples — on utilise `[module:def]`. **Pour des fonctionnalités qui ne peuvent pas s'exprimer en VVL** — appels à des modèles ML Python, accès à des bibliothèques système, opérations cryptographiques — on utilise un plugin signé.

Dans une base de code mature, les quatre mécanismes coexistent : un fichier `setup.vvl` que l'on `[import]` au démarrage charge les modules de la bibliothèque maison, enregistre quelques dialectes de validation, et déclare les plugins externes que les agents utiliseront. Ce qui était auparavant un fouillis de définitions globales devient une architecture lisible et auditable.

30.7. Cas d'usage

La présentation théorique étant complète, déroulons trois cas d'usage qui montrent comment les modules s'inscrivent dans des situations concrètes. Comme pour les chapitres précédents, aucun n'est artificiel : tous trois reprennent des patterns qui apparaissent réellement dans la base de code VVL.

30.7.1. Bibliothèque mathématique

L'usage canonique d'un module est d'encapsuler une famille cohérente d'opérations utilitaires. L'exemple ci-dessous esquisse une petite bibliothèque mathématique avec des constantes et des fonctions élémentaires.

```
001  # Fichier : lib/math.vvl
002
003  [module:def Math
004      [export pi e tau zero square cube hypot mean]
005
006      # Constantes
007      $pi   = 3.14159
008      $e    = 2.71828
009      $tau  = 6.28318
010      $zero = 0
011
012      # Operations elementaires
013      [keyword square `[* @1 @1]]
014      [keyword cube   `[* @1 (* @1 @1)]]
015      [keyword hypot  `[sqrt (+ (* @1 @1) (* @2 @2))]]
```

```
016
017     # Moyenne d'une liste
018     [keyword mean   `[/ [sum @1] [length @1]]]
019 ]
```

L'organisation suit une convention simple : les constantes en premier (sans préfixe particulier), les opérations élémentaires ensuite, les fonctions composées à la fin. L'usage depuis un script reste limpide.

```
001 [module:load "lib/math.vvl"]
002
003 [log [module:get Math pi]]              # => 3.14159
004 [log [Math:square 7]]                   # => 49
005 [log [Math:hypot 3 4]]                  # => 5
006 [log [Math:mean `[10 20 30 40]]]        # => 25
```

Le code appelant n'a jamais à se soucier des éventuelles variables internes que la bibliothèque utiliserait pour son fonctionnement. Et si demain la bibliothèque ajoutait une dépendance interne — par exemple un cache de résultats pour `hypot` — cette dépendance resterait privée et invisible des utilisateurs.

30.7.2. Logger avec niveaux et contexte

Un second cas d'usage met en évidence la valeur des **valeurs internes mutables** que la frontière protège. Une bibliothèque de logging structurée :

```
001 # Fichier : lib/logging.vvl
002
003 [module:def Log
004     [export setLevel debug info warn error]
005
006     # Etat interne du logger : niveau courant et compteur
007     $level = 2          # 0=debug 1=info 2=warn 3=error
008     $count = 0
009
010     # Reglage du niveau (effet de bord sur l'etat interne)
011     [keyword setLevel `[do ($level = @1) "OK"]]
012
013     # Les quatre niveaux
014     [keyword debug `[when ($level <= 0)
015         `[do ($count = ($count + 1))
016              [log "[DEBUG]" @1]]]]
017     [keyword info  `[when ($level <= 1)
018         `[do ($count = ($count + 1))
019              [log "[INFO]"  @1]]]]
020     [keyword warn  `[when ($level <= 2)
```

```
        `[do ($count = ($count + 1))
            [log "[WARN]"  @1]]]]
    [keyword error `[when ($level <= 3)
        `[do ($count = ($count + 1))
            [log "[ERROR]" @1]]]]
]
```

Trois propriétés notables. **Premièrement**, le module conserve un **état interne mutable** (`$level`, `$count`) qui survit d'un appel à l'autre sans fuiter dans le contexte appelant. **Deuxièmement**, le runtime expose une commande de réglage `setLevel` qui modifie cet état — un pattern classique de *setter* qui aurait nécessité une classe complète dans un langage orienté-objet. **Troisièmement**, les quatre commandes de niveau (`debug`, `info`, `warn`, `error`) coexistent gracieusement avec le builtin `[log]`, qu'elles appellent en interne — la qualification garantit qu'aucune ambiguïté ne peut s'installer.

```
[module:load "lib/logging.vvl"]

[Log:info "demarrage du systeme"]   # ignore (niveau 2 >= 1)
[Log:warn "config par defaut"]      # affiche : [WARN] config...

[Log:setLevel 1]                    # baisse au niveau INFO
[Log:info "phase 1 terminee"]       # affiche : [INFO] phase 1...

[Log:setLevel 3]                    # remonte au niveau ERROR
[Log:warn "ce sera ignore"]         # ignore (niveau 3 > 2)
```

30.7.3. Mini-domain library avec composition de modules

Un dernier cas d'usage montre comment plusieurs modules peuvent **se composer** pour bâtir une bibliothèque de domaine. Imaginons une mini-bibliothèque pour le diagnostic médical, qui combine constantes physiologiques, règles de seuil, et formatage des résultats.

```
# Fichier : medical/vital.vvl
[module:def Vital [export systolic-normal heart-rate-normal]
    $systolic-normal    = `[90 120]
    $heart-rate-normal  = `[60 100]
]

# Fichier : medical/diagnose.vvl
[module:load "medical/vital.vvl"]

[module:def Diagnose [export bp-status hr-status]
    [keyword bp-status `[do
        $range = [module:get Vital systolic-normal]
        [match @1
            (< [getValue $range 0]) `"hypotension"
            (> [getValue $range 1]) `"hypertension"
            _                       `"normal"
        ]
```

```
    ]]
    [keyword hr-status `[do
        $range = [module:get Vital heart-rate-normal]
        [match @1
            (< [getValue $range 0]) `"bradycardia"
            (> [getValue $range 1]) `"tachycardia"
            _                       `"normal"
        ]
    ]]
]
```

Le module `Diagnose` **réutilise** les valeurs exportées par `Vital` via `[module:get]`. Cette composition est totalement explicite — le lecteur voit dans le code de `Diagnose` qu'il consomme `Vital`. Aucune importation magique, aucun namespace global pollué : chaque dépendance est nommée. Un script consommateur final n'a qu'à charger `Diagnose`, qui se charge à son tour de charger ses propres dépendances.

```
[module:load "medical/diagnose.vvl"]

[log [Diagnose:bp-status 145]]    # => hypertension
[log [Diagnose:bp-status 110]]    # => normal
[log [Diagnose:hr-status 50]]     # => bradycardia
```

Ce pattern — modules qui se composent verticalement, du plus bas niveau (constantes) au plus haut (logique métier) — est l'organisation idiomatique d'une base de code VVL bien structurée. Le langage offre, en quelques pages de chapitre, l'outillage nécessaire pour passer de scripts plats à une architecture proprement stratifiée.

30.8. Patterns avancés

Au-delà des trois cas d'usage canoniques, deux patterns plus avancés méritent une présentation rapide. Ils ne sont pas requis dans la pratique quotidienne mais ils éclairent la flexibilité du mécanisme.

30.8.1. Tests intégrés à la racine du module

Le chapitre 32 introduira un système de tests (`test:assert`, `test:equals`, `test:suite`) qui permet de vérifier le comportement attendu d'une bibliothèque. Le pattern recommandé consiste à placer une **suite de tests** directement après chaque déclaration `[module:def]`, dans le même fichier. La suite s'exécute alors **à chaque chargement** du fichier — ou à chaque rechargement par `[module:reload]` — et fournit un retour immédiat sur la santé de la bibliothèque.

```
# Fichier : lib/math.vvl  (avec tests integres)

[module:def Math [export inc]
    [keyword inc `[+ @1 1]]
```

```
]

# Suite de tests collee au module
[test:suite "Math"
    [test:equals 42 [Math:inc 41] "inc de 41"]
    [test:equals 0  [Math:inc -1] "inc de -1"]
]
```

Cette discipline a deux vertus. **Première vertu** : la bibliothèque est *autodocumentée par ses tests* — un lecteur qui ouvre `lib/math.vvl` voit immédiatement, à côté de l'implémentation, des exemples canoniques d'usage et le comportement attendu. **Deuxième vertu** : le moindre rechargement du fichier sert de smoke-test ; une régression introduite par inadvertance se manifeste à l'instant même où la bibliothèque est chargée, pas trois cents lignes de code plus loin dans le programme appelant.

Le couplage *module + tests intégrés* sera abordé en détail au chapitre 32. Le présent chapitre se contente de signaler que les deux mécanismes ont été conçus pour se composer harmonieusement.

30.8.2. Modules qui exportent des dialectes

Un module peut, dans son corps d'exécution, enregistrer des dialectes via `[dialect:def]`. Les dialectes ainsi enregistrés vivent dans le contexte global (`ctx.dialects`) et restent accessibles via `[dialect:parse]`. Le module agit alors comme un *agent d'enregistrement* qui regroupe la déclaration de plusieurs grammaires associées.

```
# Module qui enregistre une famille de dialectes metier

[module:def Formats [export]
    [dialect:def serialNumber
        [[repeat 3 letter] "-" [repeat 5 digit] "-" [repeat 2 letter] end]
    ]
    [dialect:def isoDate
        [[repeat 4 digit] "-" [repeat 2 digit] "-" [repeat 2 digit] end]
    ]
    [dialect:def semver
        [[capture maj [some digit]] "." [capture min [some digit]] "."
         [capture pat [some digit]] end]
    ]
]

# Apres [module:load], les trois dialectes sont disponibles
[log [dialect:parse semver "1.2.3"]]
# => {ok:1, captures:{maj:"1", min:"2", pat:"3"}}
```

À noter dans cet exemple : la liste d'exports est **vide** (`[export]`) parce que le module ne déclare ni valeurs ni keywords à exposer ; son seul effet utile est d'enregistrer les trois dialectes dans le contexte global au moment de son chargement. C'est un usage un peu détourné du mécanisme — le module fonctionne ici comme un *side-effecting bundle* — mais c'est parfaitement légitime et utile en pratique pour grouper des déclarations de dialectes associés.

30.9. Limites du MVP et perspectives

Comme pour le moteur de dialectes du chapitre 29, le système de modules présenté ici est un **MVP**. Il couvre, à l'estimation des auteurs, plus de 90 % des besoins de structuration d'une base de code VVL typique. Les 10 % restants relèvent de fonctionnalités plus avancées dont la présente section dresse l'inventaire.

30.9.1. Pas de syntax « from import »

Le mécanisme actuel impose la qualification systématique : on écrit `[Math:add 1 2]`, jamais `[add 1 2]` après avoir importé `add` depuis `Math`. Cette discipline est volontaire — elle rend chaque appel auto-documentant — mais elle est parfois pesante quand un module est utilisé intensément dans un fichier.

Une syntaxe future pourrait introduire un mécanisme de **renommage local** ou d'**ouverture partielle** : `[module:open Math [add inc]]` qui rendrait `add` et `inc` accessibles non qualifiés dans le scope courant, ou `[module:alias Math M]` qui permettrait d'écrire `M:add` au lieu de `Math:add`. Ces idées sont à l'étude mais n'ont pas été retenues pour le MVP, dans le souci de maintenir une grammaire stable et un comportement prévisible.

30.9.2. Le builtin [use] reste réservé

Plusieurs langages — Lua, Rust, Lean — utilisent le mot-clé `use` pour la déclaration d'imports avec renommage. Le builtin `[use]` est, dans VVL, **réservé** pour un design futur. Il n'a pas encore de sémantique attribuée et il ne faut pas l'utiliser dans les programmes actuels, sous peine de devoir les modifier quand le design sera arrêté. La pratique recommandée est de s'en tenir à `[module:load]` et à la qualification standard `Module:name`.

30.9.3. Pas de versioning ni de dépendances déclarées

Les modules VVL n'ont pas de **numéro de version** intégré au mécanisme. Un fichier `lib/math.vvl` est, pour le runtime, juste un fichier ; deux versions différentes du même fichier ne peuvent pas coexister dans la même session. Pour les bibliothèques où la compatibilité ascendante est critique, la pratique recommandée est de **versionner par le nom du module** : un fichier `lib/math-v1.vvl` qui déclare `Math1`, un fichier `lib/math-v2.vvl` qui déclare `Math2`. C'est moins élégant que des outils comme `cargo` ou `pip`, mais c'est explicite et sans surprise.

De même, il n'existe pas de mécanisme de **déclaration de dépendances**. Un module qui utilise `Vital` doit s'assurer que `Vital` est chargé avant — soit par `[module:load "medical/vital.vvl"]` dans son propre corps, soit par convention dans le script appelant. Un futur système de dépendances pourrait formaliser ces relations, mais il introduirait une couche de complexité (résolution de version, détection de cycle, lazy loading) qui n'a pas été jugée justifiée au stade actuel.

30.9.4. Les builtins gardent leur priorité absolue

Cette propriété, déjà énoncée en 30.4, mérite d'être répétée comme une **limite** : il n'existe aucune manière, en VVL, de masquer un builtin par un keyword utilisateur ou par un export de module. Cette rigidité est délibérée : elle garantit que le sens des constructions fondamentales du langage reste stable d'un programme à l'autre. Mais elle interdit, par exemple, à un module de redéfinir `[log]` pour rediriger toutes les sorties vers un fichier de log — ce qui devra passer par un mécanisme externe (variable de configuration, plugin de redirection).

30.9.5. Pas de modules paramétrés

Le système OCaml — ou plus généralement les *functors* en théorie des modules — permet de **paramétrer** un module par un autre, créant ainsi des structures génériques réutilisables. VVL n'a pas cette capacité. Si l'on souhaite créer une variante d'un module qui dépende d'un paramètre — par exemple, un logger qui s'adresse à un canal spécifique —, on doit soit créer plusieurs modules concrets, soit passer le paramètre à chaque appel de keyword, soit utiliser un état interne mutable (comme dans le module `Log` ci-dessus). Aucune de ces solutions n'est parfaite, mais ensemble elles couvrent la quasi-totalité des cas pratiques.

30.10. Synthèse : trois échelles de réutilisation

La Partie VIII de l'ouvrage offre désormais une **trilogie** de mécanismes pour organiser le code et l'étendre. Avec les chapitres 28, 29 et 30, VVL dispose de trois échelles distinctes de structuration, chacune adaptée à un niveau de réutilisation et de partage. Le présent chapitre referme cette trilogie en posant la dernière brique.

30.10.1. Les trois échelles

Les trois échelles s'articulent comme suit. **Échelle 1 — méta-programmation locale** (chapitre 28). À ce niveau, le développeur manipule des fragments de code comme des données : `[load]`, `[evaluate]`, `[quote]`, `[dump]`. C'est l'outillage de la métamorphose ponctuelle : on transforme une expression, on l'évalue, on la sérialise. La portée est locale à un calcul ; rien n'est nommé, rien n'est partagé. **Échelle 2 — grammaires nommées** (chapitre 29). À ce niveau, on capture des structures syntaxiques sous forme de dialectes nommés. Les dialectes sont des données enregistrées sous un nom dans le contexte ; ils sont consommés par `[parse]`. **Échelle 3 — commandes encapsulées** (chapitre 30, le présent chapitre). À ce niveau, on regroupe des commandes VVL natives dans des frontières de package nommées : les modules.

Les trois échelles ne se concurrencent jamais. Un même programme peut, et le fait souvent, utiliser les trois en parallèle : utiliser `[load]` pour exécuter du code dynamique reçu par le réseau, parser ses arguments avec un dialecte enregistré, et appeler les services métier à travers des modules. C'est cette **triangulation** qui fait la puissance d'expression du langage et qui distingue VVL d'environnements où l'extensibilité reste un patchwork d'outils mal articulés.

30.10.2. Frontière formelle au cœur du langage

Les modules apportent à VVL ce qu'il manquait pour atteindre une **maturité industrielle** : une frontière formelle entre les unités de code. Cette frontière n'est pas, comme dans certains langages, un raffinement qu'on ajouterait *en plus* du langage de base — elle est au cœur de la sémantique d'évaluation. Le runtime sait précisément, à chaque résolution d'identifiant, dans quel scope chercher : builtin global, keyword utilisateur, scope local d'un module en cours d'exécution. Cette régularité est ce qui rend les modules **prévisibles** : il n'y a pas de magie, pas de découverte implicite, pas de comportement émergent.

Dans la pratique, cette régularité se traduit par une expérience de développement où les surprises sont rares. Un module compilé une fois fonctionne sans modification dans tous les contextes ; une dépendance entre modules est toujours explicite ; l'absence d'une bibliothèque attendue se manifeste

par un message clair plutôt que par un comportement silencieusement dégradé. Ces propriétés, qui peuvent sembler évidentes, sont en réalité le résultat de décisions de design soigneusement coordonnées entre la sémantique des modules, celle des keywords, et celle du contexte d'exécution.

30.10.3. Vers le chapitre 31 : valeurs scalaires typées

Le chapitre 31 ouvre un domaine apparemment plus modeste : les **valeurs scalaires typées** — `EMAIL`, `URL`, `DATE`, `UUID`. À première vue, c'est juste une commodité, une famille de constructeurs avec validation. Mais ces types vont, en réalité, formaliser un **principe** que les trois chapitres précédents n'avaient pas encore explicitement abordé : *la validation à la construction, plutôt qu'à l'usage*. Là où les dialectes valident une entrée au moment du parse et où les modules valident la cohérence d'une bibliothèque à la déclaration, les valeurs typées valident une valeur **dès qu'elle est créée**. Cette dernière brique complète l'arsenal de l'industrialisation.

Le lecteur qui aura suivi attentivement la Partie VIII jusqu'ici verra venir cette articulation. Les chapitres 28 à 30 ont posé l'**infrastructure** ; les chapitres 31 et 32 viendront poser les **garanties** que cette infrastructure rend possibles à coût raisonnable. Et le chapitre 33 ouvrira enfin la frontière externe avec les plugins signés, dernier mécanisme d'extension qui ne pouvait être présenté qu'une fois tout le reste en place.

31. Valeurs scalaires typées : EMAIL, URL, DATE, UUID

Mots-clés : *valeur typée, scalaire, validation à la construction, EMAIL, URL, DATE, UUID, mold, form, getType, round-trip, ontologie, normalisation, sérialisation typée, payload, mold-and-load*

Avec ce chapitre s'ouvre la deuxième moitié de la Partie VIII. Les trois chapitres précédents ont posé le socle infrastructurel — méta-programmation, dialectes, modules — qui rend VVL composable et organisable. À partir de maintenant, le livre se concentre sur les **garanties** que cette infrastructure rend possibles : valeurs validées à la construction (ch. 31), tests intégrés (ch. 32), plugins signés (ch. 33). Trois mécanismes qui répondent à une même question : *comment fait-on confiance, en pratique, à ce que produit et consomme un programme VVL ?*

La réponse du chapitre 31 est, en apparence, modeste. VVL introduit quatre **valeurs scalaires typées** — `EMAIL`, `URL`, `DATE`, `UUID` — et un petit contrat qui les organise : un constructeur qui valide, deux représentations (humaine et canonique), un round-trip garanti via la sérialisation. Quatre types, deux verbes, un format de littéral. On pourrait n'y voir qu'une commodité, un raffinement secondaire à côté des grands chantiers que sont l'induction de règles ou les agents autonomes. Ce serait une lecture incomplète.

Ce chapitre soutient une thèse précise : **les types scalaires sont le maillon manquant** entre l'ergonomie du langage et son ambition formelle. Tant qu'une adresse e-mail est juste *une string parmi d'autres*, le runtime n'a aucun moyen de garantir qu'elle soit valide ; toute la responsabilité incombe au programme. Avec un type `EMAIL`, la validation devient une **propriété de construction** — incontournable, vérifiable, opposable. Cette translation du *bon usage* vers la *garantie systémique* est précisément ce que les langages industriels apportent à la rigueur ; VVL ne pouvait pas se passer plus longtemps de ce mécanisme. Le chapitre va donc présenter les quatre types disponibles dans le MVP, leur contrat, leur articulation avec le reste du langage, et leurs limites.

31.1. Une lacune historique : le scalaire validé

Pour comprendre l'apport des valeurs typées, il faut d'abord mesurer précisément ce qui manquait avant elles. VVL dispose depuis le chapitre 4 d'une famille de types scalaires de base — Num, String, Boolean — auxquels le même chapitre ajoute `List` et `Json`. À partir du chapitre 12, les **ontologies** ont introduit une couche supplémentaire : des concepts définis par l'utilisateur avec des attributs typés et des règles d'inférence. Le chapitre 29 a complété l'arsenal avec les **dialectes**, capables de valider une chaîne arbitraire contre une grammaire. À première vue, cet arsenal devrait suffire à représenter et valider à peu près n'importe quoi. Pourquoi alors avoir besoin de quelque chose de plus ?

31.1.1. Le problème des deux strings indistinguables

Le diagnostic se résume en une phrase : dans un langage où *toute chaîne est une `String`*, le runtime ne peut pas faire la différence entre une chaîne qui *représente une adresse e-mail* et une chaîne qui *ne représente rien de particulier*. Considérons l'exemple ci-dessous, où deux variables de type `String` ne diffèrent que par leur contenu sémantique.

```
$email_valid    = "alice@example.com"
$texte_arbitraire = "patate au four"

[log [type-of $email_valid]]      # => "string"
[log [type-of $texte_arbitraire]] # => "string"

# Du point de vue du runtime, les deux variables sont indistinguables.
# Aucun moyen mecanique de distinguer "ceci est un email" de "ceci ne l'est pas"
# sans inspecter le contenu a chaque usage.
```

Cette indistinguibilité est l'origine de toute une famille de bugs récurrents : un programme reçoit une chaîne d'une source externe, la stocke comme adresse e-mail, et découvre seulement plusieurs étapes plus tard — au moment d'envoyer un message, par exemple — qu'elle n'a pas la forme attendue. Le diagnostic, à ce moment-là, est tardif et opaque ; l'erreur s'est déjà propagée dans plusieurs structures, parfois dans la base de données. On peut appeler ce phénomène la **dispersion temporelle de la validation** : on valide au moment de l'usage plutôt qu'au moment de la création, ce qui multiplie les sites de contrôle et les chemins de propagation.

31.1.2. Trois stratégies préexistantes — et leurs limites

Pour répondre à cette indistinguibilité, le langage offrait jusqu'ici trois stratégies, qu'on peut classer par ordre de poids croissant. Le tableau ci-dessous les résume avec leurs forces et leurs faiblesses respectives.

Stratégie	Mécanisme	Force	Faiblesse
Validation ad-hoc	un [when (! ...)] à chaque usage	aucun coût d'infrastructure	dispersion temporelle, oublis fréquents
Dialecte	[parse $s grammar] à chaque construction	validation explicite, grammaire réutilisable	la valeur reste une string non distinguable
Concept ontologique	[ontology:define Email ...] + is-a checks	sémantique riche, hiérarchie	lourd pour un scalaire ; pas de round-trip natif

La validation ad-hoc est la moins satisfaisante : elle exige une discipline parfaite du développeur, qui doit se souvenir de **chaque endroit** où la donnée entre dans le système. Le dialecte est plus systématique mais souffre d'un défaut profond : *même si la string a été validée par un dialecte, elle reste typée comme `string`*. Rien n'empêche, dans le code suivant, de la confondre avec une string non validée. Le concept ontologique est, lui, la solution la plus complète sur le plan formel — mais c'est aussi la plus coûteuse à mettre en place, et elle est mal proportionnée pour les usages simples où l'on veut juste *« cette valeur est un e-mail »* sans avoir à définir une taxonomie.

31.1.3. La quatrième stratégie : un type natif

Les valeurs scalaires typées comblent ce trou intermédiaire. Elles offrent **la validation systématique du dialecte, la distinction typologique de l'ontologie, et l'économie d'écriture d'un littéral**. Là où un dialecte produit *une string validée*, le constructeur `[EMAIL ...]` produit *une valeur de type `email`*, distincte de toutes les autres au regard du runtime. Là où une ontologie exige une déclaration

`ontology:define` et des règles `is-a`, le type `EMAIL` est *immédiatement disponible*, sans aucune déclaration préalable.

L'analogie qui aide à fixer la position des types scalaires est celle du **type `Datetime`** dans les langages classiques : Python, Java, C# offrent tous un type de date-heure natif, séparé de la chaîne, avec validation à la construction et sérialisation canonique. Personne ne songerait à représenter une date par une `String` dans ces langages modernes — l'usage du type natif est la convention industrielle. Les valeurs typées de VVL généralisent cette discipline à quatre familles : adresses e-mail, URL, dates, identifiants uniques. Et le mécanisme est suffisamment uniforme pour servir de **patron** au cas où le langage devrait, plus tard, en accueillir d'autres.

31.2. Le contrat des valeurs typées

Les quatre types — `EMAIL`, `URL`, `DATE`, `UUID` — partagent un contrat unique. Cette uniformité est un choix de design important : un développeur qui apprend à utiliser `EMAIL` connaît, par construction, l'API des trois autres. La figure ci-dessous donne la vue d'ensemble du contrat ; les sous-sections qui suivent en examinent chacun des éléments.

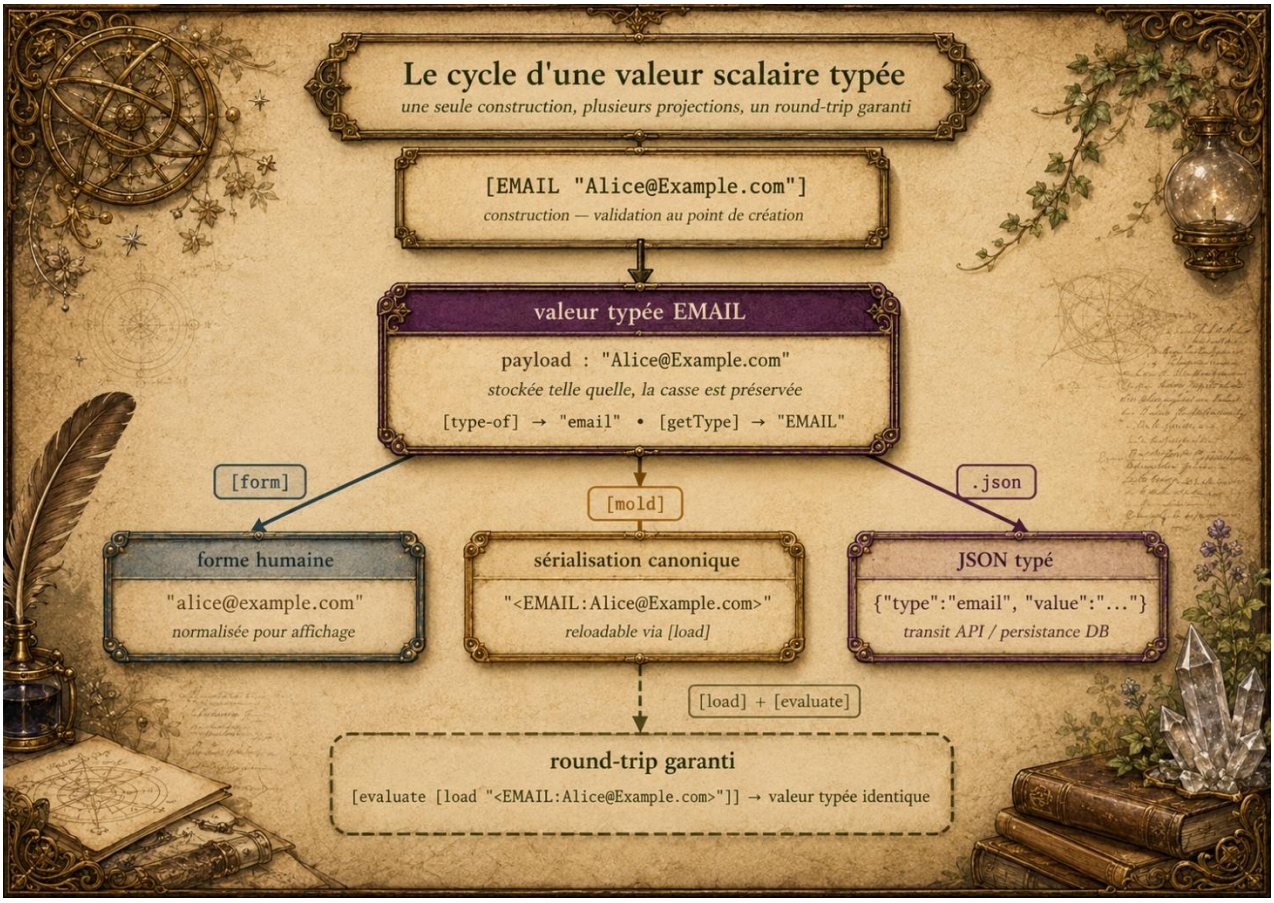


Figure 31.1 — *Le cycle de vie d'une valeur scalaire typée. Une seule construction (en haut) ; une valeur centrale immutable ; trois projections (form pour l'humain, mold pour le round-trip, .json pour la persistance) ; un round-trip garanti via [load] et [evaluate].*

31.2.1. La construction : [TYPE payload]

Toute valeur typée se crée par un constructeur qui partage la même signature : `[TYPE payload]`, où `TYPE` est le tag du type — toujours en majuscules par convention — et `payload` est la valeur à valider.

Le constructeur **valide à la construction** : si le payload ne respecte pas la forme attendue, le constructeur lève une erreur claire, qui contient le type cible et la raison du refus.

```
# Construction d'un EMAIL valide

$mail = [EMAIL "alice@example.com"]
[log [type-of $mail]]             # => "email"

# Construction d'un EMAIL invalide

$mauvais = [EMAIL "pas du tout un email"]
# => erreur : EMAIL invalide ("pas du tout un email")
```

Le payload est, dans les quatre cas du MVP, une **chaîne de caractères** — c'est le format qui couvre tous les cas où une valeur typée est ingérée depuis l'extérieur (formulaire, API, base de données). Les détails de la validation diffèrent pour chaque type et sont l'objet des sections 31.3 à 31.6. Mais la **signature** est invariante.

Une propriété qui peut surprendre : le payload est conservé **tel quel** dans la valeur typée, sans normalisation au stockage. Si l'utilisateur a écrit `[EMAIL "Alice@Example.com"]` avec une majuscule, la valeur typée garde cette casse. La normalisation n'intervient qu'au moment d'une **projection** spécifique — typiquement `[form]`, qui retourne en minuscules pour l'affichage. Cette séparation entre *stockage fidèle* et *projection normalisée* préserve l'intention de l'utilisateur tout en offrant une lecture standardisée.

31.2.2. La projection humaine : [form]

`[form $valeur]` retourne la **représentation pour humain** de la valeur. Pour les valeurs typées, cette représentation est normalisée selon la convention du type. L'idée derrière ce verbe : *« montre-moi cette valeur comme tu la montrerais à un utilisateur final »*.

```
# [form] retourne la représentation humaine, normalisee

$mail = [EMAIL "Alice@Example.com"]
[log [form $mail]]                # => alice@example.com

$uuid = [UUID "550E8400-E29B-41D4-A716-446655440000"]
[log [form $uuid]]                # => 550e8400-e29b-41d4-a716-446655440000
```

La normalisation est, dans le MVP, déterminée par chaque type. Les sections 31.3 à 31.6 précisent les règles pour chacun ; par convention, ces règles sont **conservatrices** (elles ne modifient que ce qui est strictement nécessaire à la lisibilité) et **idempotentes** (appliquer `[form]` deux fois donne le même résultat que l'appliquer une seule fois).

31.2.3. La projection canonique : [mold]

Là où `[form]` produit une chaîne pour humain, `[mold]` produit une chaîne **pour le langage** : la représentation canonique qui, repassée à `[load]` puis `[evaluate]`, reconstruit la valeur typée à l'identique. Cette représentation utilise un format strict :

```
001  $mail = [EMAIL "Alice@Example.com"]
002  [log [mold $mail]]                # => <EMAIL:Alice@Example.com>
003
004  $date = [DATE "2026-05-24"]
005  [log [mold $date]]                # => <DATE:2026-05-24>
006
007  $url  = [URL "https://prosuite.com/path?q=1"]
008  [log [mold $url]]                 # => URL:https://prosuite.com/path?q=1
```

Le format `<TYPE:payload>` est le **littéral préfixé** des valeurs typées — il sera examiné plus en détail en 31.7. Trois propriétés notables. **Première propriété** : `[mold]` préserve la casse et la forme exacte du payload stocké, à la différence de `[form]` qui peut normaliser. **Deuxième propriété** : le format est **non ambigu** — un parseur ne peut pas confondre un `<EMAIL:...>` avec une autre construction du langage. **Troisième propriété** : la représentation est **reloadable**, c'est-à-dire que `[load "<EMAIL:Alice@Example.com>"]` produit la forme VVL exacte qui, évaluée, reconstruit la valeur initiale.

31.2.4. La projection JSON : .json

Une troisième projection est disponible via l'attribut `.json` de la valeur typée. Il s'agit, formellement, d'un `VVL_Json` qui décrit la valeur de façon structurée :

```
001  $mail = [EMAIL "Alice@Example.com"]
002  [log $mail.json]
003  # => {"type":"email", "value":"Alice@Example.com"}
```

Ce format est conçu pour le **transit API** et la **persistance en base de données**. Il préserve l'information du type (sous-champ `type` en minuscules) et le payload original (sous-champ `value`, à la casse intacte). Une couche de persistance qui voudrait stocker la valeur en JSON peut simplement écrire `$mail.json` dans la base ; à la lecture, un constructeur peut reconstruire la valeur à partir de ces deux champs.

31.2.5. Le round-trip — la garantie fondamentale

La propriété qui résume le contrat est le **round-trip garanti**. Pour toute valeur typée `$v`, on dispose de la garantie suivante :

```
# Le round-trip via mold + load + evaluate

$v_original = [EMAIL "Alice@Example.com"]
$texte      = [mold $v_original]    # => "<EMAIL:Alice@Example.com>"
$forme      = [load $texte]         # => forme VVL inerte
$v_restored = [evaluate $forme]     # => valeur typee EMAIL

# Verification : les deux valeurs sont equivalentes
[log ($v_restored == $v_original)] # => 1 (true)
```

Cette garantie est **fondamentale** parce qu'elle ferme la boucle de la sérialisation. Un programme peut, sans peur de perte d'information, écrire une valeur typée dans un journal, l'envoyer par le réseau, la persister dans une base, et la retrouver plus tard exactement équivalente à l'originale. Le round-trip est ce qui rend les valeurs typées **transportables** au-delà d'une seule session de runtime.

Le tableau ci-dessous résume le contrat partagé par les quatre types, indépendamment de leur sémantique métier propre.

Opération	Signature	Retour
Construction	[TYPE payload]	la valeur typée, ou erreur si invalide
Forme humaine	[form $v]	string normalisée pour affichage
Sérialisation	[mold $v]	string canonique <TYPE:payload>
JSON typé	$v.json	{type, value} en VVL_Json
Nom de type	[type-of $v]	string minuscule ("email")
Tag de type	[getType $v]	string majuscule ("EMAIL")

31.3. EMAIL : l'adresse de courrier électronique

Le premier type — et le plus courant en pratique — est `EMAIL`. Une valeur de type `EMAIL` représente une adresse de courrier électronique validée à la construction selon une grammaire dérivée de la RFC 5322, simplifiée pour les usages les plus fréquents.

31.3.1. Validation

La validation à la construction vérifie que la chaîne reçue suit la structure générale `local@domain`, où la partie locale et le domaine respectent chacun des contraintes raisonnables : présence d'un caractère `@` unique en position non extrême, partie locale composée de caractères alphanumériques et de quelques caractères spéciaux usuels (`.`, `-`, `_`, `+`), domaine composé de sous-domaines séparés par des points avec un TLD final. Le moteur n'implémente **pas** la RFC 5322 dans son intégralité — qui autorise des constructions exotiques rarement utilisées en pratique — mais il couvre largement les usages réels.

```
# Adresses valides

[EMAIL "alice@example.com"]
[EMAIL "first.last@university.edu"]
[EMAIL "user+tag@mail.example.co.uk"]
[EMAIL "support-team@company-name.org"]

# Adresses refusees

[EMAIL ""]                          # => erreur : payload vide
[EMAIL "no-at-sign"]                # => erreur : aucun @
[EMAIL "@example.com"]              # => erreur : partie locale vide
[EMAIL "alice@"]                    # => erreur : domaine vide
```

31.3.2. Le couple form / mold : la spécificité d'EMAIL

EMAIL est le seul type, dans le MVP, où `[form]` applique une **normalisation visible** : la projection humaine est toujours retournée en minuscules. Cette convention est intentionnelle — les adresses e-mail sont, dans la majorité des spécifications de protocole, considérées comme **insensibles à la casse** dans la partie domaine, et la convention industrielle veut qu'on affiche la partie locale en minuscules aussi pour éviter les confusions visuelles.

```
# La normalisation par [form]

$mail = [EMAIL "Alice.Dupont@MyMail.COM"]

[log [form $mail]]                  # => alice.dupont@mymail.com
[log [mold $mail]]                  # => <EMAIL:Alice.Dupont@MyMail.COM>
[log $mail.json.value]              # => Alice.Dupont@MyMail.COM
```

Trois sorties distinctes, donc, pour la même valeur. `[form]` produit une chaîne *présentable* — c'est celle qu'on affichera dans une UI. `[mold]` produit une chaîne *fidèle* — c'est celle qu'on archivera pour pouvoir reconstruire la valeur initiale à l'identique. Le `.json.value` est lui aussi fidèle au stockage. Cette triple disponibilité résout définitivement le dilemme entre *normaliser* (pour la commodité) et *préserver* (pour la traçabilité).

31.3.3. Pourquoi pas une validation RFC complète

Une question légitime se pose : pourquoi le MVP n'implémente-t-il pas la RFC 5322 dans son intégralité ? La réponse tient à un constat empirique : les adresses véritablement conformes à la RFC mais exclues par la grammaire simplifiée sont **extrêmement rares en pratique**. Des cas comme `"john.doe"@example.com` (avec guillemets autour de la partie locale) ou `user@[192.0.2.1]` (adresse IP littérale dans le domaine) existent dans les spécifications mais sont presque jamais rencontrés dans la vie réelle d'un programme. À l'inverse, valider la RFC complète exige un parseur conséquent dont les bugs rares peuvent silencieusement laisser passer des adresses non conformes ailleurs. Le MVP préfère une grammaire restrictive bien documentée à une grammaire permissive opaque.

Pour les rares cas où une validation RFC complète serait nécessaire, la voie recommandée est un **plugin Python** (chapitre 33) qui utilise une bibliothèque éprouvée comme `email-validator`. Cette stratégie respecte la division du travail entre VVL natif (raisonnable, lisible, audité) et code externe signé (complet, performant, mais plus opaque).

31.4. URL : la ressource web

Le deuxième type, `URL`, représente une URL (*Uniform Resource Locator*). Comme pour EMAIL, la validation à la construction vérifie une grammaire simplifiée mais largement suffisante pour les usages courants.

31.4.1. Schemas acceptés : http et https uniquement

Une décision importante du MVP : seuls les schémas `http` et `https` sont acceptés. Les schémas exotiques (`ftp`, `file`, `mailto`, `data`, `git`, etc.) ne passent pas la validation. Cette restriction reflète l'usage dominant — la quasi-totalité des URLs manipulées dans les programmes contemporains sont des URLs web — et évite la complexité des grammaires multi-schémas qui posent souvent plus de problèmes qu'elles n'en résolvent.

```
001  # URLs acceptees
002
003  [URL "https://prosuite.com/path?q=1"]
004  [URL "http://localhost:8080/api/v1"]
005  [URL "https://docs.example.org/page#section"]
006
007  # URLs refusees
008
009  [URL "ftp://files.example.com"]     # => erreur : schema non supporte
010  [URL "file:///home/user/doc.txt"]   # => erreur : schema non supporte
011  [URL "//example.com/path"]          # => erreur : schema absent
012  [URL "just a string"]               # => erreur : pas une URL
```

L'utilisateur qui voudrait stocker une URL avec un schéma non supporté a deux options. Soit il accepte que la valeur reste une simple `String` non typée — la solution pragmatique pour les schémas vraiment marginaux. Soit il définit son propre type via un **plugin** (chapitre 33). Une troisième voie, ouverte dans la roadmap mais absente du MVP, serait l'introduction d'un type `URI` plus permissif qui couvrirait les schémas génériques.

31.4.2. Le contenu de l'URL est intégralement préservé

Contrairement à EMAIL, le type `URL` n'applique **pas de normalisation visible** par `[form]` — la projection humaine retourne le payload tel quel. Cette différence reflète le fait que les URLs sont sensibles à la casse dans plusieurs de leurs composants (la partie de chemin, les paramètres de requête, les fragments), et qu'une normalisation automatique introduirait plus d'erreurs qu'elle n'en résoudrait.

```
$url = [URL "https://API.Example.com/Users/Alice?ID=42"]

[log [form $url]]
# => https://API.Example.com/Users/Alice?ID=42

[log [mold $url]]
# => <URL:https://API.Example.com/Users/Alice?ID=42>
```

Le payload est préservé strictement. Si l'utilisateur a besoin d'une normalisation spécifique — par exemple convertir le scheme en minuscules tout en préservant le path —, il doit l'effectuer explicitement avant ou après l'usage de `[URL]`. Cette décision réduit la **surprise** d'utilisation : un utilisateur sait que ce qu'il a écrit est ce qu'il obtiendra.

31.5. DATE : la date civile

Le troisième type, `DATE`, représente une **date civile** au format ISO 8601 canonique `YYYY-MM-DD`. Quatre chiffres pour l'année, un tiret, deux chiffres pour le mois, un tiret, deux chiffres pour le jour.

31.5.1. Format strict : YYYY-MM-DD

Le format est imposé pour deux raisons. **Première raison** : ISO 8601 est la convention internationale par défaut pour la persistance et l'échange de dates, adoptée par presque tous les protocoles et bases de données modernes. **Deuxième raison** : ce format est sans ambiguïté de tri ; l'ordre lexicographique des chaînes ISO est l'ordre chronologique. Cette propriété est précieuse pour les opérations de comparaison et de tri sans avoir à parser la date.

```
# Dates valides

[DATE "2026-05-24"]
[DATE "1999-12-31"]
[DATE "2000-02-29"]                # 2000 etait bissextile

# Dates refusees

[DATE "24/05/2026"]                # => erreur : format non ISO
[DATE "2026-13-01"]                # => erreur : mois invalide
[DATE "2025-02-29"]                # => erreur : 2025 non bissextile
[DATE "2026-5-24"]                 # => erreur : mois sur 1 chiffre
```

La validation inclut la **cohérence calendaire** : un 30 février est rejeté, un 31 avril aussi, et les années bissextiles sont prises en compte correctement. Cette vérification est ce qui distingue une vraie validation d'une simple expression régulière — un parseur naïf accepterait `2026-02-30` parce qu'il vérifie seulement la structure, pas la sémantique.

31.5.2. Limites du MVP : pas de temps ni de fuseau

Le type `DATE` représente **uniquement** une date civile : pas d'heure, pas de fuseau horaire, pas de date-heure combinée. Cette modestie volontaire évite les pièges classiques du fuseau horaire (qui méritent leur propre type, leur propre validation, et leurs propres règles de normalisation) tout en couvrant les cas où la date civile suffit — date de naissance, date d'échéance, date de publication, date d'événement de calendrier civil.

Pour les besoins de **date-heure** ou **timestamp**, le MVP recommande, dans l'immédiat, de stocker la valeur comme une `String` ISO 8601 étendue (`2026-05-24T14:30:00Z`) ou comme un `Num` (timestamp Unix). Un type `DATETIME` natif est inscrit à la roadmap et sera ajouté dans une version ultérieure du runtime, avec validation du format ISO étendu et gestion explicite des fuseaux.

31.6. UUID : l'identifiant universellement unique

Le quatrième et dernier type du MVP est `UUID`, qui représente un *Universally Unique Identifier* selon la RFC 4122. Format canonique : huit chiffres hexadécimaux, tiret, quatre, tiret, quatre, tiret, quatre, tiret, douze. Soit trente-deux chiffres hex au total, séparés par quatre tirets, pour un total de 36 caractères.

31.6.1. Format et normalisation

Le constructeur accepte le format avec tirets dans la casse libre ; il **normalise systématiquement en minuscules** au moment de l'accès via `[form]`.

```
# UUIDs valides

[UUID "550e8400-e29b-41d4-a716-446655440000"]
[UUID "550E8400-E29B-41D4-A716-446655440000"]
# (les deux donnent la meme valeur logique)

# UUIDs refuses

[UUID "550e8400e29b41d4a716446655440000"]   # => erreur : tirets manquants
[UUID "550e8400-e29b-41d4-a716"]            # => erreur : trop court
[UUID "ZZZe8400-e29b-41d4-a716-446655440000"]
                                            # => erreur : caracteres non hex
```

La normalisation en minuscules suit la convention RFC 4122, qui recommande la forme minuscule pour la sérialisation canonique. Comme pour EMAIL, le payload est **stocké tel quel** dans la valeur ; ce sont uniquement `[form]` et certains usages qui appliquent la normalisation.

```
$id = [UUID "550E8400-E29B-41D4-A716-446655440000"]

[log [form $id]]
# => 550e8400-e29b-41d4-a716-446655440000
```

```
005
006 [log [mold $id]]
007 # => <UUID:550e8400-e29b-41d4-a716-446655440000>
```

À la différence d'EMAIL, où `[mold]` préserve la casse stockée, pour UUID, `[mold]` retourne **également** la forme minuscule normalisée. C'est un choix volontaire et différent : pour UUID, l'identité **logique** d'un identifiant ne dépend pas de sa casse, et il serait artificiel de préserver une casse arbitraire qui n'a aucune signification métier. Pour EMAIL au contraire, la casse stockée peut être significative (politesse vis-à-vis de l'utilisateur, traçabilité) et mérite d'être conservée dans le round-trip.

31.6.2. Génération vs construction

Une dernière précision utile : le constructeur `[UUID payload]` **valide** une chaîne reçue, il ne **génère** pas un UUID nouveau. La génération d'identifiants uniques relève d'un autre builtin — typiquement `[uuid:new]` ou `[uuid:v4]`, dont la sémantique est génératrice, pas validative. Cette séparation entre construction (un UUID que je *reçois*) et génération (un UUID que je *crée*) suit le pattern général de VVL : un constructeur valide, un générateur produit, et les deux ont des noms différents pour qu'aucune ambiguïté ne s'installe.

```
001 # Distinction generation / construction
002
003 $id_genere = [uuid:new]                        # genere un UUID v4 frais
004 # $id_genere est deja une valeur typee UUID
005
006 $id_recu   = [UUID "550e8400-e29b-41d4-a716-446655440000"]
007 # $id_recu construit une valeur typee a partir d'une chaine donnee
008
009 # Les deux valeurs ont le meme type ; seul leur origine differe.
010 [log [type-of $id_genere]]                    # => "uuid"
011 [log [type-of $id_recu]]                      # => "uuid"
```

Note de doctrine : `[uuid:new]` n'est pas le seul builtin générateur disponible. La famille `Date:today`, `Time:now`, et d'autres helpers générateurs ont vocation à compléter chacun des types scalaires — d'abord pour les besoins fréquents (date du jour, UUID frais), puis pour les besoins plus marginaux (URLs canoniques de ressources internes). Ces helpers générateurs ne font pas partie du présent chapitre, qui se concentre sur le contrat des **constructeurs validants** ; ils seront évoqués dans la documentation de référence du runtime.

31.7. La grammaire <TYPE:payload> — le littéral préfixé

Le format `<TYPE:payload>` rencontré dans les retours de `[mold]` est, plus qu'un format de sortie, **un littéral du langage**. Une chaîne `"<EMAIL:alice@example.com>"` chargée par `[load]` produit une forme VVL inerte qui, une fois évaluée, reconstruit la valeur typée. Cette propriété est ce qui rend les valeurs typées **persistables** dans des contextes textuels arbitraires — fichiers de configuration, sorties de logs, messages réseau.

31.7.1. Anatomie du littéral

Le littéral préfixé suit une grammaire stricte. Le chevron ouvrant `<`, suivi du tag de type en majuscules (`EMAIL`, `URL`, `DATE`, `UUID`), suivi de deux-points `:`, suivi du payload original (sans guillemets, jusqu'au chevron fermant), suivi du chevron fermant `>`. Aucun espace n'est toléré autour du tag ou des deux-points. Le payload peut contenir n'importe quel caractère sauf le chevron fermant, ce qui couvre largement les besoins.

Type	Littéral préfixé canonique	Reconstruction
EMAIL	<EMAIL:Alice@Example.com>	[evaluate [load "<EMAIL:Alice@Example.com>"]]
URL	<URL:https://prosuite.com/path?q=1>	[evaluate [load "<URL:https://prosuite.com/path?q=1>"]]
DATE	<DATE:2026-05-24>	[evaluate [load "<DATE:2026-05-24>"]]
UUID	<UUID:550e8400-e29b-41d4-a716-446655440000>	[evaluate [load "<UUID:...>"]]

31.7.2. Pourquoi un préfixe explicite

Une question de design mérite explication : pourquoi le langage exige-t-il un préfixe `<EMAIL:...>` plutôt que de **deviner** le type à partir du contenu ? VVL aurait pu, par exemple, considérer toute chaîne contenant un `@` central comme une adresse e-mail, et la promouvoir automatiquement en valeur typée. Cette option a été délibérément écartée.

La raison fondamentale est la **prévisibilité**. Une inférence implicite serait une source de surprises : un identifiant interne qui contient un `@`, un mot dans une langue exotique, une chaîne arbitraire — autant de candidats à des promotions accidentelles qui changeraient la sémantique du programme sans raison apparente. À l'inverse, la convention du préfixe explicite garantit que la valeur typée est *toujours* le résultat d'une **intention déclarée** : soit un constructeur `[EMAIL ...]`, soit un littéral `<EMAIL:...>` reconnu lors du parsing. Aucune ambiguïté possible.

Cette discipline a un coût modeste à l'écriture — il faut écrire `<EMAIL:...>` au lieu de simplement `"alice@example.com"` quand on veut une valeur typée — mais elle protège contre toute une classe de bugs subtils où une chaîne ordinaire serait silencieusement traitée comme une adresse e-mail et inversement.

31.8. type-of et getType : deux noms pour deux usages

Le chapitre 28 a introduit `[type-of]` comme primitive d'introspection retournant le nom de type en minuscules. Avec l'arrivée des valeurs typées, un second builtin entre en scène : `[getType]`, qui retourne le tag en majuscules. La distinction entre les deux mérite d'être éclaircie.

```
001  $mail = [EMAIL "alice@example.com"]
002  $num  = 42
003  $str  = "hello"
004
005  # type-of : nom de type stable, en minuscules
006  [log [type-of $mail]]            # => "email"
007  [log [type-of $num]]             # => "num"
008  [log [type-of $str]]             # => "string"
009
```

```
# getType : tag formel en majuscules
[log [getType $mail]]            # => "EMAIL"
[log [getType $num]]             # => "NUM"
[log [getType $str]]             # => "STRING"
```

31.8.1. Quand utiliser quoi

`[type-of]` est le **builtin de dispatch** : on l'utilise pour brancher selon le type d'une valeur, dans une logique de programme. Son retour minuscule est conçu pour les comparaisons exactes et les patterns de `[match]`. La forme minuscule s'aligne sur les conventions de typage de la plupart des langages (Python, Ruby, JavaScript) qui désignent leurs types par des noms en minuscules.

`[getType]` est le **builtin de tag** : on l'utilise pour obtenir l'étiquette formelle d'une valeur typée, typiquement pour la sérialiser, l'afficher dans une UI de débogage, ou la construire dans un message JSON typé. Son retour majuscule est cohérent avec le tag utilisé dans le littéral préfixé (`<EMAIL:...>`) et avec la convention de nommage des constructeurs (`[EMAIL ...]`).

```
# Pattern de dispatch (type-of)

[match [type-of $valeur]
    "email" `[log "C'est un email : " [form $valeur]]
    "url"   `[log "C'est une URL : " [form $valeur]]
    "date"  `[log "C'est une date : " [form $valeur]]
    "uuid"  `[log "C'est un UUID : " [form $valeur]]
    _       `[log "Type inconnu : " [type-of $valeur]]
]

# Pattern de tag (getType)

$tag = [getType $valeur]
$serialized = [concat "<" $tag ":" [form $valeur] ">"]
# Manuelle equivalent a [mold $valeur]
```

La distinction se résume en une phrase : **`type-of` pour décider, `getType` pour étiqueter**. Dans la pratique, `type-of` est le plus utilisé des deux — c'est l'outil quotidien de dispatch. `getType` est plus pointu et n'apparaît que dans les codes qui construisent eux-mêmes des littéraux préfixés ou qui interagissent avec des systèmes externes attendant un tag majuscule.

31.9. Valeurs typées vs ontologie vs dialecte

Trois mécanismes du langage offrent maintenant des moyens de représenter et valider des données structurées : les **valeurs scalaires typées** introduites dans le présent chapitre, les **concepts ontologiques** du chapitre 12, et les **dialectes** du chapitre 29. Ces trois mécanismes peuvent sembler concurrents ; ils sont en réalité complémentaires et adressent des problèmes différents. Cette section précise la division du travail.

Critère	Valeur typée	Concept ontologique	Dialecte
Granularité	scalaire (un seul atome)	composite (attributs multiples)	syntaxe consommable
Validation	à la construction du constructeur	par règles ontology:rule à l'inférence	lors de l'appel à [parse]
Hiérarchie	aucune (les 4 types sont parallèles)	taxonomie possible (is-a, part-of)	aucune (chaque dialecte est isolé)
Round-trip	natif (mold / load)	via sérialisation JSON manuelle	la grammaire ne capture pas la valeur typée
Cas d'usage	champs élémentaires (e-mail, date, ID)	objets métier (Patient, Diagnostic)	parsing de DSL et de formats

31.9.1. La division du travail

La répartition optimale, dans la pratique d'un programme bien architecturé, est la suivante. Les **valeurs typées** s'utilisent pour les **champs élémentaires** : une adresse e-mail, une URL, une date de naissance, un UUID d'objet métier. À ce niveau, on veut juste *« cette valeur est-elle bien ce qu'elle prétend être ? »* et le mécanisme natif est parfaitement adapté.

Les **concepts ontologiques** prennent le relais pour les **objets métier** composites : un Patient n'est pas un scalaire, c'est une structure avec un nom, une date de naissance, une liste de diagnostics, etc. C'est pour cela qu'il devient un concept ontologique avec des attributs typés. Et certains de ces attributs peuvent eux-mêmes être des valeurs scalaires typées : la date de naissance est un `DATE`, l'identifiant est un `UUID`. La composition est naturelle.

Les **dialectes** se déploient lorsqu'il faut **parser une entrée textuelle** dont la structure ne correspond ni à un scalaire simple ni à un objet ontologique connu : une notation métier propriétaire, une commande utilisateur, un mini-DSL embarqué. Une fois l'entrée parsée, ses captures peuvent **alimenter** des valeurs typées et des concepts ontologiques, fermant ainsi la boucle d'ingestion.

```
# Exemple combine : parser une fiche patient, en remplir un concept

[dialect:def fichePatient
    [[capture nom    [some letter]] " "
     [capture mail   [some any]]    " "
     [capture naiss  [some any]]    end]
]

$saisie = [dialect:parse fichePatient "Dupont alice@ex.com 1985-03-12"]

# On promeut chaque capture en valeur typee, puis en concept ontologique
[ontology:define Patient [name email birth]]

[ontology:assert
    [Patient
        [name  $saisie.captures.nom]
        [email [EMAIL $saisie.captures.mail]]
        [birth [DATE  $saisie.captures.naiss]]
    ]
]
```

Cet exemple illustre la **chaîne d'ingestion canonique** : un dialecte décompose, des valeurs typées valident scalaires, un concept ontologique structure le tout. Chaque mécanisme joue son rôle propre ; aucun ne se substitue à un autre. C'est cette **stratification claire** qui distingue VVL d'environnements où la même fonctionnalité est dispersée entre plusieurs outils mal articulés.

31.10. Cas d'usage

Pour fixer définitivement la valeur pratique des valeurs typées, déroulons trois cas d'usage qui montrent leur articulation avec le reste du langage. Comme dans les chapitres précédents, ces cas ne sont pas hypothétiques — ils reflètent des patterns qui apparaissent réellement dans le code de production VVL.

31.10.1. Validation d'un formulaire d'inscription

Premier cas, le plus fréquent : ingérer un formulaire utilisateur dont les champs doivent être validés avant stockage. Sans valeurs typées, chaque champ exigerait une validation manuelle dispersée. Avec les valeurs typées, la validation est **systématique** et **lisible**.

```
# Reception d'un formulaire HTTP en JSON

$form = [json-decode $request_body]

$inscription = [attempt [object
    [email     [EMAIL $form.email]]
    [website   [URL   $form.website]]
    [birth     [DATE  $form.birth]]
    [name      $form.name]
]]

[when (! $inscription.ok)
    `[do [respond-with 400 [object
            [error "validation_failed"]
            [details $inscription.error.message]
        ]]
         [return]]
]

# A ce stade, on sait que les quatre champs sont bien formes.
[db:insert "users" $inscription.value]
```

Quatre champs validés en quatre lignes, avec interception centralisée des erreurs via `[attempt]` (chapitre 28). Si l'un des champs est mal formé, le constructeur lève une erreur, l'`attempt` la capture, et le code renvoie une réponse 400 propre. Si tous les champs sont valides, l'objet est inséré tel quel — les valeurs typées y sont stockées en JSON typé via leur `.json`, ce qui permettra de les reconstruire à la lecture.

31.10.2. Filtrage d'une liste d'identifiants

Deuxième cas, plus subtil : extraire d'une liste de chaînes hétérogènes celles qui sont des UUIDs valides. La présence du type natif rend ce filtrage trivialement écrit.

```
# Une liste de chaines hétérogenes recue d'une source externe

$candidats = `[
    "550e8400-e29b-41d4-a716-446655440000"
    "not-a-uuid"
    "6ba7b810-9dad-11d1-80b4-00c04fd430c8"
    ""
    "7c9e6679-7425-40de-944b-e07fc1f90ae7"
]

# Filtrer par essai-erreur via [attempt]
$uuids_valides = [collect
    [foreach $s $candidats
        `[do
           $r = [attempt [UUID $s]]
           [when $r.ok `[keep $r.value]]
         ]
    ]
]

[log [length $uuids_valides]]     # => 3
```

Le couple `[attempt]` (qui transforme une erreur en statut JSON) et `[collect]` (qui accumule sélectivement) permet d'écrire un filtrage type-aware en quelques lignes. La même logique sans les valeurs typées aurait exigé une expression régulière externe ou un dialecte ad-hoc, avec à chaque fois la redondance entre *valider la forme* et *promouvoir au type*.

31.10.3. Persistance avec round-trip garanti

Troisième cas, qui met en valeur le round-trip : persister une valeur typée dans un format de stockage textuel, puis la reconstruire à l'identique à la lecture.

```
# Ecriture : on serialise la valeur typee en chaine canonique

$user_id = [UUID "550e8400-e29b-41d4-a716-446655440000"]
$user_email = [EMAIL "alice@example.com"]

[file:write "user.txt"
    [concat [mold $user_id] "\n" [mold $user_email]]
]
# Le fichier contient :
#   <UUID:550e8400-e29b-41d4-a716-446655440000>
#   <EMAIL:alice@example.com>

```

```
# Lecture : on reconstruit les valeurs typees

$lignes = [split [file:read "user.txt"] "\n"]

$loaded_id    = [evaluate [load [getValue $lignes 0]]]
$loaded_email = [evaluate [load [getValue $lignes 1]]]

[log [type-of $loaded_id]]        # => "uuid"
[log [type-of $loaded_email]]     # => "email"
[log ($loaded_id == $user_id)]    # => 1
```

Le round-trip est complet : aucune perte d'information entre l'écriture et la lecture, et les types sont restaurés à l'identique. Cette propriété est, à elle seule, ce qui justifie l'existence du format `<TYPE:payload>` — un format textuel sans ce garantie de réversibilité serait moins utile dans la pratique de la persistance.

31.11. Limites du MVP et perspectives

Comme pour les chapitres précédents, le système de valeurs typées présenté ici est un **MVP** : il couvre les quatre types les plus utilisés en pratique et expose un contrat uniforme suffisant pour la grande majorité des cas. Les limites listées ci-dessous sont autant de directions d'évolution pour les versions ultérieures.

31.11.1. Quatre types seulement

Le MVP n'inclut **que** `EMAIL`, `URL`, `DATE`, `UUID`. Des types évidemment utiles manquent à l'appel : `DATETIME` (date-heure avec fuseau), `TIME` (heure seule), `DURATION` (intervalle temporel), `PHONE` (numéro de téléphone), `IBAN` (compte bancaire international), `BIC`, `CURRENCY`, `COUNTRY_CODE`, `LANGUAGE_TAG`, `IPV4`, `IPV6`. Chacun mériterait son chapitre, ou au moins sa section. Aucun n'est dans le MVP.

La raison de cette restriction n'est pas un manque d'ambition : c'est une décision de **stabilisation**. Avant d'ajouter de nouveaux types, le runtime doit d'abord prouver, sur les quatre premiers, que le contrat partagé fonctionne en pratique et qu'aucune subtilité de design n'a été manquée. Une fois la stabilité atteinte — typiquement après plusieurs versions de runtime —, l'extension à une famille plus large pourra se faire avec confiance. Le rythme prévu est : un nouveau type par version mineure, après validation par la communauté.

31.11.2. Pas de types personnalisés utilisateur

Une limite plus profonde : le mécanisme des valeurs typées est **réservé au runtime**. Un utilisateur ne peut pas déclarer son propre type scalaire validé — par exemple `CODE_POSTAL_FR` qui validerait cinq chiffres conformes à la nomenclature INSEE. Pour de tels besoins, la voie recommandée est triple : *(i)* un **dialecte** qui valide la chaîne, *(ii)* un **concept ontologique** si la classification métier le justifie, *(iii)* éventuellement un **plugin** (chapitre 33) si une intégration native est indispensable.

Cette restriction est une conséquence directe de la décision précédente : tant que le mécanisme natif n'est pas stabilisé, ouvrir l'extension utilisateur produirait des collisions de noms et des incompatibilités difficiles à gérer. Une fois la stabilité atteinte, un mécanisme `[type:def MY_TYPE ...]` pourrait être envisagé, mais il n'est pas dans le MVP.

31.11.3. Pas de paramétrage de validation

La validation est, pour chaque type, **codée en dur** dans le runtime. Un utilisateur qui voudrait, par exemple, accepter seulement les e-mails d'un domaine particulier (`*@mycompany.com`) ne peut pas le faire à travers le type `EMAIL` — il devra valider d'abord avec `EMAIL` (forme générale), puis vérifier le domaine séparément. Cette absence de paramétrage est, là encore, une simplification volontaire du MVP : permettre des grammaires personnalisées sur les types natifs introduirait une complexité d'implémentation considérable pour un bénéfice marginal.

Pour les besoins de validation fine au-delà de la grammaire générale, la composition reste la voie naturelle :

```
# Validation de domaine specifique apres EMAIL

[keyword company-email `[do
    $e = [EMAIL @1]                 # forme generale
    $domain = [getValue [split [form $e] "@"] 1]
    [when ($domain != "mycompany.com")
        `[error "EMAIL hors entreprise : " $domain]
    ]
    $e                              # retour de la valeur typee
]]

[company-email "alice@mycompany.com"]   #
[company-email "bob@external.com"]      # => erreur
```

Ce pattern — un keyword qui enveloppe un constructeur de type natif avec des validations supplémentaires — couvre, en pratique, la quasi-totalité des cas où une validation paramétrée serait souhaitable. Et il a l'avantage d'être **lisible** : un lecteur du code voit immédiatement les deux étapes de validation.

31.12. Synthèse : la validation au point de création

Le présent chapitre a ajouté à VVL un mécanisme dont la **valeur formelle** dépasse largement son apparence anecdotique. Quatre constructeurs, un format de littéral, deux projections complémentaires — l'addition semble modeste. Mais ce qu'elle apporte au langage est, en réalité, une discipline méthodologique : **la validation au point de création**.

31.12.1. Le principe : valider tôt, valider une fois

Au-delà des quatre types particuliers, ce chapitre soutient un principe applicable à toute la suite. Une valeur qui *peut* être validée ne devrait jamais entrer dans un programme sans avoir été validée. Et la validation devrait se faire **au point où la valeur est créée** — pas au point où elle est utilisée. Cette discipline économise des sites de contrôle, élimine la dispersion temporelle, et garantit que toute valeur en circulation dans le programme est, par construction, bien formée.

Le coût d'écriture est minime : remplacer `"alice@example.com"` par `[EMAIL "alice@example.com"]` ne coûte qu'une douzaine de caractères. Le bénéfice cognitif est important : tout lecteur ultérieur du code sait, sans inspection supplémentaire, que la valeur est de forme attestée. Le bénéfice opérationnel

est plus important encore : aucun cas de production ne pourra introduire silencieusement une adresse mal formée, parce que le runtime aurait refusé la construction.

31.12.2. Trois lieux de validation dans la Partie VIII

Avec ce chapitre, la Partie VIII a maintenant exposé **trois lieux distincts** où la validation peut se produire dans un programme VVL. Le tableau ci-dessous les récapitule, parce que leur articulation est essentielle à la compréhension d'ensemble.

Lieu de validation	Mécanisme	Quand	Quoi
au parse	dialecte (ch. 29)	à l'ingestion d'une chaîne	forme syntaxique de l'entrée
à la déclaration	module (ch. 30)	au chargement d'un fichier	cohérence des exports et imports
à la construction	valeur typée (ch. 31)	à la création d'une valeur	forme bien-formée du scalaire

Ces trois lieux ne se concurrencent pas — ils se composent. Une chaîne reçue d'une API est d'abord *parsée* (peut-être par un dialecte), ce qui décompose ; chaque fragment décomposé est ensuite *promu* en valeur typée (par un constructeur), ce qui valide ; les valeurs typées sont ensuite *injectées* dans un objet métier d'un module qui contient lui-même la cohérence (par sa frontière de package). Trois mécanismes, trois niveaux de garantie, une chaîne sans trou.

31.12.3. Vers le chapitre 32 : les tests intégrés

La validation à la construction est puissante mais elle ne couvre pas tout. Elle vérifie qu'une valeur a la **forme attendue** ; elle ne vérifie pas qu'un **programme** a le **comportement attendu**. Pour cette seconde garantie, il faut un quatrième lieu de validation : les **tests**. Le chapitre 32, qui suit, introduit le système `test:*` de VVL — `test:assert`, `test:equals`, `test:suite`, `test:summary` — qui permet d'écrire des suites de vérification au sein même des scripts VVL et de les exécuter automatiquement à chaque chargement.

Le lecteur attentif aura noté, à la section 30.8.1 du chapitre précédent, l'annonce d'un pattern *module + tests intégrés* qui n'a pas encore été complètement développé. Le chapitre 32 livre la pièce manquante. À l'issue de ce chapitre, VVL disposera des **quatre lieux complémentaires** de validation — au parse, à la déclaration, à la construction, et à l'exécution des tests — qui couvrent ensemble l'entièreté du cycle de vie d'un programme. Le langage, à ce stade, sera complet sur le plan de la garantie formelle.

32. Tests intégrés et auditabilité formelle

Mots-clés : *test, assertion, suite, audit, validation comportementale, test:assert, test:equals, test:suite, test:summary, non-throwing, truthiness, equality structurelle, régression, tutoriel auto-validant, smoke test, auditabilité*

La Partie VIII tend, avec ce chapitre, vers sa cohérence finale. Les trois chapitres précédents — dialectes (29), modules (30), valeurs typées (31) — ont exposé trois mécanismes complémentaires qui valident la **forme** des données et des structures. Une chaîne entrante est validée *au parse* ; une bibliothèque chargée est validée *à la déclaration* ; un scalaire créé est validé *à la construction*. Trois lieux, trois mécanismes, et l'illusion serait permise de croire que la validation est désormais complète. Elle ne l'est pas.

Un programme qui n'admet que des données bien formées peut encore se comporter de manière inattendue. Une règle ontologique correctement déclarée peut produire un diagnostic faux ; un module bien encapsulé peut exporter un keyword dont l'algorithme est buggé ; une valeur `EMAIL` parfaitement valide peut être routée vers le mauvais destinataire par une logique erronée. La forme est garantie, le comportement ne l'est pas. C'est précisément le rôle du quatrième lieu de validation que ce chapitre introduit : les **tests intégrés**, qui mesurent non pas la forme mais l'effet.

VVL expose pour cela un quartet de builtins : `[test:assert]` pour les conditions booléennes, `[test:equals]` pour les comparaisons structurelles, `[test:suite]` pour grouper des vérifications, et `[test:summary]` pour récapituler. Le système est volontairement modeste — il ne rivalise pas avec les frameworks de test Python ou JavaScript matures (pytest, jest) — mais il est **natif au langage**, ce qui change tout. Les tests parlent le même langage que le code testé ; ils peuvent être placés dans les mêmes fichiers ; ils peuvent inspecter le contexte d'exécution exactement comme le programme inspecté. Cette intimité est la propriété qui rend les tests utiles pour l'audit formel des programmes neuro-symboliques — règles induites par LLM (chapitre 17), stratégies évoluées par algorithme génétique (chapitre 26), clauses synthétisées par ILP (chapitre 27). Sans tests intégrés, ces règles apprises restaient des promesses non vérifiables ; avec eux, elles deviennent des hypothèses **auditables**.

32.1. Le quatrième lieu de validation

Avant d'examiner la mécanique des quatre builtins, il vaut la peine de positionner correctement ce que les tests intégrés apportent à la Partie VIII. La figure ci-dessous, qui prolonge le tableau de synthèse du chapitre 31, place les tests dans le cycle de vie d'un programme VVL.

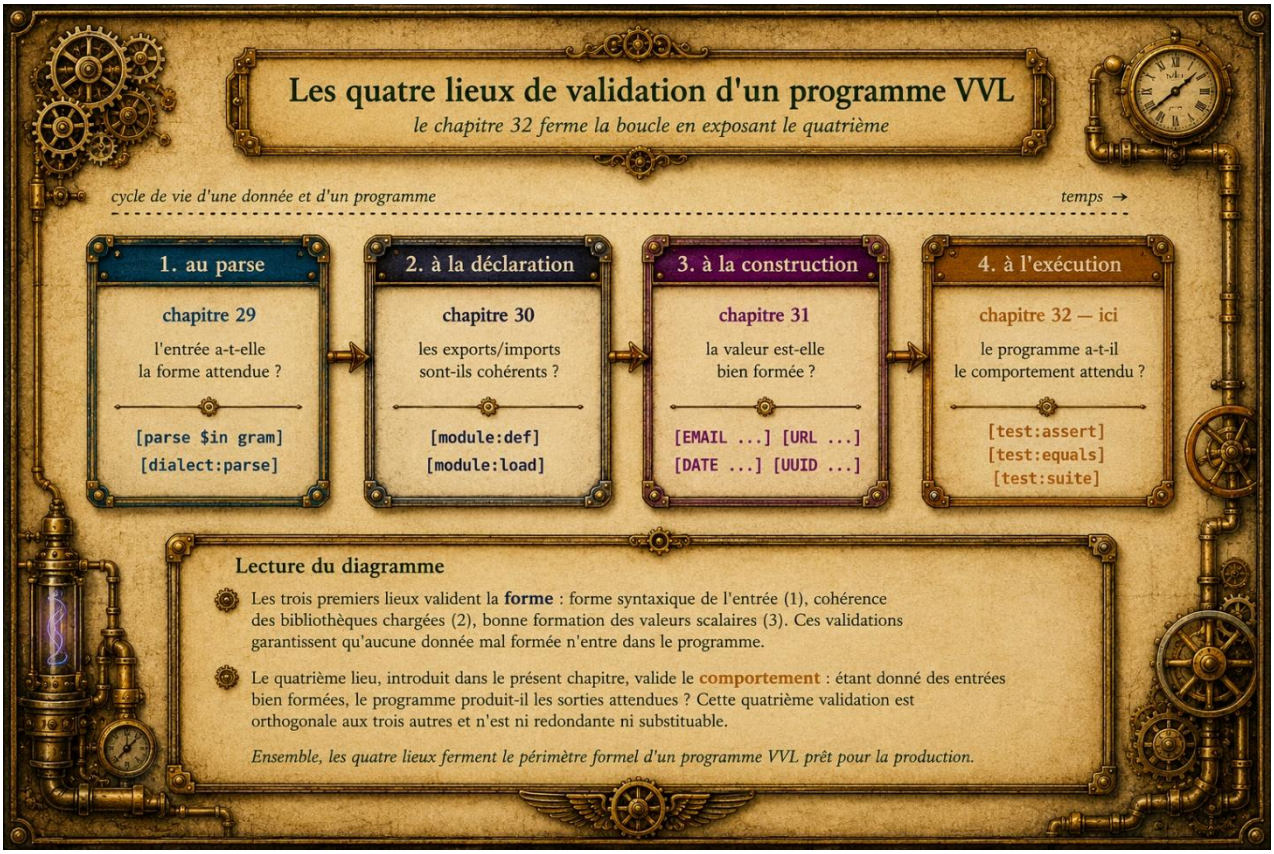


Figure 32.1 — *Les quatre lieux de validation d'un programme VVL. Les trois premiers (parse, déclaration, construction) valident la forme ; le quatrième, introduit ici, valide le comportement. Ensemble, ils ferment le périmètre formel d'un programme prêt pour la production.*

32.1.1. Pourquoi un système de test au cœur du langage

Une question légitime mérite d'être posée d'emblée : pourquoi VVL embarque-t-il son propre système de test, alors que la communauté Python — l'écosystème dans lequel le runtime VVL est implémenté — dispose déjà de cadres de test de qualité industrielle ? Pourquoi ne pas simplement déléguer les tests à pytest ou à unittest, et garder VVL focalisé sur ce qu'il fait de mieux ?

Trois raisons fondent la décision opposée. **Première raison — l'auto-audit des programmes VVL purs.** Un script VVL devrait pouvoir, à son seul exemple, démontrer son propre comportement attendu, sans dépendance sur un cadre externe qu'il faudrait installer et configurer. Un tutoriel VVL qui se termine par `[log "TASK COMPLETE"]` mais qui ne dit rien sur **ce que** la tâche a produit est moins instructif qu'un tutoriel qui se termine par une suite de tests prouvant que les résultats intermédiaires sont ceux annoncés. Sans tests intégrés au langage, cette auto-démonstration exigerait toujours une passerelle externe.

Deuxième raison — l'audit des règles apprises. Les chapitres 17, 26 et 27 ont introduit trois mécanismes d'apprentissage qui produisent des règles ou des stratégies *dynamiquement*. Un LLM synthétise une règle ; un algorithme génétique fait évoluer une population de candidates ; l'ILP infère une clause Prolog. Dans chacun de ces cas, la règle produite doit être **vérifiée** avant d'entrer en production — vérifiée non seulement sur sa forme syntaxique (que les trois lieux précédents assurent), mais sur son comportement. Tester une règle, c'est lui présenter un jeu d'exemples et vérifier qu'elle produit les conclusions attendues. Cette opération est exprimée naturellement en VVL ; elle serait pénible et fragile à travers une passerelle Python.

Troisième raison — la cohérence du périmètre. Si la philosophie du langage est de tout exprimer en VVL — programme, grammaires, modules, types, agents —, alors les tests doivent eux aussi parler VVL. L'alternative, qui ferait dériver une partie de la validation vers un autre langage, fragmenterait la chaîne de confiance : un test Python qui vérifie le comportement d'un script VVL relie deux mondes dont les sémantiques d'évaluation diffèrent, et chaque pont est un site potentiel de bug ou d'incompréhension. Les tests `test:*` natifs maintiennent la chaîne dans un seul monde.

32.1.2. La proximité avec le code testé

Une conséquence agréable du choix natif est la **proximité spatiale** entre le code et son test. Le pattern recommandé est de placer la suite de tests **juste après** la déclaration du module ou de la règle testés, dans le même fichier. Cette proximité a une vertu pédagogique évidente — un lecteur ouvre le fichier et voit, à côté de l'implémentation, les exemples canoniques d'usage — mais elle a aussi un effet plus profond sur la qualité du code : tout changement dans l'implémentation expose immédiatement les tests, qui sont rejoués au prochain chargement. Une régression silencieuse devient extrêmement difficile à introduire.

```
# Fichier : lib/math.vvl

[module:def Math [export inc square]
    [keyword inc    `[+ @1 1]]
    [keyword square `[* @1 @1]]
]

# Tests colles au module — executes a chaque [module:load]
[test:suite "Math"
    [test:equals 42 [Math:inc 41] "inc de 41"]
    [test:equals 25 [Math:square 5] "square de 5"]
    [test:equals 0  [Math:square 0] "square de 0"]
]
```

Cette discipline transforme chaque rechargement d'un fichier VVL en **smoke test** silencieux : si les tests collés au module échouent, le résumé de chargement le signale. Le pattern est si efficace qu'il est devenu, dans la pratique, la convention par défaut pour toute bibliothèque destinée à être réutilisée.

32.2. Le principe non-throwing

Avant d'examiner chacun des builtins en détail, il faut fixer le **principe central** qui les organise tous : les tests `test:*` ne lèvent pas d'exception en cas d'échec. Ils enregistrent l'échec dans une structure interne du contexte, retournent un objet JSON décrivant le résultat, et **laissent le programme continuer**. C'est cette propriété — appelée *non-throwing semantics* dans la littérature des cadres de test — qui distingue les tests VVL d'une assertion classique.

32.2.1. Ce qu'une assertion classique fait

Dans la majorité des langages, le builtin `assert` interrompt l'exécution dès qu'une condition est fausse. Le programme s'arrête, l'erreur est levée, le développeur diagnostique le premier point de divergence et corrige. Pour les vérifications d'invariants critiques dans un code de production — *« cette valeur ne*

doit jamais être nulle ici » —, ce comportement est exactement ce qu'on veut : la défaillance silencieuse est interdite, et l'interruption précoce limite la propagation des conséquences.

Mais pour les **tests de comportement**, le même comportement devient un handicap. Considérons une suite de dix vérifications qui couvrent les dix branches principales d'un algorithme. Si la première branche échoue et interrompt l'exécution, on apprend une chose ; mais on ignore si les neuf autres branches fonctionnent. Le développeur corrige la première, relance, et découvre la deuxième défaillance. Et ainsi de suite. Ce mode itératif rallonge considérablement le cycle de diagnostic et masque la **structure de la régression** : un changement qui aurait cassé trois branches sur dix se présente d'abord comme un bug unique, alors qu'il faudrait le voir comme une cascade.

32.2.2. Ce que test:* fait à la place

Les builtins `test:*` adoptent l'approche inverse. Chacun retourne **toujours** un objet JSON décrivant son résultat ; aucun ne lève d'exception. Le programme principal n'est jamais interrompu par un échec de test ; il est juste *informé*. Une suite de dix vérifications produit donc systématiquement un compte-rendu de dix résultats, qu'ils soient tous bons, tous mauvais, ou panachés.

```
# Une suite avec une defaillance intentionnelle

$suite = [test:suite "demo"
    [test:equals 3 [+ 1 2] "addition"]              # passe
    [test:equals 3 4 "intentionnel mismatch"]       # echoue
    [test:assert [== 1 1] "egalite triviale"]       # passe
]

# La suite a continue jusqu'au bout malgre l'echec central
[log $suite.total]                # => 3
[log $suite.passed]               # => 2
[log $suite.failed]               # => 1
[log $suite.ok]                   # => 0 (au moins un echec)
```

Le résultat est qu'**en une seule exécution**, le développeur reçoit la liste complète des défaillances. Si une régression a cassé trois branches, il les voit toutes les trois immédiatement, ce qui lui permet de chercher la cause racine commune. Si une refactorisation a, par chance, cassé seulement une branche annexe, il le voit aussi immédiatement, ce qui lui permet de relativiser. Le **diagnostic systémique** remplace le diagnostic ponctuel.

32.2.3. test:* coexiste avec try et attempt

Cette philosophie non-throwing s'aligne sur celle d'`[attempt]` (chapitre 28) : transformer une erreur en donnée plutôt qu'en exception. La filiation est explicite ; le runtime des tests utilise en interne le même mécanisme d'encapsulation. La conséquence pratique est que les trois mécanismes peuvent se composer sans frottement.

```
# Un test qui depend d'une operation potentiellement erronee

[test:suite "calcul-securise"
    # On enveloppe l'operation dans [attempt] pour neutraliser l'erreur,
    # puis on teste le statut.
    $r = [attempt [risky-calculation 42]]
    [test:assert $r.ok "calcul reussi"]
    [test:equals 100 $r.value "valeur attendue"]
]
```

Cette composition est ce qui permet d'écrire des tests robustes même pour des opérations qui pourraient échouer de manière imprévue. Le test n'est jamais interrompu ; il est juste informé que l'opération a échoué, et il peut le signaler comme tel.

À l'inverse, `[try try-body catch-body]` reste l'outil de la gestion d'erreur **par contrôle de flow**. Quand un programme veut continuer un calcul après une erreur — recommencer avec d'autres paramètres, par exemple —, c'est `[try]` qui convient. Quand il veut juste **constater** qu'une erreur a eu lieu pour la consigner, c'est `[attempt]`. Et quand il veut vérifier qu'un comportement attendu se produit, c'est `[test:*]`. Trois outils, trois rôles, aucun chevauchement.

32.2.4. Quand l'assertion classique reste appropriée

Un point doctrinal mérite clarification, parce qu'il revient régulièrement dans les discussions : les tests `test:*` **ne remplacent pas** les assertions classiques d'invariant. Quand un code de production veut établir qu'une certaine valeur **ne peut pas** être en dehors d'une plage attendue à un point donné, la bonne approche est encore une exception interrompante — par exemple via le builtin `[error]` du chapitre 4, qui lève une erreur sans appel à un cadre de test.

```
# Assertion d'invariant : interrompre si la postcondition est violee

[keyword sqrt-positif `[do
    $r = [sqrt @1]
    [when ($r < 0)
        `[error "sqrt-positif a recu un argument negatif"]
    ]
    $r
]]
```

Ce keyword est *en production*, pas dans une suite de tests. Si sa précondition est violée — quelqu'un l'a appelé sur un nombre négatif —, le programme doit s'arrêter immédiatement, pas continuer avec une valeur potentiellement aberrante. Le runtime n'a pas besoin de connaître `test:*` ici ; il a besoin d'une **erreur claire et arrêtante**.

La règle pratique se résume en deux mots : **`[error]` pour les invariants, `[test:*]` pour le comportement**. Les deux mécanismes vivent dans le même langage mais répondent à des questions différentes — l'un dit *« ceci ne devrait pas arriver »*, l'autre dit *« ceci devrait arriver »*. Les confondre, c'est introduire au mieux du bruit, au pire des comportements de production dégradés.

32.3. [test:assert] : l'assertion booléenne

Le builtin le plus simple du quartet est `[test:assert condition message?]`. Il évalue `condition`, retourne un objet JSON avec `ok: true` si la condition est *truthy*, `ok: false` autrement, et enregistre le résultat dans le contexte courant. Le paramètre `message` est optionnel mais fortement recommandé — il sert d'identifiant lisible dans les rapports de test.

32.3.1. Signature et retour

```
# Forme canonique avec message

$r = [test:assert [== 1 1] "egalite triviale"]
[log $r]
# => {"ok": 1, "name": "egalite triviale"}

# Forme sans message

$r2 = [test:assert [> 5 3]]
[log $r2.ok]                    # => 1
```

Le retour est un `VVL_Json` avec deux champs minimaux : `ok` (booléen) et `name` (string, le message fourni ou une valeur par défaut). En cas d'échec, le retour peut inclure des champs supplémentaires de diagnostic — par exemple `detail` pour décrire pourquoi la condition était fausse. Mais la signature minimale `{ok, name}` est garantie.

32.3.2. La truthiness en VVL

`[test:assert]` accepte n'importe quelle valeur VVL comme condition — pas seulement un booléen strict. Le runtime applique les règles de **truthiness** habituelles du langage pour décider si la valeur compte comme vraie. Ces règles sont fixées de longue date en VVL mais méritent d'être réénoncées ici parce qu'elles déterminent le comportement de `test:assert`.

Type	Truthy quand	Falsy quand
Num	la valeur est non-nulle (!= 0)	la valeur est zéro
String	la chaîne est non vide	la chaîne est vide ("")
List	la liste contient au moins un élément	la liste est vide
Json	le JSON contient au moins un champ	le JSON est {} ou null
Boolean	1 (true)	0 (false)
valeur typée	toujours truthy (la valeur existe)	— jamais —

Ces règles simplifient considérablement l'écriture des tests. On peut écrire `[test:assert [length $liste] "la liste est non vide"]` au lieu de `[test:assert [> [length $liste] 0] "la liste est non vide"]` ; les deux formes sont strictement équivalentes. De même, `[test:assert $obj.captures.head "capture extraite"]` teste à la fois la présence et la non-vacuité de la capture.

Une seule subtilité mérite d'être signalée : les **valeurs scalaires typées** du chapitre 31 (`EMAIL`, `URL`, `DATE`, `UUID`) sont **toujours truthy** dès qu'elles existent. Il n'y a pas, dans le MVP, de notion d'EMAIL vide ou d'URL nulle — un constructeur qui aurait reçu un payload invalide aurait levé une erreur. Si l'on a une valeur typée dans une variable, c'est qu'elle est bien formée, et `[test:assert]` la considère comme vraie.

32.3.3. Cas d'usage typiques

```
# Verification de presence d'une valeur
[test:assert $user "user existe"]

# Verification d'une condition complexe
[test:assert [and [exist $list "alice"] [> [length $list] 3]]
    "alice presente et liste consistante"]

# Verification de non-fuite hors d'un module
$r = [attempt [module:get LogTools internalCache]]
[test:assert (! $r.ok) "internalCache reste prive"]
```

Le troisième exemple mérite attention : il combine `[attempt]` (pour neutraliser l'erreur) et `[test:assert]` pour vérifier que l'erreur a bien eu lieu. Le pattern *« vérifier qu'une opération échoue comme attendu »* est récurrent — par exemple pour confirmer qu'un nom privé d'un module n'est effectivement pas exporté, ou qu'un type rejette correctement un payload invalide.

32.4. [test:equals] : l'égalité structurelle

Le deuxième builtin du quartet — et le plus utilisé dans la pratique — est `[test:equals expected actual message?]`. Il compare deux valeurs selon les règles d'**égalité structurelle** du langage, retourne `ok: true` si elles sont égales, et inclut dans le retour, en cas d'échec, un champ `detail` qui décrit l'écart constaté.

32.4.1. Convention de l'ordre des arguments

Une convention de doctrine importante : le premier argument est la valeur **attendue**, le second est la valeur **observée**. Cette convention suit l'usage de la majorité des frameworks de test modernes (xunit, jest, pytest avec keywords). Elle n'est pas vérifiée par le runtime — l'ordre est arithmétiquement indifférent pour la comparaison —, mais elle est essentielle à la lisibilité des messages d'erreur.

```
# Convention : expected en premier, actual en second

[test:equals 3 [+ 1 2] "addition simple"]
# expected = 3, actual = [+ 1 2] = 3, succes

[test:equals 3 4 "mismatch volontaire"]
# expected = 3, actual = 4
# => {"ok": 0, "name": "mismatch volontaire",
#     "detail": "expected 3, got 4"
```

Le champ `detail` est ce qui rend le diagnostic d'échec immédiat. Sans ce champ, un développeur lisant un rapport de test ne saurait pas, sans ouvrir le code, quelles valeurs précises ont divergé. Avec lui, le rapport donne suffisamment d'information pour pointer directement vers la cause.

32.4.2. L'égalité par type

Le comportement de `test:equals` dépend du type des valeurs comparées. Le tableau ci-dessous résume les règles d'égalité structurelle.

Type	Comparaison
Num	égalité numérique (3 == 3.0 est vrai)
String	égalité de caractères, sensible à la casse
Boolean	1 == 1, 0 == 0
List	récursive sur les éléments, dans l'ordre
Json	égalité de structure : mêmes clés, mêmes valeurs
valeur typée	mêmes type ET payload (égalité de mold)

L'égalité **récursive** sur les listes et les JSON est essentielle : elle permet de comparer des structures complètes sans avoir à écrire de boucle d'inspection. Une liste imbriquée, un objet de configuration profond, le retour d'une requête de base de données — toutes ces structures se comparent en une seule ligne.

```
# Egalite recursive sur des listes

[test:equals
    `[1 2 [3 4] "hello"]
    [list 1 2 [list 3 4] "hello"]
    "construction equivalente d'une liste imbriquee"
]
# => ok

# Egalite structurelle sur des objets JSON

[test:equals
    [object [name "Alice" age 30]]
    [object [age 30 name "Alice"]]
    "ordre des cles non significatif en JSON"
]
# => ok

# Egalite des valeurs typees

[test:equals
    [EMAIL "alice@example.com"]
    [EMAIL "alice@example.com"]
    "deux EMAIL au meme payload"
] # => ok
```

Une subtilité importante : pour les **valeurs typées**, l'égalité dépend à la fois du type ET du payload **tel que stocké** — pas du payload normalisé. Deux EMAIL construites avec `"Alice@Example.com"` et `"alice@example.com"` sont **différentes** selon `test:equals`, même si leur `[form]` est identique. Cette discipline reflète le principe du chapitre 31 : la casse stockée est préservée et significative.

32.4.3. Ce que test:equals ne fait pas

Une dernière précision utile : `test:equals` est conçu pour les **valeurs de données** — nombres, chaînes, listes, JSON, valeurs typées. Il n'est pas conçu pour les **valeurs opaques** du système : handles WebDriver, objets COM, canaux de communication, semaphores. Comparer deux semaphores avec `test:equals` retourne un résultat indéfini et ne doit pas être tenté. Pour ces objets de plateforme, la vérification doit passer par des inspections spécifiques (par exemple `[browser:title $driver]` pour comparer l'état d'un WebDriver, plutôt que la comparaison directe des handles).

32.5. [test:suite] : grouper et compter

Une suite de tests sans regroupement reste utilisable mais perd la dimension synthétique : on a dix résultats individuels, sans vue d'ensemble. Le builtin `[test:suite name body...]` répond à ce besoin en groupant un ensemble de tests sous un nom et en produisant un résumé agrégé.

32.5.1. Signature et exécution

`[test:suite]` est un builtin **raw** : il ne pré-évalue pas son corps. À l'invocation, il ouvre un compteur de tests, exécute le corps séquentiellement, agrège les résultats, et retourne un objet JSON résumé. Si un test échoue au milieu de la suite, **les tests suivants s'exécutent quand même** — c'est le principe non-throwing appliqué à l'échelle du groupe.

```
001  # Une suite typique
002
003  $resultat = [test:suite "arithmetique de base"
004      [test:equals 3 [+ 1 2] "addition"]
005      [test:equals 6 [* 2 3] "multiplication"]
006      [test:equals 5 [- 10 5] "soustraction"]
007      [test:assert [== 4 4] "egalite triviale"]
008  ]
009
010  [log $resultat]
011  # => {"ok": 1, "name": "arithmetique de base",
012  #     "total": 4, "passed": 4, "failed": 0,
013  #     "results": [...]}
```

Le champ `results` du retour contient la liste détaillée de chaque test enfant, dans l'ordre d'exécution. Pour des suites courtes, cette liste est rarement inspectée explicitement — les champs agrégés (`total`, `passed`, `failed`, `ok`) suffisent. Pour des suites plus longues, où il faut identifier précisément quels tests ont échoué, `results` est précieux.

32.5.2. Le couple ok / failed

Deux champs sont à comprendre clairement : `ok` est un booléen agrégé qui vaut `1` si **et seulement si** tous les tests de la suite ont réussi ; `failed` est un compteur entier des échecs. Les deux contiennent la même information dans des formes complémentaires.

```
# Decider en fonction du resultat agrege

$suite = [test:suite "tests-critiques"
    [test:equals 100 [calcul-critique 50] "C1"]
    [test:equals 200 [calcul-critique 100] "C2"]
]

# Forme booleenne
[when (! $suite.ok)
    `[error "tests critiques echouent — arret"]
]

# Forme compteur (utile pour les CI qui retournent un code de sortie)
[exit $suite.failed]
```

Le pattern le plus courant est d'utiliser `$suite.ok` comme prédicat pour décider de continuer ou d'interrompre. Dans un contexte d'intégration continue où la suite est exécutée par un système externe, `$suite.failed` peut être passé comme code de sortie au processus appelant — convention Unix où un code zéro signale le succès.

32.5.3. Imbrication des suites

Une suite peut contenir d'autres suites. Le runtime gère correctement l'imbrication : chaque suite enfant produit son propre résumé, qui s'agrège dans la suite parente. Cette propriété permet d'organiser des batteries de tests **hiérarchiques** quand le code testé est lui-même hiérarchique.

```
# Suite imbriquee : tester plusieurs modules en un seul appel

$global = [test:suite "bibliotheque-complete"
    [test:suite "Math"
        [test:equals 42 [Math:inc 41] "inc"]
        [test:equals 25 [Math:square 5] "square"]
    ]
    [test:suite "Strings"
        [test:equals "HELLO" [Strings:upper "hello"] "upper"]
    ]
]

[log $global.total]              # somme des tests imbriques
```

32.6. [test:summary] : la photographie du contexte

Le quatrième builtin du quartet, `[test:summary]`, retourne **toutes** les vérifications enregistrées dans le contexte courant depuis le début de la session — qu'elles aient été passées directement à `[test:assert]` ou `[test:equals]`, ou qu'elles aient été exécutées à l'intérieur d'une `[test:suite]`. Cette photographie agrégée est l'outil de récapitulation finale d'un script de test.

32.6.1. Le format de retour

```
# Apres plusieurs tests et suites disperses

[test:equals 3 [+ 1 2] "additif"]
[test:suite "module-Math"
    [test:equals 42 [Math:inc 41] "inc"]
]
[test:assert [== 1 1] "trivial"]

$bilan = [test:summary]
[log $bilan]
# => {"ok": 1, "total": 3, "passed": 3, "failed": 0,
#     "results": [...]}
```

Le format est identique à celui de `[test:suite]`, à la différence essentielle qu'il **n'isole pas** : il regroupe l'intégralité des tests du contexte. Si un script a effectué dix vérifications dispersées et trois suites de cinq vérifications, `[test:summary]` retournera un objet avec `total: 25` (10 + 3×5) et les compteurs agrégés correspondants.

32.6.2. Cas d'usage : le rapport final

Le pattern idiomatique d'un script de test bien conçu se termine par un appel à `[test:summary]`, suivi d'un log final qui peut être consommé par une chaîne d'intégration continue.

```
# Fichier : test-all.vvl

# (toutes les declarations de tests et suites)
[test:suite "Math" ...]
[test:suite "Strings" ...]
[test:suite "Dates" ...]

# Bilan final
$bilan = [test:summary]
[log "===== RAPPORT FINAL ====="]
[log "Total :  " $bilan.total]
[log "Passes : " $bilan.passed]
[log "Echecs : " $bilan.failed]

[when $bilan.ok
    [log "TASK COMPLETE"]
    [do [log "TASK FAILED"]
```

```
018          [exit $bilan.failed]] ]
```

Le `[log "TASK COMPLETE"]` final n'est pas anodin. Plusieurs chaînes d'intégration des tutoriels VVL — celles qui vérifient automatiquement que les fichiers `.vvl` de la documentation fonctionnent — utilisent précisément cette ligne comme **marqueur de succès** : si elle apparaît dans la sortie, le tutoriel est considéré comme passé ; si elle manque, le tutoriel a échoué. Le quartet `test:*` rend cette discipline systématique en attachant la décision finale au résultat des tests plutôt qu'à une présence inconditionnelle du marqueur.

32.7. Le quartet en synthèse

Pour fixer la place de chacun des quatre builtins, le tableau ci-dessous récapitule leur signature et leur rôle dans une batterie de tests typique.

Builtin	Rôle	Retour
[test:assert cond msg]	vérifier une condition truthy	{ok, name}
[test:equals exp act msg]	comparer deux valeurs structurellement	{ok, name, [detail]}
[test:suite name body]	grouper plusieurs tests sous un nom (raw)	{ok, name, total, passed, failed, results}
[test:summary]	récapituler toutes les vérifications du contexte	{ok, total, passed, failed, results}

Quatre builtins, deux formats de retour (individuel ou agrégé), une discipline non-throwing partagée. Le système est délibérément étroit, et c'est précisément ce qui le rend utilisable au quotidien. Un développeur qui apprend à écrire `[test:equals]` connaît, dans la même heure, le quartet entier.

Cette étroitesse contraste avec la richesse des cadres de test mainstream (pytest fournit plusieurs dizaines d'assertions différentes, jest une centaine). VVL fait un choix opposé : minimalité et composition. Si l'utilisateur a besoin d'une assertion plus spécifique — par exemple *« cette valeur est dans cet intervalle »* —, il l'exprime en composant `[test:assert]` avec une condition booléenne adaptée. Le langage hôte est suffisamment expressif pour que cette composition ne soit pas pénible, et l'absence de combinatoire d'API garde la surface d'apprentissage minimale.

32.8. Patterns d'écriture

La présentation des quatre builtins étant complète, on aborde maintenant la **discipline** d'écriture des tests en VVL. Plusieurs patterns recommandés se sont stabilisés au fil des projets ; ils ne sont pas imposés par le runtime mais ils gagnent à être suivis pour la lisibilité et la maintenabilité de la base de code.

32.8.1. Le tutoriel auto-validant

Le pattern le plus simple et le plus fréquent est le **tutoriel auto-validant**. Un fichier `.vvl` qui démontre une fonctionnalité contient, en plus du code didactique, une suite de tests qui prouve que le code fait ce qu'il prétend. Ce pattern transforme chaque tutoriel en sa propre garantie de fonctionnement.

```
# Fichier : 31_typed_values.vvl  (tutoriel des valeurs typees)

$email = [EMAIL "alice@example.com"]
$url   = [URL   "https://example.com"]
$date  = [DATE  "2026-05-24"]

$suite = [test:suite "31_typed_values"
    [test:equals "email" [type-of $email] "type-of EMAIL"]
    [test:equals "url"   [type-of $url]   "type-of URL"]
    [test:equals "date"  [type-of $date]  "type-of DATE"]

    [test:equals "alice@example.com" [form $email]
        "form normalise"]

    [test:equals "<DATE:2026-05-24>" [mold $date]
        "mold canonique"]
]

[log "passed: " $suite.passed "/" $suite.total]
[log "TASK COMPLETE"]
```

Le tutoriel sert deux fonctions : il est lisible comme document pédagogique, et il est exécutable comme test d'intégration. Cette double fonction réduit considérablement le coût de maintenance de la documentation : si une fonctionnalité change, ses tutoriels échouent à la première relance, et la documentation est mise à jour en même temps que le code. La dérive entre code et documentation, qui afflige la plupart des projets logiciels, devient mécaniquement difficile à introduire.

32.8.2. Le module avec tests collés

Le deuxième pattern, déjà esquissé au chapitre 30 et à la section 32.1.2, est le **module avec tests collés**. Une bibliothèque réutilisable place sa suite de tests juste après son `[module:def]`, dans le même fichier.

```
# Fichier : lib/strings.vvl

[module:def Strings [export upper lower reverse]
    [keyword upper   `[to-upper-case @1]]
    [keyword lower   `[to-lower-case @1]]
    [keyword reverse `[reverse-string @1]]
]

[test:suite "Strings"
    [test:equals "HELLO" [Strings:upper "hello"] "upper"]
    [test:equals "world" [Strings:lower "WORLD"] "lower"]
    [test:equals "olleh" [Strings:reverse "hello"] "reverse"]
    [test:equals ""      [Strings:upper ""] "upper de vide"]
]
```

Le test du **cas vide** dans la dernière ligne illustre un sous-pattern utile : tester explicitement les **cas-limites** dans le même groupe que les cas nominaux. Une bibliothèque qui plante sur une chaîne vide alors qu'elle accepte les chaînes ordinaires révèle un défaut de spécification que les tests collés capturent immédiatement.

32.8.3. Le fichier de tests externes

Pour les batteries de tests plus volumineuses — par exemple celles qui couvrent une bibliothèque mature avec des dizaines de cas-limites —, le pattern recommandé sépare les tests dans un fichier dédié, typiquement nommé `test-NAME.vvl`.

```
# Fichier : test-strings.vvl

[module:load "lib/strings.vvl"]

[test:suite "Strings — cas nominaux"
    [test:equals "HELLO" [Strings:upper "hello"] "upper"]
    [test:equals "world" [Strings:lower "WORLD"] "lower"]
]

[test:suite "Strings — cas-limites"
    [test:equals ""      [Strings:upper ""] "upper de vide"]
    [test:equals "A"     [Strings:upper "a"] "un caractere"]
]

[test:suite "Strings — unicode"
    [test:equals "ÉTÉ"   [Strings:upper "été"] "accentue"]
    [test:equals "café"  [Strings:lower "CAFÉ"] "accentue"]
]

$bilan = [test:summary]
[when (! $bilan.ok) `[exit $bilan.failed]]
[log "TASK COMPLETE"]
```

Cette séparation laisse le fichier `lib/strings.vvl` léger et concentré sur l'implémentation, tandis que `test-strings.vvl` peut grossir au fil du temps sans alourdir la bibliothèque. Le coût est que le test ne se relance pas automatiquement au chargement du module ; il faut l'invoquer explicitement, typiquement depuis un script CI.

32.8.4. Suite par fonctionnalité

Un dernier pattern, applicable surtout aux bibliothèques importantes : **une suite par fonctionnalité** plutôt qu'une suite par module. Cette discipline organise les tests par **comportement testé** plutôt que par **fichier source**, ce qui aligne la structure de test sur la structure des exigences.

```
# Fichier : test-feature-user-management.vvl

[test:suite "Inscription d'un nouvel utilisateur"
    # ...
]

[test:suite "Connexion d'un utilisateur existant"
    # ...
]

[test:suite "Reinitialisation de mot de passe"
    # ...
]
```

Ce pattern de **tests par fonctionnalité** est particulièrement utile quand la bibliothèque est consommée par plusieurs applications : chacune teste les fonctionnalités qui la concernent, sans avoir à connaître la structure interne de la bibliothèque. Le couplage entre code testé et test est faible, ce qui rend les refactorisations internes moins disruptives.

32.9. Tests des systèmes apprenants

Au-delà des patterns classiques, les tests intégrés de VVL acquièrent une importance particulière dans un contexte que les chapitres 17, 26 et 27 ont préparé : les **systèmes apprenants**. Quand une règle, une stratégie ou une clause est *produite dynamiquement* par un mécanisme d'induction — qu'il soit neural, génétique ou symbolique —, sa validité n'est pas garantie par sa simple existence. Elle doit être **vérifiée** par confrontation à un jeu d'exemples. C'est le rôle naturel des tests intégrés.

32.9.1. Audit d'une règle induite par LLM

Au chapitre 17, l'induction LLM zero-shot a produit des règles ontologiques synthétisées à partir d'une description en langage naturel. Une telle règle, une fois injectée dans le contexte, peut être testée comme n'importe quel comportement. Le pattern d'audit prend la forme suivante : on prépare un jeu de **cas connus** — exemples positifs et exemples négatifs —, on présente chacun à la règle, et on vérifie que la conclusion produite correspond à l'attendu.

```
# La regle induite (en sortie du LLM, deja injectee dans le contexte)
#
# [ontology:rule "fievre-haute"
#     when ($p ~~ "patient") and ($p.temp > 38.5)
#     then ($p.symptom = "fievre-haute")]

# Audit de la regle sur un jeu d'exemples

[test:suite "regle-fievre-haute"
    # Cas positif : devrait declencher
    $p1 = [ontology:assert [Patient [temp 39.2]]]
    [ontology:infer]
    [test:equals "fievre-haute" $p1.symptom "patient 39.2 declenche"]
```

```
014
015     # Cas limite : juste au-dessus du seuil
016     $p2 = [ontology:assert [Patient [temp 38.6]]]
017     [ontology:infer]
018     [test:equals "fievre-haute" $p2.symptom "patient 38.6 declenche"]
019
020     # Cas negatif : sous le seuil
021     $p3 = [ontology:assert [Patient [temp 37.2]]]
022     [ontology:infer]
023     [test:assert (! $p3.symptom) "patient 37.2 ne declenche pas"]
024 ]
```

Ce pattern transforme la **règle produite par LLM** en hypothèse formellement testée. Si le LLM a induit une règle trop laxiste — par exemple un seuil de 38.0 au lieu de 38.5 —, le cas négatif échoue et le diagnostic est immédiat. Si la règle est correcte, les trois tests passent et la règle est validée pour l'usage. Sans la suite de tests, la règle resterait une promesse non vérifiée.

32.9.2. Audit d'une stratégie évoluée par algorithme génétique

Au chapitre 26, un algorithme génétique a fait évoluer des populations de candidates jusqu'à converger sur une solution. Le champion de la dernière génération est une stratégie qu'il faut désormais **valider** sur un ensemble de tests indépendant — typiquement un *jeu de validation* distinct du jeu d'entraînement, pour mesurer la généralisation.

```
001 # Apres convergence du GA, on dispose d'un champion
002
003 $champion = [ga:best-of-generation $population]
004
005 # Validation sur un jeu independant
006
007 [test:suite "champion-GA-validation"
008     [foreach $cas $validation-set
009         `[do
010             $resultat = [evaluate-strategy $champion $cas.input]
011             [test:equals $cas.expected $resultat
012                 [concat "cas " $cas.id]]
013          ]
014     ]
015 ]
016
017 $bilan = [test:summary]
018 [log "Taux de generalisation : "
019     (* 100 (/ $bilan.passed $bilan.total)) "%"]
```

Le test ne se contente pas de constater le succès ou l'échec ; il **mesure** le taux de généralisation. Si le ratio `passed/total` est de 95 %, la stratégie est considérée comme robuste ; s'il est de 70 %, la convergence du GA était probablement trop précoce et il faut relancer avec d'autres paramètres. Cette **boucle d'audit quantifiable** est ce qui distingue un système de recherche bricolé d'un pipeline de production discipliné.

32.9.3. Audit d'une clause synthétisée par ILP

Le chapitre 27 a présenté l'ILP sound-and-complete, qui synthétise une clause Prolog à partir d'exemples positifs et négatifs. La propriété de complétude garantit, par construction, que la clause couvre tous les positifs et aucun négatif **du jeu d'entraînement**. Mais elle ne dit rien de son comportement sur de **nouveaux exemples**. Là encore, les tests intégrés viennent combler l'écart.

```
# La clause synthetisee par ILP

$clause = [ilp:learn $positifs $negatifs]

# On l'enregistre dans le contexte Prolog
[prolog:assert $clause]

# Audit sur un jeu de test independant

[test:suite "clause-ILP-generalisation"
    [foreach $ex $test-positifs
        `[test:assert [prolog:query $clause.head $ex]
            [concat "doit accepter " $ex]]
    ]
    [foreach $ex $test-negatifs
        `[test:assert (! [prolog:query $clause.head $ex])
            [concat "doit rejeter " $ex]]
    ]
]
```

Le double parcours — exemples positifs attendus comme acceptés, exemples négatifs attendus comme rejetés — donne une mesure d'efficacité par cas. Si tous les positifs sont acceptés mais que certains négatifs sont aussi acceptés à tort, la clause est **trop générale** ; à l'inverse, si certains positifs sont rejetés, la clause est **trop spécifique**. Les tests permettent un diagnostic raffiné qui peut nourrir une nouvelle itération d'ILP avec un jeu d'exemples enrichi.

32.9.4. Test-driven learning

Une discipline avancée, née de la combinaison des trois cas précédents, mérite mention : le **test-driven learning**. Plutôt que de produire d'abord une règle par induction puis de la tester *a posteriori*, on inverse l'ordre : on écrit d'abord les tests qui décrivent le comportement attendu, et on utilise ces tests comme **critère de fitness** pour orienter l'apprentissage.

Dans le cas d'un GA, le score de fitness d'une candidate est simplement le nombre de tests qu'elle passe. Dans le cas d'un LLM zero-shot, on peut prompt-engineer pour qu'il produise des règles candidates et les retenir uniquement si elles passent la suite. Dans le cas d'un ILP, les tests positifs et négatifs servent directement de jeu d'entraînement. Dans les trois cas, les tests intégrés deviennent l'**oracle** qui guide l'apprentissage.

Cette articulation est l'une des promesses les plus fortes de la programmation neuro-symbolique en VVL : le langage formel des tests et le langage de l'apprentissage ne sont qu'**un seul langage**. Les tests qui définissent l'attendu sont les mêmes qui valident l'appris. La distance entre spécification et apprentissage, qui dans d'autres environnements exige une traduction explicite, s'efface ici. On verra, à la fin du chapitre 34, comment cette articulation prend toute sa portée dans le bilan d'ensemble de l'ouvrage.

32.10. Limites du MVP et perspectives

Le système de tests présenté ici est, comme les autres composantes de la Partie VIII, un **MVP** qui couvre les besoins essentiels sans prétendre rivaliser avec les cadres de test matures. Cette section dresse l'inventaire des limites connues et des directions d'évolution.

32.10.1. Pas de mocks ni de stubs natifs

Une fonctionnalité couramment offerte par les cadres de test modernes — pytest avec `monkeypatch`, jest avec ses mocks — est la possibilité de **remplacer temporairement** une dépendance par une version contrôlée pour la durée d'un test. Le système VVL ne fournit pas, dans son MVP, de mécanisme natif de *mocking*. Un test qui dépend d'un appel réseau, d'une lecture de fichier, ou d'un service externe doit soit **réellement** invoquer ces dépendances (avec les coûts et la fragilité associés), soit envelopper les dépendances dans un keyword qu'on remplacera explicitement pour le test.

Le pattern de contournement repose sur la flexibilité des `[keyword]` redéfinissables. Le test commence par sauvegarder la définition courante d'un keyword, le remplace par une version contrôlée, exécute le test, et restaure la définition originale. C'est rustique mais ça fonctionne. Une API native — typiquement `[test:mock keyword new-impl]` — est inscrite à la roadmap mais n'est pas dans le MVP.

32.10.2. Pas de paramétrisation de tests

Les cadres modernes offrent souvent un mécanisme de **paramétrisation** : un même test exécuté plusieurs fois avec des entrées différentes. En pytest, c'est `@pytest.mark.parametrize` ; en jest, c'est `test.each`. VVL n'a pas, dans le MVP, de syntaxe dédiée à la paramétrisation. Le pattern de contournement est l'utilisation de `[foreach]` à l'intérieur d'une suite :

```
# Test parametre par enumeration explicite

[test:suite "addition — cas multiples"
    [for $cas `[
            [0 0 0]
            [1 2 3]
            [-1 1 0]
            [100 -50 50]
        ]
        `[test:equals [getValue $cas 2]
            [+ [getValue $cas 0] [getValue $cas 1]]
            [concat "addition cas " [dump $cas]]
        ]
    ]
]
```

Le pattern est verbeux comparé à la syntaxe dédiée des cadres mainstream, mais il est explicite et lisible. Pour les utilisateurs qui en feraient un usage intensif, une keyword utilitaire `[test:each list body]` pourrait être définie en une dizaine de lignes au-dessus du quartet `test:*`.

32.10.3. Pas d'isolation entre suites

Une limitation plus subtile : les tests `test:*` partagent **le même contexte VVL**. Une variable créée dans une suite reste accessible aux suites suivantes, un module chargé reste chargé, une règle ontologique assertée reste asserter. Cette absence d'isolation peut produire des **interférences** entre tests : un test qui modifie l'état (en ajoutant un fait, en appelant un setter, en altérant une variable globale) influence les tests suivants.

Pour des suites où l'isolation est critique, la pratique recommandée est de **réinitialiser explicitement** l'état au début de chaque test, soit en utilisant `[ontology:clear]` (chapitre 12) pour les faits, soit en encapsulant la logique sous test dans des modules avec frontière (chapitre 30), soit en lançant la suite dans une session VVL séparée. Une fonctionnalité native d'isolation — typiquement *« chaque test ouvre un sous-contexte qui est jeté à la fin »* — est inscrite à la roadmap.

32.10.4. Pas de timing ni de tests asynchrones

Le quartet `test:*` ne fournit pas de mécanismes natifs pour le **timing** des tests (mesure du temps d'exécution) ni pour les **tests asynchrones** (assertions sur des opérations qui se résolvent dans le futur). Pour les besoins de timing, l'utilisateur peut combiner les builtins horaires standards de VVL avec `test:assert` sur une condition de borne — pattern rustique mais effectif. Pour les opérations asynchrones, le mode parallèle introduit au chapitre 5 fournit les outils nécessaires, mais leur intégration native aux tests reste un chantier ouvert.

32.10.5. Pas d'assertions négatives spécialisées

Un cadre comme pytest fournit un `pytest.raises` pour vérifier qu'une opération lève bien l'exception attendue. VVL ne fournit pas, dans le MVP, de `test:throws` équivalent. Le pattern de contournement est la composition `[attempt]` + `[test:assert]` (déjà vu en 32.3.3) : on enveloppe l'opération dans `[attempt]`, on inspecte le statut, on vérifie qu'il est en échec. C'est suffisant pour les cas courants ; une API plus directe est envisageable mais non prioritaire.

32.11. Synthèse : le 4e lieu, et vers les plugins

Avec ce chapitre, la Partie VIII a refermé son périmètre principal : les quatre lieux de validation d'un programme VVL — au parse (29), à la déclaration (30), à la construction (31), à l'exécution (32) — sont désormais tous exposés. Un programme qui couvre les quatre dispose, par construction, des garanties suivantes : aucune donnée mal formée n'entre dans le système, aucune bibliothèque incohérente n'est chargée, aucune valeur scalaire mal-typée n'est créée, et aucune régression de comportement ne reste silencieuse plus longtemps qu'il ne faut pour exécuter la suite de tests.

32.11.1. Le sens du 4e lieu

Le présent chapitre a ajouté à la trilogie *parse/déclaration/construction* la dimension qui leur manquait : la validation **opérationnelle**, qui mesure non pas ce que les valeurs *sont* mais ce qu'elles *font* quand

on les laisse interagir. C'est cette dimension qui ferme la chaîne formelle, parce qu'elle est la seule à pouvoir attraper les bugs d'**algorithme**, par opposition aux bugs de **donnée**.

L'apport spécifique des tests intégrés VVL est triple. **Premièrement**, ils parlent le même langage que le code testé — pas de passerelle, pas de traduction, pas d'effet de frontière. **Deuxièmement**, ils sont non-throwing — un diagnostic complet en une exécution, pas une cascade itérative de corrections. **Troisièmement**, ils s'articulent naturellement avec les mécanismes d'apprentissage des chapitres 17, 26 et 27 — un test peut servir d'oracle, de critère de fitness, de jeu de validation, sans qu'aucun outil supplémentaire ne soit nécessaire.

Cette dernière propriété est ce qui rend les tests `test:*` distinctifs dans le paysage des cadres de validation. Dans la plupart des environnements, les tests vivent dans un monde séparé du code de production ; ils observent, ils n'agissent pas. Dans VVL, les tests sont des **expressions du langage** comme les autres. Ils peuvent être construits dynamiquement, transformés, sérialisés. Ils peuvent même être **appris** par les mêmes mécanismes que ce qu'ils sont supposés tester — une boucle réflexive qui ouvre des perspectives explorées par les chapitres qui suivent.

32.11.2. Articulation avec le chapitre 33 : les plugins signés

La Partie VIII compte un dernier chapitre technique avant la synthèse finale : le chapitre 33, qui introduit les **plugins signés Ed25519**. Le rapport avec les tests intégrés est intime mais pas immédiat — il mérite donc d'être explicité.

Les quatre lieux de validation présentés jusqu'ici se déploient **dans le langage hôte** : ils valident du code VVL natif, des données VVL natives, des structures VVL natives. Mais un programme VVL réel n'est pas un système fermé ; il interagit avec des bibliothèques externes — modèles ML, drivers de bases de données, APIs web. Ces interactions passent par des **plugins**, dont le code n'est pas écrit en VVL mais en Python (ou VScript). Le chapitre 33 expose le mécanisme par lequel ces plugins sont **autorisés à entrer** dans l'écosystème VVL : la signature cryptographique Ed25519, le magasin de clés de confiance, les modes d'exécution subprocess et inproc, la double garde de l'`inproc`.

Ce mécanisme est, en un sens, un **cinquième lieu de validation** : la validation **à la frontière externe**. Elle ne valide ni la forme ni le comportement du code en lui-même — un plugin signé peut tout à fait être buggé —, mais elle valide l'**origine** et l'**intégrité** du code, ce qui est la condition minimale pour que les quatre autres lieux puissent exercer leur fonction. Sans confiance dans l'origine du code externe, aucun test n'a de sens, parce que tout résultat peut être falsifié.

32.11.3. La gradation complète

On peut, en attendant le chapitre 33, fixer la gradation complète des cinq lieux de validation d'un programme VVL prêt pour la production :

Numéro	Lieu	Mécanisme	Question
1	au parse	dialecte	*la forme syntaxique est-elle correcte ?*
2	à la déclaration	module	*les bibliothèques chargées sont-elles cohérentes ?*
3	à la construction	valeur typée	*la valeur scalaire est-elle bien formée ?*
4	à l'exécution	test:*	*le comportement attendu se produit-il ?*
5	à la frontière	plugin signé	*le code externe est-il authentique ?*

Cinq lieux, cinq mécanismes, cinq questions complémentaires. Un programme VVL qui couvre les cinq dispose d'une **chaîne de validation sans trou** depuis la donnée brute jusqu'au code externe. Ce n'est pas une chaîne de garanties formelles au sens des langages dépendamment typés — VVL ne prétend pas être Coq ou Lean — mais c'est une chaîne **raisonnée**, où chaque maillon a un rôle clair et où aucun risque opérationnel important n'est laissé sans réponse. Le chapitre 33 va clore cette construction en exposant le cinquième et dernier maillon.

33. Plugins signés : Ed25519, subprocess/inproc, VScript

Mots-clés : *plugin, signature numérique, Ed25519, magasin de clés, anti-TOCTOU, subprocess, inproc, sandbox, VScript, SDK Python, allowlist réseau, secret, coffre, audit, tier, plugin-install, plugin-load, plugin-trust-add*

La Partie VIII touche à sa fin. Quatre chapitres ont exposé les quatre lieux de validation internes au langage : *parse, déclaration, construction, exécution*. Un programme qui couvre ces quatre lieux dispose, sur le **code VVL natif**, d'une chaîne de garanties presque complète. *Presque* — parce qu'un programme réel ne reste jamais dans les limites du code natif. Il appelle des modèles d'apprentissage automatique implémentés en Python, il consulte des bases de données via des bibliothèques C, il fait des requêtes HTTP vers des services externes, il manipule des images, des PDF, des protocoles industriels propriétaires. Tout cela vit en dehors de VVL — et tout cela doit, à un moment ou un autre, pouvoir être appelé depuis VVL.

Le mécanisme par lequel cette intégration externe se fait s'appelle, en VVL, le **plugin signé**. Un plugin est, à la racine, un keyword utilisateur dont l'implémentation est un **script Python** (ou, alternativement, un script **VScript**) — pas une expression VVL native. C'est exactement le pont qui manquait au langage : un moyen propre d'embarquer du code externe sans renoncer à la chaîne de confiance que les chapitres 29-32 ont patiemment construite.

Mais embarquer du code externe est, par nature, dangereux. Un script Python a accès au système de fichiers, au réseau, aux processus, à la mémoire. Laisser n'importe quel fichier Python déposé dans un répertoire du workspace s'exécuter à la demande reviendrait à abandonner toutes les garanties que le langage hôte essaie de maintenir. VVL résout cette tension par une discipline cryptographique : *un plugin ne s'exécute que s'il a été **signé** par un auteur dont la **clé publique** a été **explicitement importée** dans le **magasin de confiance** par un **administrateur** du workspace*. Cinq mots-clés en italique, cinq niveaux de contrôle — c'est le minimum pour qu'un script externe puisse être considéré comme légitime.

Ce chapitre déroule ce mécanisme dans le détail. Il sera plus long que les autres chapitres de la Partie VIII, parce que la discipline cryptographique a beaucoup de pièces et que les pièges sont nombreux. À l'inverse, le chapitre référence la version courte de la table des commandes, qui se trouve dans l'**Annexe B** de l'ouvrage ; ici, on privilégie la doctrine et les patterns d'usage.

33.1. La frontière externe : le 5e lieu de validation

Le chapitre 32 s'est conclu par la promesse d'un *cinquième lieu de validation* à examiner dans le présent chapitre : la frontière entre le code VVL natif et le code externe. Pour donner toute sa portée à cette promesse, il faut d'abord positionner précisément ce que la frontière externe valide — et surtout, ce qu'elle ne valide pas.

33.1.1. Modules vs plugins : la division du travail

Le chapitre 30 a introduit les **modules**, qui regroupent des commandes VVL natives dans des frontières de package nommées. Les modules sont l'outil idéal pour structurer du code écrit en VVL. Les plugins viennent compléter ce dispositif pour les cas où le code **ne peut pas** être écrit en VVL — parce qu'il faut appeler une bibliothèque qui n'existe que dans un autre langage, parce qu'il faut

atteindre des performances que le runtime VVL ne peut pas offrir, ou parce que la fonctionnalité visée touche au système d'exploitation à un niveau que le langage hôte ne souhaite pas exposer.

Critère	Module (ch. 30)	Plugin (ch. 33)
Langage d'implémentation	VVL natif	Python ou VScript
Exécution	runtime VVL ordinaire	sous-processus ou inproc sandboxé
Modèle de confiance	aucune signature requise	signature Ed25519 vérifiée
Audit du code	inspection VVL directe	inspection limitée + signature
Performance	celle du runtime VVL	celle du langage natif sous-jacent
Cas d'usage	logique métier, helpers	ML, I/O système, ponts externes

Ces deux mécanismes sont **strictement complémentaires** — ils ne se concurrencent jamais sur le même problème. Si la fonctionnalité visée est exprimable en VVL, on écrit un module ; si elle exige un langage externe, on écrit un plugin. Une fonctionnalité hybride — par exemple un service d'inférence ML qui combine une partie de traitement Python et une partie de logique de routage en VVL — se décompose naturellement en *plugin Python pour l'inférence* + *module VVL pour le routage*. La frontière entre les deux est nette.

33.1.2. Pourquoi une signature cryptographique

Une question légitime mérite d'être abordée frontalement : pourquoi exiger une signature cryptographique sur les plugins, plutôt que de se contenter — comme tant d'écosystèmes — d'une vérification *ad hoc* (somme de contrôle simple, vérification du chemin d'origine, white-list de noms autorisés) ? Trois raisons fondent la décision.

Première raison — la non-répudiation. Une signature cryptographique authentifie l'**auteur** du plugin, pas seulement son intégrité. Quand un plugin signé par la clé `prosuite-2026` s'exécute sur un workspace, l'audit peut tracer sans ambiguïté qui a publié ce code. Cette information serait perdue avec une simple somme de contrôle, qui ne dit rien sur l'origine.

Deuxième raison — la séparation des rôles. Le modèle cryptographique sépare physiquement l'**auteur** (qui détient la clé privée, hors-ligne) de l'**administrateur du workspace** (qui décide quelles clés publiques accepter). Cette séparation est essentielle dans les déploiements d'entreprise : l'auteur d'un plugin n'a pas besoin d'avoir un compte sur tous les workspaces qui voudront l'utiliser ; un administrateur peut, à l'inverse, accepter des plugins d'auteurs qu'il ne connaît pas en personne, simplement parce qu'il fait confiance à leur clé publique.

Troisième raison — la résistance au TOCTOU. Une fois la signature vérifiée à l'installation, le runtime conserve dans son record interne le **digest cryptographique** du fichier validé. À chaque chargement ultérieur du plugin (`[plugin-load]`), il recompare le digest on-disk au digest stocké dans le record. Si le fichier a été altéré entre l'installation et le chargement — par un attaquant qui aurait remplacé le `.py` sur disque, par exemple —, le digest ne correspond plus et le chargement est refusé. Cette **double vérification** (à l'installation puis à chaque load) protège contre une classe entière d'attaques que la littérature de sécurité désigne sous l'acronyme TOCTOU (*Time Of Check / Time Of Use*).

33.1.3. Ed25519 : le choix de l'algorithme

L'algorithme de signature retenu est **Ed25519** (EdDSA sur la courbe Edwards 25519). Ce choix est délibéré et mérite une justification rapide, parce qu'il conditionne plusieurs propriétés opérationnelles importantes.

Performance : Ed25519 produit des signatures en quelques microsecondes sur le matériel courant — la vérification au chargement d'un plugin n'introduit aucun délai perceptible, même pour des plugins chargés en boucle. **Taille de signature** : 64 octets fixes, ce qui rend le bundle `.vvlplugin` léger et la signature facilement transportable. **Robustesse cryptographique** : Ed25519 résiste aux attaques par timing et n'exige pas de source d'aléa de qualité au moment de la signature (déterministe), ce qui simplifie l'implémentation et réduit la surface d'attaque. **Maturité** : l'algorithme est standardisé (RFC 8032), implémenté dans toutes les bibliothèques cryptographiques modernes, et étudié depuis plus d'une décennie.

La dépendance d'implémentation est la bibliothèque Python `cryptography`. Si cette bibliothèque est absente d'un build figé (cas rare mais possible sur certains environnements minimalistes), le sous-système de plugins **se désactive proprement** : les commandes `[plugin-*]` lèvent des erreurs claires expliquant la dépendance manquante, mais le reste du langage continue de fonctionner. C'est une dégradation gracieuse plutôt qu'un échec catastrophique.

33.2. Le flux de confiance, vue d'ensemble

Avant de plonger dans le détail des commandes, il vaut la peine de fixer la **vue d'ensemble** du flux : qui signe quoi, qui transmet quoi, qui vérifie quoi. La figure ci-dessous présente le parcours complet, de la génération de clé par l'auteur jusqu'à l'utilisation effective du plugin dans un script.

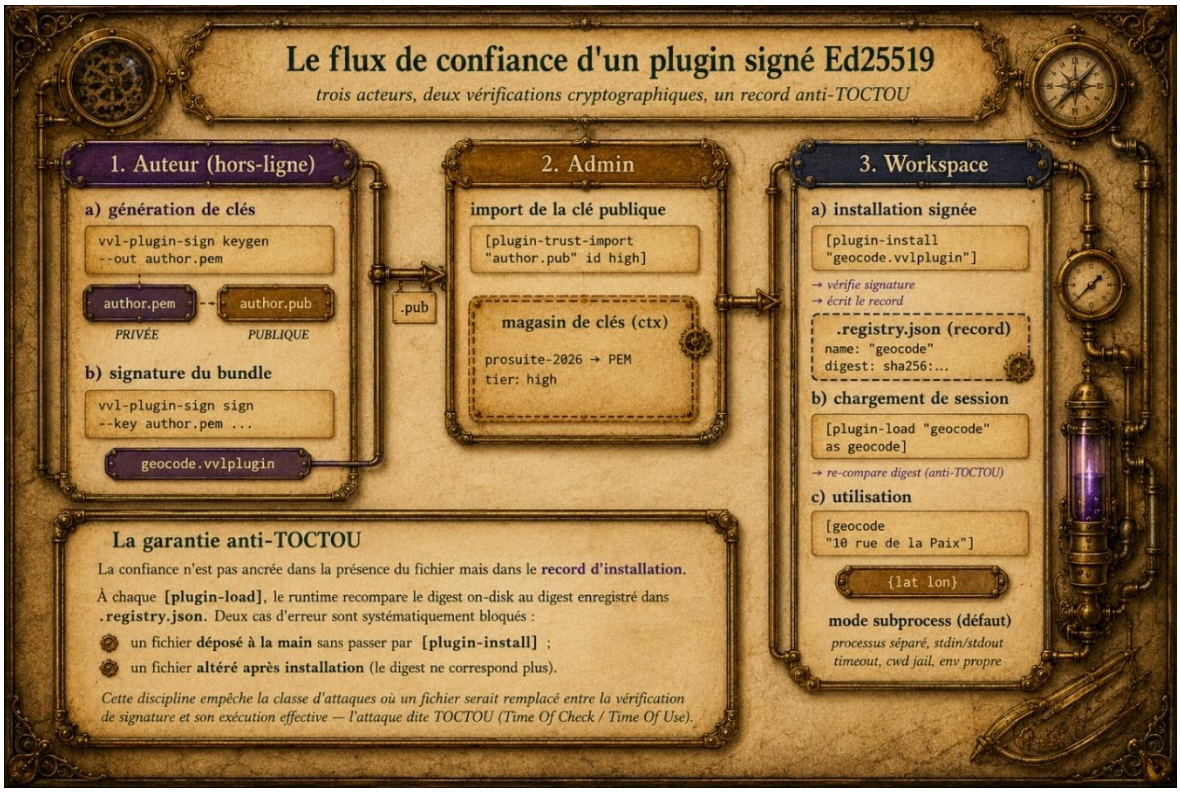


Figure 33.1 — *Le flux de confiance d'un plugin signé Ed25519. Trois acteurs (auteur, admin, workspace) ; deux vérifications cryptographiques (à l'installation, à chaque chargement) ; un record anti-TOCTOU qui ancre la confiance dans le digest plutôt que dans la présence du fichier.*

33.2.1. Les trois acteurs

Trois acteurs distincts participent au flux, avec des responsabilités strictement séparées.

L'auteur est l'entité — personne physique ou organisation — qui développe le plugin. Il détient la **clé privée** Ed25519, qu'il ne partage avec personne. Sur sa propre machine, hors-ligne, il génère le bundle `.vvlplugin` qui contient le code Python (ou VScript), le manifeste de déclaration, et la signature cryptographique de l'ensemble. Le bundle ainsi produit est *transportable* : il peut être envoyé par e-mail, déposé sur un serveur de distribution, partagé via un système de fichiers commun. Il est, en lui-même, suffisant pour être installé sur n'importe quel workspace dont l'administrateur aura préalablement autorisé la clé.

L'administrateur du workspace est l'entité qui décide *quels auteurs sont autorisés* à publier des plugins sur ce workspace particulier. Il dispose pour cela d'un **magasin de clés de confiance**, qu'il gère par les builtins `[plugin-trust-*]`. L'administrateur ne signe rien lui-même ; il **accepte** ou **refuse** les clés publiques. Une fois une clé acceptée et son tier fixé, les plugins signés par cette clé peuvent être installés sur le workspace par tout utilisateur qui en a les droits.

Le workspace lui-même — c'est-à-dire le runtime VVL en exécution — est l'acteur qui *applique* les décisions précédentes. Quand un utilisateur appelle `[plugin-install]`, le runtime vérifie que le bundle a une signature valide pour une clé présente dans le magasin de confiance. Quand un utilisateur appelle `[plugin-load]`, le runtime vérifie que le fichier on-disk correspond toujours au record d'installation. Quand le keyword exposé par le plugin est appelé, le runtime déclenche l'exécution du code Python (en sous-processus ou inproc selon la configuration) et collecte le résultat.

33.2.2. Les deux vérifications cryptographiques

Le flux comporte **deux vérifications cryptographiques** distinctes, qui jouent des rôles complémentaires.

Vérification 1 — à l'installation. Quand `[plugin-install]` est appelé, le runtime extrait du bundle la signature, le manifeste signé, et le code. Il identifie la clé publique référencée dans le manifeste, la cherche dans le magasin de confiance, et vérifie que la signature correspond bien à un message signé par cette clé. Si la vérification échoue — clé inconnue, signature corrompue, manifeste altéré —, l'installation est refusée. Si elle réussit, le runtime calcule un **digest SHA-256** du fichier code installé et l'enregistre dans `{workspace}/plugins/.registry.json` avec le nom du plugin, la version, le nom de la clé signataire, et son tier.

Vérification 2 — à chaque chargement. Quand `[plugin-load]` est appelé, le runtime ne re-vérifie pas la signature (qui pourrait être coûteuse pour un plugin volumineux), mais il **recalcule le digest** du fichier on-disk et le compare à celui stocké dans `.registry.json`. Si les deux digests correspondent, le chargement procède ; sinon, il est refusé. Cette deuxième vérification est ce qui ferme la fenêtre TOCTOU : un fichier modifié sur disque entre l'installation et le chargement est immédiatement détecté.

L'asymétrie entre les deux vérifications est intentionnelle. La vérification de signature est *coûteuse* (cryptographie asymétrique), donc on la fait *rarement* — une seule fois à l'installation. La vérification de digest est *peu coûteuse* (hash symétrique), donc on la fait *souvent* — à chaque chargement de session. Ensemble, les deux suffisent à garantir l'authenticité et l'intégrité du code à tout moment.

33.3. Le flux complet en commandes

Pour fixer concrètement la séquence des opérations, déroulons le flux complet sur un exemple : un plugin `geocode` qui transforme une adresse postale en coordonnées GPS via une API externe. Cet exemple servira de fil rouge dans plusieurs des sections suivantes.

33.3.1. Côté auteur : générer la clé et signer le bundle

L'auteur travaille sur sa propre machine, hors-ligne. Il utilise le script Python `vvl-plugin-sign.py` fourni avec le runtime — ce script est complètement indépendant de VVL et n'a besoin que d'un Python standard avec la bibliothèque `cryptography`. Deux étapes successives produisent les artefacts.

```
# Etape 1 : generation du couple de cles
#
# python scripts/vvl-plugin-sign.py keygen \
#     --out author.pem \
#     --pub author.pub
#
# Resultat :
#   - author.pem  (cle PRIVEE — a garder secrete, hors-ligne)
#   - author.pub  (cle PUBLIQUE — a transmettre aux admins)

# Etape 2 : signature du bundle
#
# python scripts/vvl-plugin-sign.py sign \
#     --key author.pem \
#     --key-id prosuite-2026 \
#     --name geocode --version 1.2.0 \
#     --entry geocode.py \
#     --mode subprocess \
#     --arity-min 1 --arity-max 1 \
#     --network api.example.com \
#     --secret api_key \
#     --auth default \
#     --out geocode.vvlplugin
```

Le bundle `geocode.vvlplugin` produit est une archive auto-suffisante qui contient : le code source du plugin (`geocode.py`), le manifeste de déclaration (nom, version, mode d'exécution, ressources demandées), et la signature cryptographique de l'ensemble. Le `--key-id` est l'identifiant que l'admin utilisera pour référencer cette clé dans son magasin — c'est lui qui apparaîtra dans les commandes `[plugin-trust-*]`.

Les paramètres du manifeste méritent commentaire. Le `--mode subprocess` indique que le plugin sera exécuté dans un sous-processus Python séparé (mode par défaut, examiné en détail en 33.6). L'`--arity-min` et l'`--arity-max` déclarent que le plugin attend exactement un argument. Les `--network`, `--secret`, et `--auth` déclarent les ressources sensibles auxquelles le plugin demande accès — examinées en détail en 33.7. Tous ces paramètres sont **signés** avec le code : un attaquant qui voudrait modifier le manifeste après signature serait détecté à la vérification.

33.3.2. Côté admin : importer la clé publique

L'auteur transmet la clé publique `author.pub` à l'administrateur du workspace — par n'importe quel canal raisonnable (e-mail, dépôt partagé, transmission physique sur clé USB pour les déploiements sensibles). L'administrateur dépose le fichier dans le sous-répertoire `_inbox` de son répertoire de plugins, se connecte avec ses droits administrateur (`[proadmin email pass]`), puis importe la clé.

```
# Connexion admin requise
[proadmin "admin@workspace.local" "password"]

# Import de la cle publique avec un tier de confiance
[plugin-trust-import "author.pub" prosuite-2026 high]

# Verification de l'enregistrement
[log [plugin-trust-list]]
# => [{"id":"prosuite-2026", "tier":"high",
#      "fingerlog":"a3:f2:..."}]
```

Le deuxième argument (`prosuite-2026`) est l'identifiant choisi pour cette clé — il doit correspondre au `--key-id` utilisé par l'auteur au moment de la signature. Si les deux ne correspondent pas, les plugins signés ne pourront pas être installés parce que la signature référencera un identifiant absent du magasin.

Le troisième argument est le **tier de confiance** : `standard` ou `high`. Cette distinction conditionne ce que les plugins signés par cette clé sont autorisés à faire. Les plugins en tier `standard` ne peuvent fonctionner qu'en mode **subprocess** (isolation par processus séparé). Les plugins en tier `high` peuvent demander le mode **inproc** (in-process, avec privilèges serveur) — mais sous condition supplémentaire d'un drapeau sur le workspace (examiné en 33.6). Un admin peut, à tout moment, promouvoir une clé de `standard` à `high` ou la rétrograder, via `[plugin-trust-update]`.

33.3.3. Côté workspace : installer, configurer, charger

Une fois la clé publique présente dans le magasin, n'importe quel utilisateur du workspace peut installer le bundle. Là encore, le bundle est attendu dans le sous-répertoire `_inbox` du répertoire de plugins.

```
# Installation du bundle (verifie la signature)
[plugin-install "geocode.vvlplugin"]

# Si le plugin declare des secrets, on les configure (admin)
[plugin-secret-set "geocode" "api_key" "sk_xxxxxxxxxxxxxxxx"]

# Si le plugin declare un profil d'auth, on l'enregistre (admin)
[plugin-login "geocode" "default" [object
    [type "bearer"]
    [token "tok_xxxxxxxxxxxxxxxxxxxxxxx"]
]]
```

```
012
013 # Chargement de session : lie le plugin a un keyword nomme
014 [plugin-load "geocode" as geocode]
015
016 # Utilisation : appel comme n'importe quel keyword
017 [log [geocode "10 rue de la Paix, Paris"]]
018 # => {"lat": 48.8688, "lon": 2.3293}
```

Cinq commandes ont suffi à passer du bundle nu à un keyword opérationnel. L'utilisateur final qui appelle `[geocode ...]` n'a pas besoin de savoir que c'est un plugin, encore moins qu'il est implémenté en Python — l'invocation est syntaxiquement identique à celle d'un keyword natif. Cette **transparence d'appel** est ce qui rend les plugins ergonomiques en pratique : ils s'intègrent dans le flux de programme sans rupture syntaxique.

33.3.4. Persister les liaisons : le pattern init.vvl

Une subtilité importante : la liaison `[plugin-load "geocode" as geocode]` est **per-session**. Si l'utilisateur ouvre un nouveau workspace ou redémarre la session, la liaison est perdue — le plugin reste installé (le record est durable), mais le keyword `geocode` n'est plus disponible jusqu'au prochain `plugin-load`. Pour automatiser ce rechargement, le pattern recommandé est de regrouper toutes les liaisons dans un fichier `init.vvl` chargé via `[import]` au démarrage.

```
001 # Fichier : init.vvl  (charge a chaque ouverture de session)
002
003 [plugin-load "geocode"    as geocode]
004 [plugin-load "translator" as translate]
005 [plugin-load "summarizer" as summarize]
006
007 # Maintenant tous les plugins sont rebindes a leurs keywords
008 [log "Plugins charges : geocode, translate, summarize"]
```

Ce fichier est exécuté par `[import "init.vvl"]` au début de chaque session ; les liaisons reprennent vie. Cette discipline imite, en plus léger, ce que les langages avec gestion de packages font automatiquement à l'import. La différence avec un `import` automatique est volontaire : VVL préfère que l'utilisateur **déclare explicitement** quels plugins il veut activer dans sa session, plutôt que de tout charger en aveugle.

33.4. Le magasin de clés de confiance

Le magasin de clés est, dans l'architecture du système de plugins, **la racine de la confiance**. Toute clé publique qu'il contient est considérée comme légitime ; tout plugin signé par une clé absente du magasin sera refusé. La gestion du magasin est donc une opération sensible, réservée aux administrateurs et encadrée par un ensemble de commandes dédiées.

Toutes les commandes `[plugin-trust-*]` exigent une connexion administrative préalable via `[proadmin email pass]`. Un appel sans cette précaution lève une `PermissionError` claire. Cette discipline empêche qu'un script utilisateur ordinaire — même malveillant — puisse modifier le magasin de confiance sans qu'un administrateur en soit informé.

33.4.1. Les commandes du magasin

Le tableau ci-dessous récapitule les huit commandes du magasin avec leur effet principal. La table de référence exhaustive — avec paramètres optionnels, codes d'erreur, valeurs de retour précises — se trouve dans l'**Annexe B** de l'ouvrage.

Commande	Effet principal
[plugin-trust-add id "PEM" tier]	ajoute une clé publique (PEM en argument direct)
[plugin-trust-import "f.pub" id tier]	lit la clé depuis plugins/_inbox/ (plus pratique)
[plugin-trust-list]	liste les clés enregistrées (sans la PEM brute)
[plugin-trust-get id]	détail d'une clé avec sa PEM complète
[plugin-trust-update id tier]	change le tier d'une clé existante
[plugin-trust-remove id]	révoque une clé (les plugins signés ne se rechargeront plus)
[plugin-trust-export id "out.pub"]	exporte une PEM vers plugins/_inbox/
[plugin-workspace-trust [ws] 0|1]	(dés)active plugins_trusted sur un workspace (gate inproc)

33.4.2. Les tiers standard et high

Le **tier de confiance** est un attribut associé à chaque clé enregistrée. Il prend l'une de deux valeurs : `standard` ou `high`. La distinction conditionne uniquement le mode d'exécution autorisé pour les plugins signés.

Tier	Modes autorisés	Quand l'utiliser
standard	subprocess uniquement	auteurs habituels, plugins fonctionnels ordinaires
high	subprocess + inproc (sous garde)	auteurs maison ou audités, plugins de confiance maximale

Le tier ne contrôle pas l'**installation** — un plugin signé par une clé `standard` peut être parfaitement installé et utilisé. Il contrôle seulement le **mode d'exécution** : un plugin qui demande `mode: "inproc"` dans son manifeste mais qui est signé par une clé `standard` verra le runtime refuser le chargement et tomber en erreur claire.

La pratique recommandée est conservatrice : par défaut, on enregistre les nouvelles clés en tier `standard`. La promotion à `high` se fait par décision explicite, après audit du code du plugin et confiance établie envers l'auteur. Cette promotion est réversible — un `[plugin-trust-update id standard]` rétrograde une clé `high` au tier inférieur, ce qui invalide immédiatement la possibilité d'exécution en `inproc` pour les plugins concernés.

33.4.3. La révocation : [plugin-trust-remove]

Un cas particulier mérite attention : que se passe-t-il quand une clé est **révoquée** ? Si un administrateur appelle `[plugin-trust-remove prosuite-2026]`, la clé disparaît du magasin. Les plugins déjà installés dont la signature référence cette clé restent **techniquement présents sur disque**, mais leur chargement échoue à la prochaine tentative — parce que la vérification de signature, redéclenchée si la clé manque dans le magasin, ne trouve plus de quoi vérifier.

```
# Scenario de revocation

[proadmin "admin@workspace.local" "password"]
[plugin-trust-remove prosuite-2026]

# Les plugins signes par cette cle ne se rechargeront plus
$r = [attempt [plugin-load "geocode" as geocode]]
[log $r]
# => {"ok":0, "error":{"type":"TrustError",
#                      "message":"key prosuite-2026 not in trust store"}}
```

Cette propriété est essentielle pour les cas de **compromission** : si l'on apprend qu'une clé privée a été volée, sa révocation immédiate dans le magasin de toutes les workspaces concernées **bloque l'exécution** des plugins potentiellement compromis, sans qu'il soit nécessaire de chasser chaque fichier `.vvlplugin` un par un. Le mécanisme suit, à grande échelle, la discipline des autorités de certification du web : c'est la **clé** qu'on révoque, pas chaque certificat individuel.

33.5. Cycle de vie d'un plugin

Une fois la clé du signataire enregistrée, le cycle de vie d'un plugin sur le workspace passe par cinq commandes principales : `[plugin-install]`, `[plugin-load]`, `[plugin-unload]`, `[plugin-remove]`, plus les deux commandes d'introspection `[plugins-list]` et `[plugin-info]`. Cette section examine chacune brièvement.

33.5.1. Installation : [plugin-install]

`[plugin-install "name.vvlplugin"]` est la **commande qui ancre** un plugin sur un workspace. Elle est la seule qui modifie le record durable des plugins installés. Quatre actions sont effectuées en séquence par le runtime.

(1) Lecture du bundle depuis `{workspace}/plugins/_inbox/`. **(2) Vérification de la signature** : la clé référencée par le manifeste est cherchée dans le magasin de confiance ; la signature est vérifiée contre l'algorithme Ed25519. Si l'une ou l'autre étape échoue, l'installation est refusée et aucune trace n'est laissée sur le disque. **(3) Vérification de la policy** : le manifeste est inspecté pour s'assurer qu'il ne déclare pas de ressources interdites au tier de la clé signataire. **(4) Écriture du `.py`** (ou `.vscript`) dans `{workspace}/plugins/`, calcul du digest SHA-256, et enregistrement du record dans `.registry.json`.

Le bundle d'entrée (`_inbox`) est, à l'issue de l'installation, conservé pour permettre une éventuelle réinstallation ou audit. Le fichier de code installé (`{name}.py`) est, lui, **inviolable** au sens où toute altération ultérieure sera détectée au prochain chargement.

33.5.2. Chargement de session : [plugin-load]

`[plugin-load "name" as keyword]` est la **commande de session** qui lie le code installé à un keyword utilisable. Elle ne modifie pas le record d'installation ; elle ouvre simplement une entrée dans `ctx.plugins[keyword]` qui dirige le runtime vers l'exécuteur approprié.

Avant de lier, la commande effectue la **re-vérification du digest** déjà évoquée en 33.2.2 : le fichier on-disk est rehashé et comparé au digest stocké dans `.registry.json`. Si les deux correspondent, la liaison procède. Sinon, le chargement est refusé.

Deux variantes existent. La forme par défaut, `[plugin-load "name" as keyword]`, utilise le mode d'exécution déclaré dans le manifeste (typiquement `subprocess`). La forme étendue, `[plugin-load "name" as keyword mode "inproc"]`, force explicitement le mode inproc — sous réserve des conditions de tier et de workspace examinées en 33.6. Une mention importante : `[plugin-load]` est un builtin **raw**, comme `[keyword]`, parce que le nom du keyword (`as geocode`) ne doit pas être pré-évalué.

33.5.3. Désactivation : [plugin-unload]

`[plugin-unload keyword]` est la **commande inverse** de load : elle retire la liaison de session. Le keyword cesse d'être disponible pour appel. Le plugin reste installé — son fichier sur disque et son record ne sont pas touchés —, mais il n'est plus actif. C'est utile pour libérer des ressources (un plugin en mode inproc peut tenir un sous-état en mémoire), pour effectuer des tests d'isolation, ou pour préparer un rechargement avec d'autres paramètres.

```
# Decharger une liaison de session
[plugin-unload geocode]

# Le keyword n'est plus disponible
$r = [attempt [geocode "Paris"]]
[log $r.ok]                      # => 0

# Mais le plugin reste installe et peut etre rebinde
[plugin-load "geocode" as geocode]
[log [geocode "Paris"]]          # => {lat lon}
```

33.5.4. Désinstallation : [plugin-remove]

`[plugin-remove "name"]` est la commande de **désinstallation complète**. Elle effectue trois actions : (1) si le plugin est chargé en session, elle le décharge automatiquement ; (2) elle supprime le fichier de code de `{workspace}/plugins/` ; (3) elle retire l'entrée correspondante du record `.registry.json`. La commande est **idempotente** — l'appeler deux fois ne provoque pas d'erreur, et l'appeler sur un plugin déjà absent retourne simplement un statut non-modifié.

33.5.5. Introspection : plugins-list et plugin-info

Les deux dernières commandes du cycle de vie sont d'**introspection**. Elles n'ont aucun effet de bord et ne modifient ni le record ni la session.

```
# Liste de tous les plugins installes avec leur statut
[log [plugins-list]]
# => [{name: "geocode",    installed: 1, loaded: 1, mode: "subprocess"},
#     {name: "translator", installed: 1, loaded: 0, mode: "subprocess"},
#     {name: "summarizer", installed: 1, loaded: 1, mode: "inproc"}]

# Detail d'un plugin specifique
```

```
[log [plugin-info "geocode"]]
# => {name: "geocode", version: "1.2.0",
#     signer: "prosuite-2026", signer-tier: "high",
#     digest: "sha256:...",
#     loaded: 1, keyword: "geocode",
#     mode: "subprocess",
#     declared-network: ["api.example.com"],
#     declared-secrets: ["api_key"],
#     declared-auth: ["default"]}
```

Ces deux commandes sont fréquemment utilisées dans les scripts d'audit, de migration, ou simplement de débogage interactif. La table complète des champs de retour est documentée en Annexe B.

33.6. Modes d'exécution : subprocess vs inproc

L'une des décisions de design les plus discutées de l'architecture des plugins concerne le **mode d'exécution**. Comment exactement le code Python est-il exécuté quand l'utilisateur appelle son keyword ? VVL offre deux réponses : `subprocess` et `inproc`. Chacune a ses propriétés, ses contraintes, et ses cas d'usage.

33.6.1. Mode subprocess : l'isolation par processus

Le mode `subprocess` est le **mode par défaut** et le plus restrictif. Quand un keyword de plugin est appelé en ce mode, le runtime fait quatre choses : il lance un nouveau processus Python qui exécute le fichier `{plugin}.py` ; il transmet la requête (arguments, secrets, paramètres) en JSON sur l'`stdin` du processus ; il lit la réponse en JSON sur l'`stdout` ; il tue le processus à la fin (ou par timeout).

```
# Cote VVL : appel ordinaire
[log [geocode "10 rue de la Paix"]]

# Cote runtime (invisible pour l'utilisateur) :
#
# 1. lancement : python geocode.py
# 2. envoi via stdin : {"args": ["10 rue de la Paix"],
#                       "keyword": "geocode",
#                       "plugin": "geocode",
#                       "secrets": {"api_key": "sk_..."},
#                       "auth": {"default": {...}}}
# 3. lecture sur stdout : {"ok": true, "value": {"lat": ..., "lon": ...}}
# 4. terminaison du sous-processus
```

Le mode subprocess offre plusieurs propriétés d'isolation importantes. **Cwd contraint** : le sous-processus a son répertoire courant fixé à `{workspace}/files/`, ce qui limite naturellement les accès au système de fichiers. **Environnement nettoyé** : seules les variables d'environnement strictement nécessaires sont passées, ce qui empêche le code d'inspecter des secrets dans `$PATH`, `$HOME`, etc. **Timeout** : le sous-processus est tué après un délai configurable (typiquement quelques dizaines de secondes), ce qui borne le coût d'une exécution malveillante ou buggée. **Mort du processus à la fin** : aucun état ne survit entre deux appels, ce qui rend chaque invocation indépendante.

Le coût est la **performance** : lancer un sous-processus Python prend typiquement 50 à 200 millisecondes selon la machine, ce qui rend le mode subprocess inadapté aux appels très fréquents ou aux opérations qui exigent une latence faible. Pour la majorité des cas d'usage des plugins — appels API, traitements ponctuels, transformations isolées —, ce coût est acceptable. Pour les cas où il devient prohibitif, le mode `inproc` apporte une alternative.

33.6.2. Mode inproc : la performance et ses contreparties

Le mode `inproc` (in-process) charge le module Python **directement** dans le runtime VVL et appelle la fonction `handle(args, ctx)` du plugin avec le contexte réel. Le coût d'invocation devient celui d'un appel de fonction Python ordinaire — moins d'une milliseconde —, ce qui rend possible des plugins appelés en boucle ou dans des chemins de code critiques.

La contrepartie est lourde : le plugin **partage l'espace mémoire et les privilèges** du serveur VVL. Il peut accéder à tout ce que le serveur peut accéder : système de fichiers complet, réseau sans allowlist, processus système, contexte VVL en lecture-écriture. Il peut, théoriquement, modifier le runtime lui-même. Cette dangerosité est ce qui justifie la **double garde** que le runtime applique avant d'autoriser ce mode.

33.6.3. La double garde du mode inproc

Pour qu'un plugin puisse être chargé en mode inproc, **deux conditions cumulatives** doivent être satisfaites.

Condition	Mécanisme	Qui décide
1. clé du signataire en tier high	[plugin-trust-update id high]	admin global du runtime
2. workspace marqué plugins_trusted=1	[plugin-workspace-trust ws 1]	admin du runtime, par workspace

Cette double garde n'est pas un raffinement décoratif. Elle reflète une **séparation des responsabilités** : la décision *« cet auteur est suffisamment fiable pour avoir un accès privilégié »* (tier high) est prise une fois, au niveau du magasin global. La décision *« ce workspace particulier accepte les plugins privilégiés »* (plugins_trusted) est prise par workspace, ce qui permet de configurer différemment un workspace de production (qui n'autorise pas inproc) et un workspace de développement (qui l'autorise).

Si l'une ou l'autre des deux conditions n'est pas remplie, un chargement avec `mode "inproc"` est refusé avec un message d'erreur explicite qui pointe la condition manquante. Cette précision facilite le diagnostic : un développeur qui ne comprend pas pourquoi son plugin ne se charge pas en inproc voit immédiatement quel attribut activer.

```
001  # Tentative de chargement inproc sans les conditions
002  $r = [attempt [plugin-load "ml-classifier" as classify mode "inproc"]]
003  [log $r]
004  # => {"ok":0, "error":{"type":"PolicyError",
005  #     "message":"inproc requires tier=high AND plugins_trusted=1 ;
006  #                key tier is standard"}}
```

33.6.4. Quand choisir quoi

Pour les cas pratiques, la règle de choix se résume ainsi : **par défaut, subprocess**. C'est le mode sûr, qui couvre 95 % des cas d'usage des plugins (appels API, ponts vers des bibliothèques de traitement de données, exports vers des formats variés). Le coût d'invocation est compatible avec presque tous les patterns interactifs.

Le mode `inproc` ne se justifie que dans trois situations spécifiques. **Situation 1** : plugin appelé à très haute fréquence dans une boucle critique (par exemple un classifieur ML invoqué pour chaque enregistrement d'une stream). **Situation 2** : plugin qui maintient un état coûteux à initialiser (par exemple un modèle de langage chargé en mémoire). **Situation 3** : plugin qui interagit avec le contexte VVL en lecture-écriture sur des structures complexes (ce qui passerait mal par sérialisation JSON).

Hors de ces trois cas, l'usage de `inproc` est un sur-engagement de sécurité pour un bénéfice marginal. La pratique mature consiste à commencer en subprocess et à passer en inproc uniquement quand le profil de performance le justifie clairement, avec un audit explicite du code du plugin et une décision documentée.

33.7. Plugins VScript : le compromis sandboxé

Le runtime accepte, à côté des plugins Python, des plugins écrits en **VScript** — un langage de script embarqué décrit en détail dans la documentation du runtime. Le manifeste du plugin déclare le langage via `lang: "vscript"` (par opposition à `lang: "python"` qui est le défaut) ; l'entrée du plugin est alors un fichier `.vscript` au lieu d'un fichier `.py`.

33.7.1. Pourquoi un troisième langage

À première vue, ajouter une troisième option — après VVL natif et Python — peut sembler une complication injustifiée. La raison d'être de VScript tient à un **compromis de sécurité** que ni VVL natif ni Python ne couvrent.

VVL natif est sûr (toute la chaîne du langage hôte le contrôle) mais peu performant et limité aux fonctionnalités natives. Python est puissant mais dangereux (accès complet à l'OS via les sous-processus). VScript se place au milieu : il est **plus expressif que VVL** sur certaines opérations (manipulation de chaînes, calcul numérique, structures de contrôle) et **plus restrictif que Python** sur les capacités systèmes — notamment, il **n'a pas d'I/O réseau native**. Cette absence est cruciale : elle permet d'exécuter un plugin VScript directement **in-process et sandboxé** sans la double garde du mode inproc Python.

Caractéristique	Plugin Python	Plugin VScript
Expressivité	maximale	intermédiaire
Performance subprocess	50-200 ms par appel	— sans objet —
Performance inproc	< 1 ms par appel	< 1 ms par appel
I/O réseau	accessible (allowlist SDK)	allowlist stricte (sandbox)
I/O fichier	accessible	restreinte au workspace
Double-garde requise	oui pour inproc	non (sandbox suffit)
Cas d'usage	ML, ponts externes	logique métier complexe

33.7.2. L'objet plugin en VScript

Un plugin VScript exécute dans un environnement où l'objet global `plugin` expose les ressources sensibles déclarées par le manifeste. C'est cet objet qui sert de point d'entrée à l'allowlist : les secrets, les profils d'auth, et les appels réseau passent **tous** par lui.

```
' Plugin VScript : un exemple de geocode minimal
' (syntaxe pseudo-VScript, simplifiee pour l'illustration)

Dim apiKey, bearerToken, response

' L'objet "plugin" est injecte par le runtime VVL
apiKey      = plugin.secret("api_key")
bearerToken = plugin.bearer("default")

response = plugin.get("https://api.example.com/v1/geocode?q=" & args(0))

' Retour ordinaire VScript
result = response
```

L'allowlist réseau de VScript est **stricte par construction** : le moteur VScript embarqué dans VVL n'expose pas de primitive d'ouverture de socket. Tout appel réseau doit obligatoirement passer par `plugin.get`, `plugin.post`, etc., qui vérifient que le domaine visé figure dans la liste `network` déclarée par le manifeste. Un plugin qui tenterait de contourner cette restriction ne trouverait simplement pas l'API nécessaire — la barrière est *au niveau du langage*, pas au niveau d'une vérification d'exécution.

Cette propriété — la **barrière dans le langage** — est ce qui distingue qualitativement le sandboxing VScript du sandboxing Python. Pour un plugin Python, l'allowlist réseau est appliquée par les helpers du SDK officiel (`http_get`, etc.), mais rien n'empêche techniquement un script Python de faire `import urllib.request` et d'ignorer le SDK. La discipline doit alors être maintenue par revue de code. Pour VScript, la discipline est mécanique : si le langage ne fournit pas l'API, le script ne peut pas l'utiliser.

33.8. Connecteurs API : secrets, login, allowlist réseau

La majorité des plugins déployés en pratique sont des **connecteurs API** : ils appellent un service externe (REST, GraphQL, gRPC) qui exige une authentification et qui réside derrière une URL. Trois mécanismes complémentaires encadrent ces connecteurs dans VVL : la déclaration du **réseau autorisé** dans le manifeste, le **coffre de secrets** géré par l'admin, et les **profils d'authentification** stockés par plugin.

33.8.1. Déclaration dans le manifeste

Le manifeste signé du bundle déclare les ressources sensibles auxquelles le plugin demande accès. Trois listes principales y figurent :

```
{
  "name": "geocode",
  "version": "1.2.0",
  "entry": "geocode.py",
  "mode": "subprocess",
  "network": ["api.example.com", "*.service.example"],
  "secrets": ["api_key"],
  "auth":    ["default"]
}
```

Ces déclarations ont deux fonctions complémentaires. **Fonction informative** : l'administrateur qui inspecte `[plugin-info "geocode"]` voit exactement ce que le plugin demandera comme accès, ce qui éclaire la décision d'installation. **Fonction restrictive** : à l'exécution, le runtime ne transmet au plugin que les secrets et profils d'auth listés ici. Un plugin qui essaierait de lire un secret non déclaré ne le trouverait simplement pas — le runtime ne lui aurait pas transmis l'information.

Crucialement, **toutes ces déclarations sont signées** avec le code. Un attaquant qui modifierait le manifeste après signature — par exemple pour étendre la liste `network` à des domaines arbitraires — verrait sa modification détectée à l'installation. La frontière des accès n'est donc pas seulement déclarative ; elle est cryptographiquement attestée par la signature de l'auteur.

33.8.2. Le coffre de secrets

Les secrets — clés d'API, tokens, mots de passe — sont stockés **chiffrés** dans `{workspace}/plugins/.secrets.json`. Leur gestion est entièrement réservée à l'admin, via cinq commandes.

Commande	Effet
[plugin-secret-set plugin key value]	stocke un secret chiffré
[plugin-secret-get plugin key]	lit la valeur d'un secret (admin uniquement)
[plugin-secret-list plugin]	liste les noms de secrets stockés (pas les valeurs)
[plugin-secret-remove plugin key]	supprime un secret
[plugin-audit n?]	retourne les n derniers événements runtime des plugins

Le pattern d'usage est de configurer les secrets **une fois** par admin, généralement au moment de l'installation du plugin. Le code du plugin ne lit jamais le fichier `.secrets.json` directement ; il reçoit les secrets par l'intermédiaire du runtime, qui ne lui transmet que ceux explicitement déclarés dans le manifeste.

33.8.3. Profils d'authentification

Beaucoup d'APIs exigent une authentification plus structurée qu'une simple clé — par exemple un *bearer token* OAuth, ou un couple identifiant/mot de passe pour HTTP Basic Auth. Pour ces cas, le mécanisme des **profils d'authentification** stocke des structures JSON complètes par plugin.

```
# Configuration d'un profil bearer pour le plugin geocode

[proadmin "admin@workspace.local" "password"]

[plugin-login "geocode" "default" [object
    [type  "bearer"]
    [token "tok_xxxxxxxxxxxxxxxxxxxxxxxx"]
]]

# Plus tard, on peut lister les profils...
[log [plugin-login-list "geocode"]]
# => [{"name":"default", "type":"bearer"}]

# ... ou les supprimer
[plugin-logout "geocode" "default"]
```

Comme pour les secrets, les profils sont transmis au plugin **uniquement** s'ils figurent dans la liste `auth` du manifeste signé. Le nommage d'un profil — `default`, ou tout autre identifiant choisi par l'admin — permet à un plugin de gérer plusieurs comptes de service en parallèle (par exemple un compte production et un compte staging) sans complication particulière.

33.8.4. Audit : qui a fait quoi

La dernière commande de la famille, `[plugin-audit n?]`, retourne les `n` derniers événements runtime des plugins : qui a chargé quoi, à quel moment, avec quel résultat. Cette traçabilité est destinée aux administrateurs qui veulent vérifier, *a posteriori*, l'activité des plugins sur leur workspace. L'audit n'est pas un mécanisme d'observabilité en temps réel — pour cela, les logs ordinaires du runtime conviennent mieux — mais un journal forensique d'événements significatifs.

33.9. Le SDK Python : stdlib-only

Côté Python, le runtime expose un **SDK officiel** qui simplifie l'écriture des plugins en mode subprocess (et accessoirement en inproc). Ce SDK est volontairement **stdlib-only** : il ne dépend d'aucune bibliothèque externe au-delà de la bibliothèque standard de Python. Cette contrainte est imposée pour deux raisons : portabilité (un plugin peut tourner sur n'importe quelle installation Python sans dépendance à installer), et auditabilité (toutes les opérations passent par des modules de la stdlib, dont le comportement est documenté et stable).

33.9.1. La structure minimale d'un plugin

Un plugin Python en mode subprocess suit un patron canonique : il importe les helpers du SDK, définit une fonction `handle(args, ctx=None)` qui implémente la logique, et appelle `serve(handle)` à la fin pour démarrer la boucle d'interaction avec le runtime.

```
# Fichier : geocode.py
#
# from vvl_plugin_sdk import serve, ok, fail
#
# def handle(args, ctx=None):
#     adresse = args[0]
#     if not adresse:
#         return fail("adresse vide")
#     # ... logique de geocoding ...
#     return ok({"lat": 48.8566, "lon": 2.3522})
#
# if __name__ == "__main__":
#     serve(handle)
```

Le SDK fournit trois fonctions de retour. `ok(value)` enveloppe une valeur de succès. `fail(message)` enveloppe une erreur. La fonction `handle` peut aussi retourner directement une valeur — auquel cas le SDK l'enveloppe automatiquement en `ok`. Cette uniformité du protocole de retour est essentielle : du côté VVL, le runtime reçoit toujours un JSON de la forme `{"ok": ..., ...}`, indépendamment de la complexité du calcul.

La même fonction `handle` fonctionne dans les deux modes d'exécution. En mode subprocess, c'est `serve(handle)` qui établit la boucle stdin/stdout ; en mode inproc, le runtime appelle `handle(args, ctx)` directement, en passant un contexte VVL réel. Cette portabilité d'écriture est ce qui permet de basculer entre les deux modes sans modifier le code source du plugin.

33.9.2. Les helpers pour APIs externes

Pour les plugins qui appellent des APIs HTTP, le SDK expose un ensemble de helpers qui encapsulent la complexité des requêtes et appliquent automatiquement l'allowlist réseau déclarée dans le manifeste.

Helper	Effet
get_secret(name)	lit un secret par son nom (KeyError si non déclaré)
get_auth(name)	lit un profil d'auth complet
bearer_auth(name)	retourne un header Authorization: Bearer ...
api_key_header(name, header)	retourne un header X-API-Key: ... (ou autre)
http_get(url, headers)	requête GET avec vérification d'allowlist
http_post(url, body, headers)	requête POST avec vérification d'allowlist
http_request(method, url, ...)	forme générique avec contrôles complets

Le pattern d'usage combine secrets et requêtes en quelques lignes lisibles :

```
# Helper d'appel API typique
#
# from vvl_plugin_sdk import serve, ok,
#                            get_secret, bearer_auth, http_get
#
# def handle(args, ctx=None):
#     headers = bearer_auth("default")
#     headers.update({"X-API-Key": get_secret("api_key")})
#     data = http_get(
#         "https://api.example.com/v1/me",
#         headers=headers,
#     )
#     return ok(data)
#
# if __name__ == "__main__":
#     serve(handle)
```

Pour les plugins VScript, l'équivalent a été présenté en 33.7.2 : l'objet `plugin` expose `secret`, `bearer`, `apikey`, `get`, et `post` avec une sémantique strictement équivalente. La différence d'implémentation (Python utilise le module `urllib.request` de la stdlib ; VScript utilise un client interne au moteur) est invisible pour l'auteur du plugin.

33.9.3. Le protocole sous-jacent

Pour les développeurs avancés qui voudraient comprendre ce qui se passe sous le SDK, le protocole d'échange en mode subprocess est minimal. Le runtime envoie sur `stdin` un JSON de la forme :

```
{
  "args":    ["10 rue de la Paix"],
  "keyword": "geocode",
  "plugin":  "geocode",
  "policy":  {"mode": "subprocess", "tier": "high"},
  "secrets": {"api_key": "sk_xxx"},
  "auth":    {"default": {"type": "bearer", "token": "tok_xxx"}}
}
```

Et il attend sur `stdout` une réponse de la forme `{"ok": true, "value": ...}` ou `{"ok": false, "error": "..."}`. Cette simplicité est délibérée : un développeur qui voudrait implémenter un SDK alternatif — par exemple en Go, en Rust, ou dans un autre langage qui supporterait l'exécution en subprocess — n'aurait qu'à respecter ce protocole minimal pour interopérer avec le runtime VVL. Aucune dépendance Python spécifique n'est exigée par le protocole lui-même.

33.10. Limites du MVP et perspectives

Le système de plugins présenté ici est, comme tous les autres mécanismes de la Partie VIII, un MVP. Il couvre solidement les besoins courants — connecteurs API, ponts ML, intégrations système — mais il a des limites connues qui méritent d'être explicitement signalées.

33.10.1. Pas de sandbox OS pour les plugins Python

Le mode subprocess offre une **isolation au sens VVL** : cwd contraint, environnement nettoyé, timeout, mort du processus. Mais ce n'est pas une **sandbox OS au sens strict**. Un script Python en sous-processus peut, par exemple, importer `urllib.request` directement et ignorer l'allowlist du SDK pour faire des requêtes vers des domaines arbitraires. La discipline d'allowlist repose, dans cette version, sur la convention d'utiliser le SDK officiel et sur la revue de code à l'audit.

Une vraie sandbox OS — typiquement par conteneur Linux (cgroups + namespaces) ou par Job Object Windows — exigerait une intégration plus profonde avec le système hôte et alourdirait considérablement le déploiement. Ce chantier est ouvert dans la roadmap mais hors-périmètre du MVP. Pour les déploiements sensibles où l'allowlist doit être *mécaniquement* infranchissable, la voie recommandée actuellement est soit le plugin VScript (sandbox au niveau du langage), soit le déploiement de l'ensemble du runtime VVL lui-même dans un conteneur contraint.

33.10.2. Windows et la limite d'isolation

Sur Windows, le sous-processus Python ne bénéficie d'aucune isolation réseau ou système au-delà du jail applicatif fourni par le runtime VVL lui-même. Cette limite est explicitement documentée pour éviter toute fausse confiance : un plugin signé par une clé `standard` sur Windows reste un sous-processus normal, qui peut lire et écrire des fichiers en dehors du workspace si le système le permet. La signature garantit l'authenticité du code, pas son inoffensivité au niveau du système d'exploitation.

Le mode `inproc`, par construction, est encore plus permissif : le plugin partage les privilèges du serveur VVL, sans aucune frontière OS. Cette dangerosité est précisément ce qui justifie la double garde (tier `high` + plugins_trusted). L'utilisateur qui active inproc le fait en connaissance de cause.

33.10.3. Pas de mise à jour automatique

Quand l'auteur d'un plugin publie une nouvelle version, le workspace ne se met pas automatiquement à jour. La nouvelle version doit être déposée explicitement dans `_inbox` et installée par `[plugin-install]`, qui remplace alors le record précédent. Cette discipline manuelle est volontaire — elle empêche les comportements opaques où un plugin pourrait évoluer sans que l'admin en soit informé —, mais elle a un coût opérationnel sur les déploiements avec plusieurs dizaines de plugins.

Une fonctionnalité de **mise à jour assistée** est envisageable pour les versions futures : par exemple, une commande `[plugin-update]` qui détecte automatiquement les nouvelles versions disponibles dans un dépôt configuré et propose de les installer avec un diff de manifeste. Cette commande pose des questions de gouvernance (où le dépôt vit, qui le contrôle) qui sortent du périmètre du MVP.

33.10.4. Renvoi à l'Annexe B pour la table exhaustive

Le présent chapitre a privilégié la **doctrine** et les **patterns d'usage** sur la **référence exhaustive** des commandes. Pour la table complète — paramètres optionnels, valeurs de retour précises, codes d'erreur, comportement aux cas-limites —, le lecteur est renvoyé à l'**Annexe B** de l'ouvrage, qui regroupe l'ensemble des commandes `[plugin-*]` avec leur signature formelle et leurs exemples canoniques.

Cette répartition documentaire est intentionnelle. Un chapitre principal qui inclurait la table complète serait illisible ; une annexe qui ne contiendrait que la table serait insuffisante sans le contexte doctrinal.

La division *doctrine en chapitre, référence en annexe* est devenue, au fil de la rédaction, le format standard pour les sujets dont l'API est volumineuse.

33.11. Cas d'usage

Avant la synthèse finale, trois cas d'usage illustrent comment les plugins s'inscrivent dans des contextes réels de programmation neuro-symbolique. Comme dans les chapitres précédents, ces cas ne sont pas hypothétiques : ils reflètent des patterns rencontrés dans la base de code VVL en production.

33.11.1. Pont vers un modèle de langage

Le premier cas d'usage est l'**intégration d'un modèle de langage** — typiquement un LLM accessible par API ou par appel local. Le plugin encapsule l'appel HTTP et la sérialisation des réponses ; il expose à VVL un keyword `llm:complete` qui prend un prompt et retourne une complétion.

```
# Cote VVL : utilisation du plugin LLM
[plugin-load "openai-bridge" as llm:complete]

$response = [llm:complete [object
    [prompt "Generer une regle ontologique pour ..."]
    [max-tokens 500]
    [temperature 0.2]
]]

[log $response.content]
```

Cet usage est l'incarnation directe du **pont neuro-symbolique** : le plugin Python encapsule la partie *neurale* (l'appel au modèle de langage), pendant que le contexte VVL appelant continue à exercer la partie *symbolique* (gestion de l'ontologie, validation des règles produites, audit par les tests intégrés). La séparation est nette et auditable.

33.11.2. Connecteur pour base de données vectorielle

Le deuxième cas d'usage est l'**accès à une base de données vectorielle** — typiquement utilisée pour la recherche sémantique dans les systèmes RAG du chapitre 14. Le plugin expose un keyword qui prend une requête en langage naturel et retourne les documents les plus proches selon une distance cosinus dans un espace d'embeddings.

```
[plugin-load "vector-db" as vdb:search]

$docs = [vdb:search "patients avec syndrome metabolique" 5]

# Les documents retournes sont des structures VVL ordinaires
[foreach $doc $docs
    `[log $doc.title " (" $doc.score ")"]
]
```

Une particularité de ce cas : le plugin maintient typiquement une **connexion persistante** à la base de données, ce qui pourrait justifier l'usage du mode `inproc` pour éviter de recréer la connexion à chaque appel. Si la décision en ce sens est prise, l'auteur du plugin doit la documenter clairement et son admin doit promouvoir la clé en tier `high` après audit.

33.11.3. Export vers un format propriétaire

Le troisième cas d'usage est plus prosaïque : l'**export** d'un état VVL vers un format propriétaire qu'aucune bibliothèque Python publique ne supporte — par exemple un format binaire interne à une entreprise. Le plugin sert de pont entre la structure VVL et la spécification du format.

```
[plugin-load "internal-format-exporter" as export:internal]

$rapport = [object
    [titre "Bilan Q3"]
    [donnees $resultats]
    [auteur "Service Analyses"]
]

$bytes = [export:internal $rapport]
[file:write "rapport.iexf" $bytes]
```

Ce cas est le plus simple et le plus fréquent : aucune dépendance externe au-delà du runtime Python, pas de réseau, pas de secret. Le plugin sert juste de **traducteur de format**. La signature Ed25519 reste néanmoins exigée — c'est la convention uniforme qui prévaut, quelle que soit la complexité du plugin.

33.12. Synthèse : la chaîne fermée à cinq lieux

Avec ce chapitre, la Partie VIII achève son projet : exposer les **cinq lieux de validation** qui ferment le périmètre formel d'un programme VVL prêt pour la production. La gradation, annoncée en 32.11.3, est désormais complète et peut être contemplée dans son ensemble.

Numéro	Lieu	Chapitre	Mécanisme	Question
1	au parse	29	dialecte	forme syntaxique ?
2	à la déclaration	30	module	bibliothèques cohérentes ?
3	à la construction	31	valeur typée	valeur bien formée ?
4	à l'exécution	32	test:*	comportement attendu ?
5	à la frontière externe	33	plugin signé Ed25519	code externe authentique ?

33.12.1. Une chaîne, pas un empilement

Ces cinq lieux ne forment pas un empilement de vérifications redondantes. Chacun valide une **dimension différente** de la confiance opérationnelle : la forme syntaxique des entrées textuelles ; la cohérence des bibliothèques chargées ; la bonne formation des valeurs scalaires individuelles ; le comportement attendu de la logique programmée ; l'authenticité du code externe invoqué. Aucune dimension n'en remplace une autre, et une faille à un niveau quelconque compromet l'ensemble.

Un programme qui valide la forme de ses entrées mais n'authentifie pas son code externe est vulnérable au code malveillant. Un programme qui authentifie son code externe mais ne teste pas son comportement est sujet aux régressions silencieuses. Un programme qui teste son comportement mais ne valide pas la forme de ses entrées est fragile aux ingestions malformées. C'est l'**articulation des cinq lieux** qui ferme le périmètre, pas la présence de l'un quelconque pris isolément.

33.12.2. Modules + plugins : la complémentarité finale

Le présent chapitre a, en particulier, complété l'apparente concurrence entre **modules** (chapitre 30) et **plugins** (chapitre 33). Ces deux mécanismes vivent dans le même espace conceptuel — *un namespace de commandes nommées et réutilisables* — mais ils répondent à deux questions différentes. Le module gère le **code VVL natif**, avec encapsulation et frontière de package, sans signature requise parce que le code est inspectable. Le plugin gère le **code externe**, avec signature cryptographique parce que le code dépasse le périmètre d'inspection du langage hôte.

Dans une base de code mature, les deux mécanismes coexistent harmonieusement. Quelques plugins assurent les ponts critiques vers l'extérieur (modèles ML, bases vectorielles, connecteurs API) ; plusieurs dizaines de modules structurent la logique métier en VVL natif ; et tout cela vit dans un même workspace avec la garantie que rien n'entre, n'exécute, ou ne produit de comportement non-attesté.

33.12.3. Vers le chapitre 34 : la synthèse de l'ouvrage

La Partie VIII est complète. Avec ce chapitre, l'ensemble des fonctionnalités infrastructurelles du langage est exposé : méta-programmation (28), dialectes (29), modules (30), valeurs typées (31), tests intégrés (32), plugins signés (33). Mais l'ouvrage lui-même n'est pas terminé. Il reste à **synthétiser** l'ensemble du parcours — des chapitres fondationnels sur la syntaxe (1-10) aux chapitres ontologiques (11-15), aux chapitres d'apprentissage (16-21), aux chapitres avancés sur les agents et l'auditabilité (22-27), et enfin à la présente partie sur l'infrastructure (28-33). Le chapitre 34, qui suit, occupe ce rôle de synthèse.

Ce dernier chapitre ne présentera pas de nouveau builtin. Il dressera, à hauteur de surplomb, le bilan transversal des **paradigmes d'écriture des règles — main, induction LLM, induction ILP — et de la plateforme infrastructurelle qui les rend tous utilisables en production. Il évoquera aussi les directions d'évolution restées ouvertes. C'est la clôture de la Partie VIII : le langage y est complet en tant que plateforme, et la Partie IX qui suit en expose les extensions les plus récentes**.

34. Synthèse — VVL comme plateforme neuro-symbolique mûre

Mots-clés : *synthèse, paradigme, plateforme, neuro-symbolique, industrialisation, audit, règle, pipeline, hybride, perspective, conclusion, bilan, cartographie, strates, mot final*

Trente-trois chapitres plus tôt, ce livre s'ouvrait sur une question apparemment simple : *comment programmer un système qui raisonne ?* Trois cents pages plus tard, la question a reçu plusieurs réponses superposées, et l'objet qu'elle décrivait — le langage VVL et sa plateforme — est suffisamment exposé pour qu'on puisse enfin le contempler d'un seul regard.

Ce chapitre n'introduit aucun nouveau builtin, aucune nouvelle syntaxe, aucune nouvelle convention. Il n'est qu'un **bilan**, et il assume cette modestie : la nouveauté ne se trouve plus dans le contenu mais dans la mise en perspective. À ce stade, l'ouvrage a fait son travail technique ; il lui reste à dire, en termes simples, ce qu'il a construit, à quoi cela sert, et ce que cela ouvre. C'est cette triple tâche que les sections suivantes accomplissent — par la rétrospection (34.1), par la systématisation des quatre paradigmes d'écriture des règles (34.2, incluant la quatrième voie bayésienne traitée au chapitre 35), par l'examen de la plateforme infrastructurelle (34.3), par la description du pipeline canonique qui les met en mouvement (34.4), par l'inventaire de ce qui reste ouvert (34.5), et par un dernier mot adressé au lecteur (34.6).

34.1. Cartographie de l'ouvrage

L'ouvrage s'est organisé en neuf parties, dont la cohérence ne se révèle pleinement qu'à rebours. La cartographie ci-dessous en donne la vue d'ensemble : chaque partie répond à une question distincte, et l'ordre dans lequel elles se succèdent n'est pas arbitraire — chacune suppose acquise la précédente.

Partie	Chapitres	Thème
I	1–3	fondements : histoire des paradigmes, théorie du neuro-symbolique, architecture VAILS
II	4–6	le langage : syntaxe et types, programmation fonctionnelle, orienté objet
III	7–10	le LLM comme co-processeur : opérateurs sémantiques, @, réification, §§
IV	11–13	le pont symbolique : base Prolog, système ontologique, opérateurs de Zalta
V	14–17	le pont neuronal : RAG, extraction structurée, Gherkin, induction de règles
VI	18–21	pipelines de production : oscillation formalisée et trois cas industriels
VII	22–27	autonomie, connaissance et optimisation : agents, atomes, expert, contraintes, génétique, ILP
VIII	28–34	la plateforme : méta-programmation, dialectes, modules, types, tests, plugins, synthèse

34.1.1. Trois grands arcs

Au-delà de la division formelle, l'ouvrage suit trois grands arcs, contigus mais de nature différente. Le premier arc — le langage — couvre les parties I et II. Il pose le contexte historique et théorique, l'architecture de VAILS, puis la grammaire de VVL : syntaxe et types, programmation fonctionnelle,

système objet. À l'issue de cet arc, VVL est un langage généraliste, homoiconique, mais encore muet sur le neuro-symbolique.

Le deuxième arc — *les deux versants — couvre les parties III à V, et c'est le cœur conceptuel du livre. La Partie III intègre le LLM comme co-processeur du langage : opérateurs sémantiques, génération de code par @, réification de concepts, compilation de questions par §§. La Partie IV pose le versant opposé — la base Prolog, le système ontologique, l'algèbre de Zalta — et la Partie V construit le pont qui alimente le symbolique depuis le neuronal : RAG, extraction structurée, ontologie dynamique par Gherkin, induction de règles. À l'issue de cet arc, l'oscillation neuro-symbolique n'est plus une promesse mais un mécanisme*.

Le troisième arc — *l'industrialisation — se déploie sur les parties VI à IX. Il fait passer le langage du prototype expressif au système de production. Oscillation formalisée et trois cas industriels (chapitres 18 à 21), agents autonomes et protocole d'atomes (22-23), systèmes experts et optimisation sous contraintes (24-25), apprentissage de règles par évolution et par induction logique (26-27), plateforme infrastructurelle (28-34), puis les extensions les plus récentes du runtime (35-40) : preuve formelle, logique relationnelle, exécution native et calcul distribué. T*outes ces étapes répondent à la même question : *comment faire pour qu'un programme VVL résiste au monde réel ?*

34.1.2. Le rôle particulier de la Partie VIII

Les sept chapitres de la Partie VIII (28-34) occupent une position singulière dans l'économie de l'ouvrage. Ils n'apportent pas de paradigme cognitif nouveau — ni nouveau type de représentation, ni nouvelle stratégie d'apprentissage — mais ils **achèvent** la chaîne formelle qui rend les paradigmes précédents utilisables ensemble. La méta-programmation (28) nomme la machinerie implicite qui sous-tendait les chapitres 12, 17 et 22. Les dialectes (29) formalisent l'ingestion d'entrées non-VVL — y compris les sorties de LLM. Les modules (30) donnent enfin une notion propre de bibliothèque. Les valeurs typées (31) consolident la validation au point de création. Les tests intégrés (32) ouvrent l'audit du comportement, en particulier celui des règles apprises. Les plugins signés (33) ouvrent la frontière externe sans renoncer aux garanties internes.

Cette partie est, en somme, la **clôture de la plateforme. Elle transforme ce qui était une promesse — VVL est un langage neuro-symbolique — en un fait opérationnel : VVL est une plateforme sur laquelle on peut construire et auditer des systèmes neuro-symboliques en production. Sans elle, les chapitres précédents resteraient des esquisses élégantes mais fragiles ; avec elle, ils deviennent les pièces d'une infrastructure réutilisable. La Partie IX qui suit ne remet pas cette clôture en cause : elle en expose les développements les plus récents**.

34.2. Les quatre paradigmes d'écriture des règles

Le concept central qui se dégage de la lecture intégrale de l'ouvrage est probablement le suivant : **VVL offre quatre façons distinctes d'écrire les règles formelles qui dirigent un système**. Trois de ces voies — manuelle, neurale, symbolique — sont stables et documentées dans les chapitres principaux. Une quatrième — bayésienne — fait l'objet du chapitre 35. Ces quatre paradigmes ne sont pas concurrents au sens strict : ils peuvent se composer dans un même programme. Ils répondent à des situations différentes et appellent des disciplines différentes. La figure ci-dessous donne la vue d'ensemble qui sera examinée dans cette section.

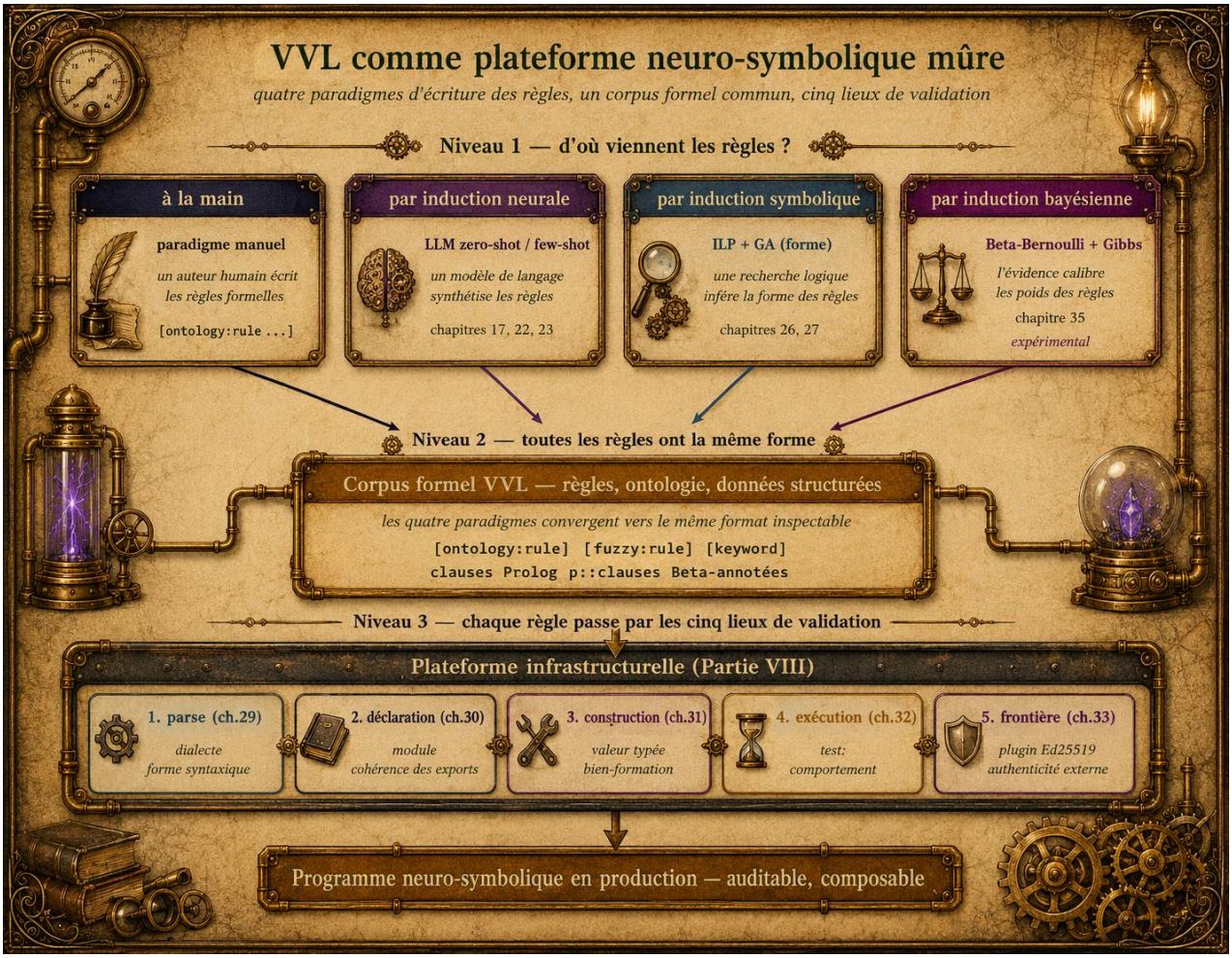


Figure 34.1 — *Les quatre paradigmes d'écriture des règles convergent vers un corpus formel commun, qui est ensuite traversé par les cinq lieux de validation de la plateforme infrastructurelle. La production en sortie est un programme neuro-symbolique auditable et composable.*

34.2.1. Paradigme manuel : un humain écrit les règles

Le premier paradigme est aussi le plus ancien dans l'histoire des langages déclaratifs. Un développeur — ou un expert métier accompagné — **écrit explicitement** les règles formelles qui gouvernent le système. En VVL, cela prend la forme d'expressions `[ontology:rule "name" when ... then ...]` saisies dans un script ou un module, complétées éventuellement par des règles floues `[fuzzy:rule]`, des keywords `[keyword]`, ou des contraintes de Horn directement encodées comme clauses Prolog.

Les chapitres 11 à 15 ont déroulé ce paradigme en profondeur. La force du paradigme manuel est l'**intelligibilité** : la règle écrite est lue et comprise par n'importe quel développeur compétent, sans dépendance à un mécanisme d'apprentissage opaque. Sa faiblesse est l'**effort cognitif** : l'auteur humain doit énoncer correctement chacune des règles, ce qui suppose une connaissance fine du domaine. Pour les systèmes où la connaissance experte existe et peut être verbalisée (logique médicale d'un protocole bien établi, règles fiscales, conventions normatives), le paradigme manuel est optimal.

34.2.2. Paradigme neural : un LLM synthétise les règles

Le deuxième paradigme tire parti d'une décennie de progrès en modèles de langage. Au lieu d'écrire les règles à la main, l'auteur **demande à un LLM** de les synthétiser à partir d'une description en langage naturel, d'exemples annotés, ou d'un corpus de référence. Les chapitres 17 (induction zero-

shot), 22 (agents autonomes) et 23 (systèmes experts compilés) ont chacun exploité cette voie selon des modalités légèrement différentes.

La force du paradigme neural est l'**économie d'effort** : une description en deux paragraphes peut produire vingt règles formelles, ce qui ouvre un débit de prototypage sans précédent. Sa faiblesse est l'**incertitude de génération** : le LLM produit parfois des règles syntaxiquement valides mais sémantiquement aberrantes, des règles correctes mais incomplètes, ou des règles redondantes. Cette incertitude est la raison même pour laquelle l'infrastructure de validation de la Partie VIII a été conçue — sans elle, le paradigme neural resterait un prototype divertissant mais inutilisable en production. Avec elle, il devient une voie sérieuse pour le démarrage rapide de systèmes experts.

34.2.3. Paradigme symbolique : une recherche logique infère les règles

Le troisième paradigme — historiquement le plus ancien dans la tradition de l'intelligence artificielle, mais redécouvert ces dernières années sous des formes nouvelles — fait **inférer** les règles à partir d'exemples par des techniques de recherche logique. VVL en expose deux variantes principales : les algorithmes génétiques du chapitre 26 et l'ILP sound-and-complete du chapitre 27.

La force du paradigme symbolique est la **garantie de correction** : un algorithme d'ILP correct produit des règles qui couvrent rigoureusement les exemples positifs et excluent rigoureusement les négatifs, ce qui n'est pas une propriété qu'un LLM offre. Sa faiblesse est le **coût computationnel** : la recherche dans l'espace des hypothèses peut être prohibitive pour les domaines de grande dimension. Le paradigme symbolique trouve naturellement sa place pour les systèmes où la garantie formelle est essentielle (audit médical, conformité réglementaire) et où le volume d'exemples reste maîtrisable.

34.2.4. Paradigme bayésien : l'évidence calibre les poids

Le quatrième paradigme — exposé en détail au chapitre 35 — adopte une perspective radicalement différente des trois précédents. Plutôt que d'apprendre la **forme** des règles, il suppose les règles déjà données et apprend leur **poids** : la probabilité qu'elles se déclenchent dans le monde. Cette inversion — *forme connue, poids inconnu* — appelle des outils mathématiques distincts, en l'occurrence la conjugaison Beta-Bernoulli et l'échantillonnage de Gibbs.

La force du paradigme bayésien est la **calibration de l'incertitude** : il ne se contente pas d'apprendre un poids moyen, il porte une *croyance* sur ce poids, qui se précise à mesure que l'évidence s'accumule. Cette capacité à distinguer l'incertitude *aléatoire* (un phénomène intrinsèquement variable) de l'incertitude *épistémique* (notre ignorance d'un paramètre fixe) ouvre des possibilités d'audit que les trois paradigmes précédents ne couvrent pas. Sa contrepartie est le coût : l'échantillonnage de Gibbs suppose des atomes-requêtes observés en nombre suffisant, et la négation exacte se paie par une énumération des mondes qui borne la taille des programmes traitables. Le paradigme bayésien trouve sa place quand les règles sont stables mais que les poids doivent s'adapter à un domaine particulier ; il complète, plutôt qu'il ne remplace, les voies symbolique et neurale.

34.2.5. Comparaison synthétique

Pour fixer définitivement les quatre paradigmes, le tableau ci-dessous résume leurs propriétés sur six axes critiques.

Axe	Manuel (ch.11-15)	Neural (ch.17,22,23)	Symbolique (ch.26-27)	Bayésien (ch.35)
Apprend	rien (écrit)	forme + plausibilité	forme (sound-complete)	poids + incertitude
Effort de saisie	élevé	faible	modéré	faible (priors par défaut)
Garantie	celle de l'auteur	aucune	sound-and-complete (ILP)	IC épistémique
Intelligibilité	maximale	variable	élevée	élevée + confiance graduée
Coût d'exécution	nul	appel LLM par règle	recherche, parfois lourde	Gibbs, ~ms à ~s
Cas d'usage type	connaissance experte	prototypage rapide	audit formel	calibration sous évidence

34.2.6. La voie hybride

L'expérience montre que les **systèmes réels** combinent les paradigmes plutôt que de choisir l'un d'eux. Un système expert médical typique peut commencer par un **prototype LLM** (paradigme neural) qui produit une première version des règles à partir d'une revue de la littérature, suivi d'une **vérification manuelle** par un expert humain (paradigme manuel) qui révise les règles produites, puis d'un **audit symbolique** par ILP sur un jeu d'exemples cliniques (paradigme symbolique) qui valide la couverture et la précision, et enfin — pour les systèmes qui en disposent — d'une **calibration bayésienne** (chapitre 35) qui ajuste les poids des règles aux observations recueillies en production. À chacune des étapes, les règles ont la même forme — `[ontology:rule]`, clauses Prolog, ou `p::clause` annotée —, ce qui permet la transition fluide entre paradigmes sans réécriture.

Cette voie hybride n'a rien d'utopique : elle est régulièrement empruntée dans les projets industriels qui utilisent VVL en production. Le langage rend cette hybridation possible précisément parce qu'il offre **un seul format** pour les quatre paradigmes. Aucune passerelle entre représentations, aucune compilation entre formalismes : la règle est, à la fin, ce qu'elle a toujours été — une expression VVL inspectable par un humain, vérifiable par un test, sérialisable par `[dump]`, et désormais annotable par une croyance probabiliste. Cette unité de représentation est probablement la propriété la plus précieuse du langage.

34.3. La plateforme : sept strates en cohérence

La Partie VIII a, en sept chapitres, exposé l'**infrastructure** qui rend les trois paradigmes utilisables ensemble. Mais l'infrastructure elle-même n'est pas un monolithe : elle se laisse décrire comme un empilement de sept strates, chacune s'appuyant sur les précédentes. Cette section restitue la vue stratifiée pour fixer définitivement l'image.

Strate	Apport principal	Chapitre
1. langage de base	syntaxe, types primitifs, structures de contrôle	I-II (1-10)
2. méta-programmation	code-comme-donnée, evaluate, quote, load, dump	28
3. dialectes	grammaire-comme-donnée, ingestion de DSL	29
4. modules	namespaces, encapsulation, frontières de package	30
5. valeurs typées	validation scalaire à la construction	31

Strate	Apport principal	Chapitre
6. tests intégrés	validation comportementale, audit des règles apprises	32
7. plugins signés	extension externe authentifiée par Ed25519	33

34.3.1. Lecture verticale des strates

Chaque strate **présuppose** les précédentes et **enrichit** ce qu'elles offrent. La strate 1 — le langage de base — est la fondation : sans elle, rien n'existe. La strate 2 — la méta-programmation — ne devient utile qu'une fois qu'on dispose de structures à manipuler ; elle s'appuie donc sur la strate 1, qu'elle expose réflexivement. La strate 3 — les dialectes — utilise la machinerie réflexive de la strate 2 pour exécuter des grammaires-comme-données ; elle ne pourrait pas exister sans elle. La strate 4 — les modules — utilise la même machinerie pour encapsuler des espaces de noms ; elle s'appuie sur 1, 2, et indirectement sur 3 pour ses besoins de parsing internes.

Les strates 5, 6 et 7 s'élèvent au-dessus de cet édifice. Les valeurs typées (5) sont des constructeurs natifs qui pourraient théoriquement être implémentés via dialectes + ontologie, mais qui méritent leur strate propre pour des raisons d'ergonomie et de performance. Les tests intégrés (6) consomment toutes les strates précédentes : ils valident le comportement de modules construits avec valeurs typées et dialectes, eux-mêmes écrits avec la machinerie méta-programmatique. Les plugins (7), enfin, étendent l'ensemble vers l'extérieur, en ajoutant une frontière cryptographique qui protège l'écosystème des intrusions non authentifiées.

34.3.2. Lecture horizontale : ce que chaque strate garantit

Au-delà de l'empilement vertical, chaque strate offre une **garantie horizontale** qui s'ajoute à celle des autres. Énoncer ces garanties est le moyen le plus rapide de saisir ce que la plateforme apporte concrètement.

Strate	Garantie offerte
1. langage de base	évaluation déterministe d'expressions bien-formées
2. méta-programmation	tout fragment de code est manipulable comme donnée
3. dialectes	toute chaîne entrante peut être validée contre une grammaire VVL
4. modules	aucune définition n'écrase silencieusement une autre
5. valeurs typées	aucune valeur scalaire ne circule sans validation à sa création
6. tests intégrés	aucun comportement attendu ne reste non-vérifiable
7. plugins signés	aucun code externe ne s'exécute sans authentification cryptographique

Lue de bout en bout, cette liste constitue la **promesse formelle** de la plateforme VVL. Un système qui couvre les sept strates dispose, sans effort additionnel, des sept garanties cumulées. C'est ce cumul qui distingue VVL d'environnements où l'on doit assembler manuellement un cadre de validation à partir de bibliothèques disparates. Ici, la cohérence est par construction.

34.4. Le pipeline canonique

Pour rendre concrète la mise en mouvement des trois paradigmes au-dessus des sept strates, on peut décrire ce qu'on appelle, dans la pratique des projets, le **pipeline canonique** d'un système neuro-symbolique VVL. Ce pipeline n'est pas une obligation — chaque projet ajuste —, mais il représente le déroulement typique d'un système qui ingère, raisonne, et produit.

34.4.1. Description du pipeline

Le pipeline canonique comporte sept étapes, dont chacune mobilise une portion identifiable de la plateforme.

Étape 1 — ingestion. L'entrée arrive sous forme de texte (sortie LLM, formulaire utilisateur, message API, document scanné). Un dialecte (chapitre 29) la consomme avec une grammaire qui isole les fragments significatifs. Les captures résultantes sont des chaînes ou des listes.

Étape 2 — promotion en valeurs typées. Chaque fragment capturé qui correspond à un scalaire validable est passé à son constructeur typé : `[EMAIL ...]`, `[URL ...]`, `[DATE ...]`, `[UUID ...]` (chapitre 31). Les fragments invalides lèvent une erreur, interceptée par `[attempt]`.

Étape 3 — assemblage en concept ontologique. Les valeurs typées sont assemblées en un objet structuré, qui est asserté dans l'ontologie courante via `[ontology:assert]` (chapitre 12). Le concept est désormais un fait formel manipulable par le moteur d'inférence.

Étape 4 — inférence. Le moteur d'inférence applique les règles présentes dans le contexte — qu'elles aient été écrites à la main, induites par LLM, ou synthétisées par ILP. Les nouvelles conclusions sont elles-mêmes assertées comme nouveaux faits, qui peuvent à leur tour déclencher d'autres règles. La boucle converge.

Étape 5 — décision et action. Le programme principal lit les conclusions de l'inférence et décide quoi faire : notifier un utilisateur, déclencher un workflow externe, persister un état. Si l'action exige une fonctionnalité non disponible nativement — appel à un service ML, requête HTTP signée, manipulation de fichier propriétaire —, un **plugin signé** (chapitre 33) est invoqué pour effectuer l'opération en sous-processus Python.

Étape 6 — audit. Avant de mettre en production, ou périodiquement pendant l'exploitation, le programme exécute une **suite de tests** (chapitre 32) qui rejoue des cas connus pour vérifier que le comportement reste conforme. Tout écart est détecté immédiatement et signalé.

Étape 7 — persistance. Les états importants — concepts ontologiques, règles induites, configurations — sont sérialisés via `[dump]` (chapitre 28) au format `<TYPE:payload>` (chapitre 31), écrits dans la base de connaissance (chapitre 19), et peuvent être rechargés à l'identique en début de session.

34.4.2. Un exemple concret de bout en bout

Pour fixer le pipeline sur un cas concret, déroulons un exemple : un système de **triage médical** qui reçoit des fiches patient en texte libre et produit une recommandation de priorité de prise en charge. Ce système combine les trois paradigmes et traverse les sept strates.

```
# Etape 1 : un dialecte pour ingerer la fiche patient

[dialect:def fichePatient
    [[capture nom    [some letter]]   space
     [capture age    [some digit]]    space
     [capture symp   [some any]]      end]
]

$saisie = [dialect:parse fichePatient
    "Dupont 45 fievre persistante"]

```

```
# Etape 2 : promotion en valeurs typees + concept ontologique

[ontology:define Patient [name age symptom]]
$p = [ontology:assert [Patient
    [name    $saisie.captures.nom]
    [age     (+ 0 $saisie.captures.age)]
    [symptom $saisie.captures.symp]
]]

# Etape 3 : les regles ont ete induites par LLM puis revisees
#           a la main (paradigme hybride)
#
# [ontology:rule "haute-priorite"
#     when ($p ~~ "Patient") and ($p.age > 65)
#                            and ($p.symptom ~~ "fievre")
#     then ($p.priorite = "haute")]
#
# [ontology:rule "moyenne-priorite"
#     when ($p ~~ "Patient") and ($p.symptom ~~ "fievre")
#                            and (! ($p.age > 65))
#     then ($p.priorite = "moyenne")]

# Etape 4 : inference automatique
[ontology:infer]

# Etape 5 : decision et action via plugin
[plugin-load "hospital-bridge" as hospital:notify]
[hospital:notify [object
    [patient_id $p.id]
    [priority   $p.priorite]
]]

# Etape 6 : audit (a executer regulierement)
[test:suite "regles-triage"
    [test:equals "moyenne"
        [evaluate-triage [object [age 45] [symptom "fievre"]]]
        "cas adulte avec fievre"]
    [test:equals "haute"
        [evaluate-triage [object [age 72] [symptom "fievre"]]]
        "cas age avec fievre"]
]

# Etape 7 : persistance
[kb:append "patients-traites" [dump $p]]
```

Cet exemple — volontairement simplifié — montre que toutes les pièces de la plateforme participent à un même pipeline cohérent. Le dialecte ingère ; les valeurs typées valident ; l'ontologie structure ; les règles inférent ; le plugin agit ; les tests auditent ; la base de connaissances persiste. Aucune ligne n'est de la *plomberie* — chacune contribue à une fonction métier identifiable.

34.4.3. La densité d'expression

Un point qui mérite d'être souligné : l'exemple ci-dessus tient en moins de quarante lignes, et il couvre **les sept strates** et **trois des cinq lieux de validation** (parse via dialecte, construction via Patient/valeurs typées, exécution via tests). Cette densité d'expression est ce qui rend VVL économique à utiliser en pratique. Un système équivalent écrit en Python aurait probablement dépassé deux cents lignes — sans même inclure les vérifications de validation, qui auraient été soit absentes, soit dispersées.

Cette économie n'est pas un raccourci au détriment de la rigueur. Au contraire : c'est précisément parce que les vérifications sont **intégrées au langage** (dialecte qui valide à l'ingestion, type qui valide à la construction, test qui valide le comportement) qu'on n'a pas besoin de les écrire à chaque fois. Le programme VVL court parce que le langage prend en charge ce qui, ailleurs, devrait être réécrit.

34.5. Limites et perspectives

Un bilan honnête doit aussi nommer ce qui reste **hors du périmètre actuel** de la plateforme. Cette section dresse l'inventaire des limites connues et des directions d'évolution. La liste n'est pas exhaustive — les listes de roadmap ne le sont jamais —, mais elle couvre les points qui reviennent le plus fréquemment dans les discussions des utilisateurs avancés.

34.5.1. Limites du MVP infrastructurel

Chacun des chapitres 28 à 33 a énuméré ses limites propres. On peut les rappeler en quelques lignes synthétiques. La méta-programmation (28) ne fournit pas d'outils de transformation d'arbre de haut niveau — tout passe par les opérations de liste standard. Les dialectes (29) sont limités à un MVP sans backtracking complet ni captures arborescentes. Les modules (30) n'offrent pas encore de syntaxe `from import` ni de mécanisme de versioning. Les valeurs typées (31) sont au nombre de quatre — `EMAIL`, `URL`, `DATE`, `UUID` — sans extension utilisateur. Les tests intégrés (32) n'incluent ni mocks natifs ni paramétrisation. Les plugins signés (33) n'offrent pas de sandbox OS véritable sur Windows, et le mécanisme d'allowlist réseau pour les plugins Python repose sur l'usage conventionnel du SDK officiel.

Aucune de ces limites n'est rédhibitoire pour les usages courants — l'expérience montre que le MVP couvre l'écrasante majorité des besoins pratiques. Mais aucune n'est non plus une décision définitive : chacune représente un trade-off entre simplicité et complétude, et chacune peut être levée dans une version ultérieure si la communauté en exprime le besoin.

34.5.2. Limites des paradigmes apprenants

Les trois paradigmes d'écriture des règles ont, eux aussi, leurs limites. Le paradigme manuel souffre d'une **scalabilité limitée** : un humain ne peut pas, en pratique, écrire et maintenir plus de quelques centaines de règles formelles, et les bases de règles très volumineuses deviennent inintelligibles. Le paradigme neural souffre d'une **incertitude résiduelle** : même avec validation a posteriori, un LLM peut produire des règles plausibles mais subtilement fausses, et la détection de ces erreurs subtiles dépend de la qualité des tests d'audit. Le paradigme symbolique souffre d'une **explosion combinatoire** : l'ILP sound-and-complete peut devenir prohibitif au-delà de quelques dizaines d'attributs et quelques milliers d'exemples.

Une direction de recherche prometteuse est l'**hybridation profonde** des trois paradigmes : un LLM qui *propose* des règles candidates, un module ILP qui les *vérifie* contre un jeu d'exemples, et un humain

qui *arbitre* sur les cas ambigus. Les premières expériences sur cette boucle suggèrent qu'elle combine les vertus des trois sources tout en limitant leurs vices respectifs. La plateforme actuelle de VVL offre toute l'infrastructure pour mener ces expériences ; la formalisation théorique reste ouverte.

34.5.3. Limites cognitives plus profondes

Au-delà des limites techniques, deux limites cognitives méritent d'être nommées avec honnêteté. **Première limite** : VVL est un langage de **raisonnement déductif** — il excelle dans la chaîne *prémisses → conclusions* mais ne prétend pas formaliser le raisonnement abductif (chercher la meilleure explication d'un fait observé) ni le raisonnement contrefactuel (que se passerait-il si...). Pour ces formes de raisonnement, les outils du langage permettent des approximations utiles, mais une formalisation native reste à construire.

Deuxième limite : VVL est un langage de **représentation symbolique**. Il représente bien les connaissances qui se laissent encoder en faits et règles. Il représente mal les connaissances *implicites*, *tacites*, *gestuelles*, *perceptuelles*. Pour ces dernières, l'intégration avec des modèles neuraux (par plugin) reste la voie naturelle, mais le langage symbolique ne *contient* pas la connaissance neurale — il la *consulte*. Cette frontière n'est pas un défaut accidentel mais une **propriété structurelle** du paradigme symbolique en général. La reconnaître clairement permet d'éviter les promesses excessives qui ont, par le passé, déconsidéré l'intelligence artificielle symbolique.

34.5.4. Directions de recherche

Plusieurs directions de recherche se dessinent au-delà du MVP actuel. **Direction 1 — l'ILP+LLM hybride** : un module qui ferait coopérer la garantie symbolique d'ILP avec la créativité génératrice d'un LLM, par exemple en utilisant le LLM pour proposer un ordre d'exploration des hypothèses qu'ILP raffine ensuite formellement. **Direction 2 — le test-driven learning** : prendre au sérieux la suggestion de la section 32.9.4 et faire des suites de tests `test:*` l'oracle direct d'un algorithme d'apprentissage, qui itère jusqu'à passer tous les tests. **Direction 3 — la composition compositionnelle de modules apprenants** : permettre à un module qui contient des règles induites de **déclarer son incertitude** et de propager cette information à ses appelants, de façon à ce que la chaîne complète d'inférence reste interprétable en termes probabilistes.

Ces directions sont, pour la plupart, des **questions ouvertes** : il n'existe pas, à ce jour, de consensus sur la bonne forme à leur donner. Le présent ouvrage ne les résout pas. Il offre simplement l'**infrastructure** sur laquelle ces explorations peuvent se mener — un langage cohérent, un format de représentation unifié, des cinq lieux de validation, un mécanisme propre d'extension externe. La suite appartient aux utilisateurs.

34.6. Au lecteur — le mot final

Le programme principal de cet ouvrage touche à sa fin. Trente-quatre chapitres dans la trame canonique, prolongés par six chapitres qui couvrent les extensions les plus récentes du runtime, plusieurs centaines de builtins, plus de trois cents figures et tableaux : l'inventaire matériel est imposant. Mais l'inventaire ne dit pas ce que le livre **prétend** vraiment. Quelques phrases pour le dire.

34.6.1. Ce que le livre prétend

Le livre prétend, en premier lieu, **un fait technique** : VVL existe, il est utilisable, et il couvre un ensemble fonctionnel cohérent. Cette prétention est vérifiable : un lecteur qui télécharge le runtime,

qui ouvre une session, qui suit les exemples du livre — ce lecteur retrouvera, à peu de chose près, les comportements décrits. Cette prétention factuelle est la moins ambitieuse mais aussi la plus solide.

Le livre prétend, en second lieu, **une thèse architecturale** : la programmation neuro-symbolique gagne à reposer sur **un seul langage** qui formalise les règles, plutôt que sur une fédération hétérogène de cadres techniques. Cette thèse est plus contestable. Beaucoup d'écosystèmes contemporains ont fait le choix inverse — utiliser Python comme glu, et déléguer la logique symbolique à des bibliothèques externes (CLIPS, Prolog, Datalog). Le livre soutient que ce choix produit des systèmes fragiles, où la traçabilité formelle se perd dans les jointures entre paradigmes ; et que VVL, en hébergeant les quatre paradigmes d'écriture des règles dans un même format inspectable, propose une alternative plus saine. Le lecteur en jugera.

Le livre prétend, en troisième lieu, **une promesse d'usage** : la programmation neuro-symbolique n'est pas réservée aux chercheurs ; elle est accessible à tout développeur compétent, pour autant qu'il dispose d'un langage à la hauteur. Cette promesse est plus ouverte. Elle implique un pari sur le lecteur — qu'il prendra le temps d'apprendre, qu'il acceptera les disciplines de validation, qu'il osera l'exigence formelle. Le livre a essayé, page après page, de rendre ce pari raisonnable. Mais c'est, en définitive, au lecteur de le confirmer.

34.6.2. Trois invitations

Plutôt qu'une exhortation générique, on offrira au lecteur trois invitations concrètes — qu'il pourra, ou non, accepter.

Première invitation : prenez **un problème réel** que vous avez sous la main — un script de traitement de données, un outil d'audit, un système de recommandation, une intégration API — et essayez de le reformuler en VVL. Vous découvrirez probablement que certaines parties s'écrivent plus naturellement qu'en Python ; que d'autres exigent un effort d'apprentissage ; et que quelques-unes sortent du domaine pour lequel le langage est conçu. Cette exploration vous donnera une carte personnelle — bien plus utile que toutes les promesses du présent chapitre.

Deuxième invitation : choisissez **un paradigme** parmi ceux que l'ouvrage présente — manuel, neural, symbolique, ou bayésien avec le chapitre 35 — sur lequel vous n'avez encore aucune expérience, et écrivez avec lui le système le plus simple possible : un classifieur binaire, un parseur de format propriétaire, un assistant de réponse à une question fréquente. Mesurez ensuite, en lignes de code et en effort cognitif, ce que ce paradigme vous a coûté par rapport à ce qu'il vous a rapporté. Cette mesure individuelle est ce qui fait, à terme, qu'un développeur choisit ses outils en connaissance de cause.

Troisième invitation : participez à la **discussion publique** autour de VVL. Le langage évolue. Les limites mentionnées au chapitre 34.5 ne sont pas figées ; les directions de recherche restent ouvertes. Un retour d'expérience d'utilisateur — qu'il soit enthousiaste, mitigé, ou critique — pèse plus que les opinions des concepteurs. La plateforme aujourd'hui décrite est ce qu'elle est ; ce qu'elle deviendra dépend en partie de la communauté qui l'utilisera.

34.6.3. Une dernière note — et un pont vers la Partie IX

L'auteur de ce livre a passé plusieurs années à concevoir et à éprouver VVL avant de pouvoir l'exposer dans la forme stabilisée que cet ouvrage présente. Il a, comme tous les concepteurs de langage, fait des choix qu'il regrette parfois, des choix qu'il défendra toujours, et des choix qu'il reverra peut-être un jour. Le résultat — la plateforme telle qu'elle est aujourd'hui — n'est pas le meilleur des résultats

possibles ; c'est un résultat qui essaie d'être **cohérent**, qui prend au sérieux la programmation neuro-symbolique comme **discipline industrielle**, et qui offre au lecteur **un outil** plutôt qu'un manifeste.

À cette plateforme s'ajoutent six chapitres qui prolongent l'ouvrage vers l'état courant du runtime. Le chapitre 35 apporte une quatrième voie d'apprentissage, l'annotation stochastique bayésienne, qui apprend les poids des règles plutôt que leur forme et complète l'arsenal des chapitres 17, 26 et 27 par la dimension probabiliste qui manquait. Le chapitre 36 introduit le calcul symbolique exact. Les chapitres 37 à 40 couvrent la preuve formelle native, la logique relationnelle miniKanren, la machine virtuelle Rust et le cluster de calcul distribué. Tous décrivent des sous-systèmes livrés dans le serveur courant, avec leurs limites explicitement documentées.

Si ce livre l'a accompagné jusqu'ici, et s'il en ressort avec, ne serait-ce qu'une fois, l'envie d'essayer un fragment de VVL sur un problème qui lui est propre, alors la Partie VIII aura accompli sa fonction. Le reste — l'élargissement de la plateforme, l'évolution des paradigmes, l'invention de cas d'usage que personne n'avait anticipés — appartient à ceux qui prendront le langage en main et qui en feront ce qu'eux seuls peuvent en faire.

Ici s'achève la trame canonique de l'ouvrage. La Partie IX qui s'ouvre au chapitre suivant en est le prolongement direct : six chapitres consacrés aux sous-systèmes que le runtime a acquis depuis, et qui ne modifient rien de ce qui précède — ils l'étendent. Le lecteur pressé peut s'arrêter ici sans rien manquer d'essentiel ; le lecteur qui veut voir jusqu'où va la plateforme aujourd'hui poursuivra jusqu'à la conclusion générale, qui referme l'ensemble après le chapitre 40.



Partie IX — Extensions du runtime

Chapitres 35 à 40 — ce que le serveur a acquis depuis

35. Annotation stochastique bayésienne — la quatrième voie d'apprentissage

Mots-clés : *bayésien, Beta-Bernoulli, croyance, prior, posterior, conjugaison, Gibbs, incertitude aléatoire, incertitude épistémique, négation exacte, propagation, label, mise à jour séquentielle, additif, bfacts, brules, bannotate, blearn, bprob, bapprox, bposterior, bbeliefs*

Périmètre. Ce chapitre présente la couche probabiliste du serveur courant. Les builtins préfixés b* décrits ici n'apparaissent dans aucun des chapitres précédents : ils forment une couche strictement additive, posée sur la base de connaissances Prolog sans la modifier. Un fichier .vkb antérieur se recharge en Prolog booléen pur, et les croyances vivent dans une table latérale. Le chapitre s'adresse à deux lectorats : les praticiens qui veulent exploiter la couche probabiliste, et les lecteurs qui souhaitent comprendre la direction de recherche prolongeant la trilogie d'apprentissage des chapitres 17, 26 et 27.

Le chapitre 34 avait fermé l'ouvrage sur trois paradigmes d'écriture des règles : manuel, neural, symbolique. Ce chapitre additionnel — écrit après le mot final, et qui reprend ce mot final à sa propre fin — introduit une **quatrième voie** : l'annotation stochastique bayésienne. Plus précisément, une **couche probabiliste additive** posée sur le mini-Prolog VAILS, qui transforme chaque clause en une *croyance* sur son poids plutôt qu'en un poids figé, et qui apprend ces croyances depuis des observations par échantillonnage de Gibbs.

Le mot *bayésien* effraie souvent dans la littérature des langages : il évoque des formalismes mathématiques denses, des temps de calcul prohibitifs, des modèles graphiques opaques. On va voir, au fil du chapitre, que la version présentée ici reste maniable. Huit builtins suffisent à exposer la fonctionnalité ; un modèle génératif à quatre niveaux suffit à en formaliser la sémantique ; un algorithme à deux pas — Gibbs E/M — suffit à en assurer l'apprentissage. La modestie des moyens est ce qui rend la couche **utilisable** plutôt que spéculative.

35.1. Position du module : couche probabiliste additive

Avant d'examiner les huit builtins en détail, il vaut la peine de positionner précisément ce que le module `bayes/` est, et ce qu'il n'est pas. La précision de cette mise en place conditionne la lecture de tout le reste du chapitre.

35.1.1. Ce qu'apporte le module

La base de connaissance Prolog présentée au chapitre 11, et étendue depuis par les mécanismes d'induction du chapitre 27, est **booléenne** : une clause est vraie ou absente. Cette dichotomie convient parfaitement aux faits certains et aux règles déterministes, mais elle ne sait pas représenter une notion essentielle dans les systèmes réels : l'**incertitude graduée**. Une règle métier qui se vérifie *« la plupart du temps »*, un fait extrait par un LLM avec un logprob moyen, une corrélation observée mais non garantie — toutes ces formes de connaissance sont *probables sans être certaines*, et le langage Prolog nu n'a aucun moyen propre de les exprimer.

Le module `bayes/` ajoute exactement cette dimension manquante. À chaque clause Prolog, on peut désormais associer une **croyance** probabiliste sur son poids de déclenchement. Cette croyance peut

être déclarée à l'écriture (par l'auteur humain), héritée d'une source de provenance (fait ancré, sortie LLM, saisie utilisateur), ou *apprise* depuis des observations par un algorithme dédié. Dans tous les cas, la clause reste **lisible et formellement identique** à une clause Prolog ordinaire ; seul son traitement par le moteur d'évaluation diffère.

35.1.2. Ce qui reste inchangé

La doctrine d'extension du module est rigoureusement **additive** : rien dans la couche Prolog booléenne n'est modifié. Les fichiers `prolog.py` et `vails.vvl_prolog.PrologEngine` du runtime ne sont **pas touchés**. Les croyances vivent dans une *table latérale* indexée par la signature des clauses ; un programme qui n'utilise aucun builtin `b*` voit le runtime se comporter exactement comme avant. Une base de connaissance sauvegardée *avant* la disponibilité du module bayésien se recharge **sans modification** dans la version étendue ; la table latérale est simplement vide, et la base se comporte comme du Prolog booléen pur.

Cette discipline d'extension est ce qui justifie la qualification de *couche additive*. À aucun moment l'utilisateur n'est forcé d'adopter le formalisme probabiliste ; il l'invoque s'il en a besoin et il l'ignore autrement. Une clause non annotée garde une probabilité implicite de 1.0 et n'apparaît dans aucune des structures de croyance que les builtins `b*` exposent.

35.1.3. Place dans l'ouvrage

Le chapitre 34 — qui était initialement la conclusion de l'ouvrage — a été rouvert pour mentionner ce quatrième paradigme dans sa récapitulation. Le tableau des paradigmes d'écriture des règles, que la version originale du chapitre listait en trois colonnes (manuel, neural, symbolique), comporte désormais une quatrième colonne : le **paradigme bayésien**, qui apprend des **poids** plutôt que des **formes**. Cette distinction est subtile mais essentielle, et nous y reviendrons à la section 35.11 quand l'ensemble des mécanismes aura été présenté.

35.2. Le glissement : du poids scalaire à la croyance Beta

Le concept central du module `bayes/` se résume en une phrase, que nous allons déplier soigneusement : *une clause ne porte pas un poids $p \in [0,1]$ figé, mais une croyance sur ce poids*. Ce glissement — d'une valeur scalaire à une distribution sur les valeurs scalaires — est ce qui permet de représenter, dans le même cadre formel, deux incertitudes que les approches naïves confondent : l'incertitude *aléatoire* et l'incertitude *épistémique*.

35.2.1. La limite de l'approche scalaire

Les langages probabilistes classiques — ProbLog, PRISM, certaines extensions de Prolog — attachent un poids scalaire à chaque clause : `0.8 :: pluie.` signifie *« la clause pluie se déclenche avec probabilité 0.8 »*. Le moteur sait alors calculer la probabilité de n'importe quel but dérivé. Cette approche est puissante et bien comprise, mais elle souffre d'une limite conceptuelle : *elle traite le poids comme une donnée connue, alors qu'il est souvent lui-même incertain*.

Considérons deux clauses, toutes deux annotées `p = 0.8`. La première vient d'une étude statistique massive : on a observé que pluie se produit 80 % du temps dans la région concernée, sur dix ans d'enregistrements. La seconde vient d'un LLM qui, sur une question vague, a renvoyé « *je dirais environ 0.8* ». Le scalaire `0.8` est le même dans les deux cas, mais notre **confiance dans ce scalaire** est radicalement différente. L'approche scalaire ne sait pas représenter cette différence.

35.2.2. La solution : porter une distribution

La solution adoptée par le module `bayes/` est de remplacer le scalaire `p` par une **distribution Beta** sur les valeurs possibles de ce poids :

```
clause i :  θ_i  ~  Beta(α_i, β_i)

#  ϑ_i        : le "vrai" poids de la clause (inconnu)
#  Beta(α,β)  : notre croyance sur sa valeur
#  mean       = α / (α + β)
#  concentration = α + β   ("force" de la croyance)
```

La distribution Beta a deux paramètres positifs : α (alpha) et β (beta). Sa moyenne vaut `α / (α + β)` — c'est ce que l'approche scalaire aurait nommé `p`. Mais la Beta porte une information supplémentaire qu'aucun scalaire ne peut capturer : sa **concentration**, mesurée par `α + β`. Une concentration faible (disons 2) signifie *« je ne suis pas sûr »* ; une concentration forte (disons 100) signifie *« j'en suis pratiquement certain »*. Avec la même moyenne `0.8`, on peut donc avoir Beta(8, 2) [modérément sûr] ou Beta(80, 20) [très sûr].

35.2.3. Pourquoi Beta et pas une autre distribution

Le choix de la distribution Beta n'est pas arbitraire. Trois propriétés en font le candidat naturel pour porter une croyance sur une probabilité.

Première propriété — le support. La Beta est définie sur l'intervalle `[0, 1]`, exactement le support des probabilités. Aucune transformation ni reparamétrage n'est nécessaire. **Deuxième propriété — la conjugaison.** La Beta est la *conjuguée* de la Bernoulli, ce qui veut dire que la mise à jour bayésienne d'un prior Beta par une observation Bernoulli donne un posterior qui est *aussi* une Beta, avec des paramètres mis à jour par simple addition. Cette conjugaison rend l'apprentissage **closed-form** — exact, pas approximatif. **Troisième propriété — la flexibilité.** En faisant varier α et β, la Beta peut prendre toutes les formes utiles : uniforme (Beta(1,1)), piquée à gauche, piquée à droite, en cloche centrée, en U bimodal. Elle couvre l'intégralité de l'éventail des croyances pratiques.

35.2.4. Aléatoire vs épistémique : la distinction enfin nommée

L'avantage le plus important de la croyance Beta sur le scalaire est qu'elle sépare proprement deux incertitudes que la littérature scientifique distingue depuis longtemps mais que les langages probabilistes confondent souvent : l'incertitude **aléatoire** et l'incertitude **épistémique**.

L'incertitude *aléatoire* est la part inhérente au phénomène lui-même : un événement se produit X % du temps, et même avec une connaissance parfaite, le tirage individuel reste incertain. C'est ce que mesure la **moyenne** de la Beta — `α / (α + β)`. Une moyenne de 0.5 signifie un phénomène intrinsèquement 50/50.

L'incertitude *épistémique* est la part qui dépend de notre **ignorance** : on ignore le vrai taux, et plus on observe, plus on le précise. C'est ce que mesure la **concentration** de la Beta — `α + β`. Une concentration faible (Beta plate) signifie une grande ignorance ; une concentration forte (Beta piquée) signifie une connaissance solide.

L'exemple canonique fixe la distinction : Beta(2, 2) et Beta(200, 200) ont la même moyenne 0.5, mais l'une est **ignorante** (on n'a rien observé qui distingue plus de 0.5 d'autre chose) tandis que l'autre est

certaine (200 observations confirment que c'est vraiment 50/50). Aucun scalaire ne peut distinguer ces deux cas, ce qui justifie à lui seul l'introduction du formalisme bayésien.

35.3. Cinq sources de priors

Si chaque clause porte une Beta(α, β), il faut bien décider, au moment de l'annotation, *quels α et β* lui attribuer. Le module `bayes/` adopte pour cela une discipline simple et productive : la **provenance de la clause** détermine la forme de son prior. Concrètement, l'auteur fournit la *moyenne* (le `p` qu'il visait) et la *source*, et le module calcule la concentration appropriée. La figure ci-dessous donne les cinq sources prédéfinies.



Figure 35.1 — *Les cinq sources de priors et leurs formes caractéristiques de Beta. L'aire sous chaque courbe est 1 ; la hauteur du pic mesure la concentration. La provenance d'une clause détermine la force du prior associé.*

35.3.1. Le tableau des cinq sources

Pour fixer la sémantique, voici le tableau récapitulatif des cinq sources, avec leurs hyperparamètres et leur interprétation.

Source	Prior typique	Interprétation
fact	Beta(~38, ~2)	fait ancré, quasi-certain, coûte cher à déloger
user	concentration 20	affirmé par l'utilisateur, déplaceable par l'évidence
LLM	concentration variable	calibrée par le logprob de la réponse du modèle
?::clause	Beta(1, 1)	ignorance totale, prior plat, « à apprendre »
learned	posterior d'un apprentissage	posterior recyclé en prior (séquentiel)

35.3.2. La logique d'élicitation

L'élicitation suit une intuition simple : *plus la provenance est fiable, plus la concentration est forte*. Un fait certifié (`fact`) se voit attribuer une concentration de l'ordre de 40, ce qui produit une Beta très piquée — il faudra de très nombreuses observations contradictoires pour la déplacer. Une saisie utilisateur (`user`) obtient une concentration de 20, plus modérée : l'utilisateur peut se tromper, et l'évidence aura raison de son intuition si elle diverge.

Une règle produite par un LLM (`LLM`) reçoit une concentration **proportionnelle au logprob** de la réponse : si le modèle était très confiant, le prior est plus piqué ; s'il hésitait, le prior est plat. Cette calibration réutilise la fonction `vvl_symbolic_query.llm_weight_from_logprob` qui sert déjà ailleurs dans le langage à pondérer les sorties LLM. La cohérence fonctionnelle évite à l'utilisateur de doubler les conventions.

Le cas `?::clause` est le plus pur : il représente l'**ignorance totale**. La Beta(1, 1) est uniforme — toutes les valeurs entre 0 et 1 sont également plausibles a priori. Ce prior dit littéralement *« je n'ai aucune idée du poids de cette clause, à toi d'apprendre depuis les observations »*. C'est le prior à utiliser quand on introduit une **cause latente** dont on veut faire émerger le taux à partir d'observations.

Le cas `learned` est légèrement particulier : il représente non un prior écrit à la main mais un posterior **issu d'un apprentissage** précédent. Quand le module Gibbs converge sur une posterior, il écrit dans la KB une nouvelle annotation marquée `learned`, dont la Beta est exactement celle qui a été apprise. Cette annotation peut ensuite servir de prior à un nouvel apprentissage — c'est la **mise à jour séquentielle** examinée en 35.10.

35.4. Le modèle génératif

Une fois les croyances priors fixées, il faut décrire comment elles se composent pour produire les observations. Cette description prend la forme d'un **modèle génératif** : un récit probabiliste qui explique, étape par étape, comment le monde fait apparaître les données qu'on observe. Le modèle du module `bayes/` comporte quatre niveaux, représentés dans la figure ci-dessous.

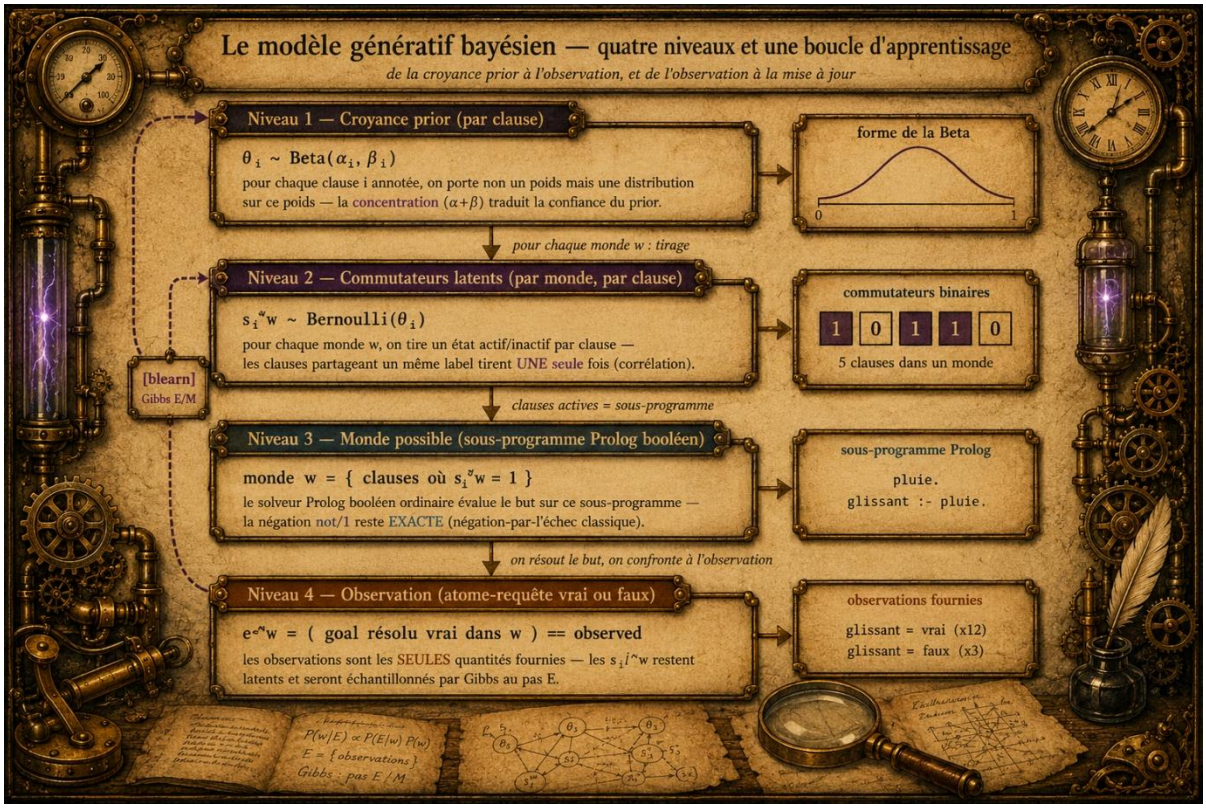


Figure 35.2 — *Le modèle génératif bayésien à quatre niveaux. Les priors Beta produisent des poids θ ; les poids produisent des commutateurs Bernoulli par monde ; les commutateurs définissent un sous-programme Prolog ; le sous-programme évalué donne l'observation. La boucle Gibbs en violet remonte des observations vers les priors.*

35.4.1. Lecture descendante : de la croyance à l'observation

Le modèle se lit de haut en bas, niveau par niveau.

Niveau 1 — la croyance prior. Pour chaque clause `i` annotée, le module porte une croyance sur son poids : θ_i ~ Beta(α_i, β_i). Cette croyance est la racine du modèle ; elle reflète à la fois la moyenne attendue (`α / (α + β)`) et la confiance dans cette moyenne (`α + β`).

Niveau 2 — les commutateurs latents. Pour chaque *monde possible* `w` et chaque clause `i`, on tire un **commutateur binaire** `s_i^w ~ Bernoulli(θ_i)`. Ce commutateur indique si la clause est *active* dans le monde `w` (1) ou *inactive* (0). On parle ici de *monde* au sens de la sémantique des mondes possibles : un même modèle génératif peut produire de nombreux mondes différents par tirage indépendant des commutateurs. Une nuance cruciale : les clauses partageant un même **label** tirent **un seul commutateur par monde** — c'est le mécanisme de corrélation que nous verrons en 35.10.

Niveau 3 — le monde comme sous-programme Prolog. L'ensemble des clauses actives dans un monde `w` constitue un **sous-programme Prolog booléen** — autrement dit, un programme Prolog ordinaire, ni probabiliste ni étendu. Ce sous-programme est résolu par le solveur Prolog booléen existant, sans modification. La négation y est traitée comme partout en Prolog : par négation-par-l'échec classique, c'est-à-dire *exactement*. Ce point méritera son propre développement en 35.8.

Niveau 4 — l'observation. Pour chaque monde, on évalue le ou les buts visés par les observations. L'observation associée à un monde est l'égalité entre le résultat de l'évaluation et la valeur fournie par l'utilisateur : `e^w = (goal résolu vrai dans w) == observed`. Les observations sont les **seules quantités fournies** au module — les commutateurs `s_i^w` restent latents et seront échantillonnés par l'algorithme de Gibbs au pas E.

35.4.2. La conjugaison Beta-Bernoulli

Le couplage entre les niveaux 1 et 2 du modèle — Beta sur les poids, Bernoulli sur les commutateurs — n'est pas un choix esthétique. C'est l'application directe d'une **propriété mathématique** : la Beta est la *conjuguée* de la Bernoulli. Concrètement, si l'on a un prior `θ ~ Beta(α, β)` et qu'on observe une suite de commutateurs Bernoulli avec `k` succès et `n - k` échecs, alors la posterior est *aussi* une Beta : `θ | données ~ Beta(α + k, β + n - k)`.

Cette propriété est ce qui rend l'apprentissage praticable. Sans elle, la mise à jour du prior par les observations exigerait une intégration numérique coûteuse à chaque pas ; avec elle, la mise à jour se résume à une **addition de comptages**. Les versions modernes des langages probabilistes — Stan, PyMC, Pyro — utilisent toutes ce principe quand elles le peuvent, parce que rien d'autre n'offre simultanément l'élégance théorique et l'efficacité de calcul.

35.4.3. Indépendance des observations

Chaque observation est, dans le modèle, un **tirage indépendant** du processus génératif. C'est-à-dire : pour chaque observation `w`, on tire **un nouveau jeu de commutateurs** indépendamment des tirages précédents. Cette hypothèse est essentielle à la validité de l'algorithme de Gibbs ; elle correspond à l'intuition que chaque observation est une *réalisation* du processus, et non une continuation des précédentes.

Concrètement, si l'utilisateur fournit quinze observations de `glissant` (vrai ou faux), chacune correspond à un *monde différent* où les clauses se sont déclenchées indépendamment. Le module ne modélise pas, par défaut, les dépendances temporelles entre observations ; pour des séries temporelles ou des chaînes causales explicites, des extensions seraient nécessaires (et restent hors périmètre du MVP).

35.5. Annotation des clauses

Le modèle génératif étant en place, passons à la **mécanique d'annotation** : comment, concrètement, déclarer en VVL qu'une clause est annotée d'une croyance bayésienne ? Le module expose trois builtins pour cela : `[bfacts]`, `[brules]`, et `[bannotate]`. Tous trois partagent une discipline commune et se composent avec les builtins ordinaires `[facts]` et `[rules]` du chapitre 19.

35.5.1. La syntaxe préfixée p::clause

L'annotation la plus directe se fait par une syntaxe préfixée : on préfixe la clause Prolog ordinaire par sa probabilité et le séparateur `::`. Cette convention syntaxique reprend celle de ProbLog, ce qui facilite la transition pour les utilisateurs qui connaissent ce langage.

```
[kb set "bayes_demo"]

# Faits annotes : la syntaxe p::fact
[bfacts "bayes_demo"
    "0.8::pluie."
    "0.3::brouillard."
]

# Regle annotee : la syntaxe p::rule
```

```
[brules "bayes_demo"
    "0.9::glissant(R) :- pluie, route(R)."
]

# Faits DETERMINISTES ordinaires (proba implicite 1.0)
[facts "bayes_demo"
    "route(r1)."
    "route(r2)."
]
```

Trois observations sur cet exemple. **Première observation** : la syntaxe `0.8::pluie.` se lit *« la clause pluie a une moyenne prior de 0.8 »*. La concentration est attribuée par défaut selon la source `user` (concentration 20), sauf indication contraire (voir 35.5.3). **Deuxième observation** : les builtins `[bfacts]` et `[brules]` acceptent **plusieurs** clauses dans le même appel, comme leurs homologues `[facts]` et `[rules]`. **Troisième observation** : on peut **mélanger** des clauses annotées et des clauses ordinaires dans la même base sans aucun conflit. Les clauses ordinaires conservent une probabilité implicite de 1.0 et n'apparaissent dans aucune structure de croyance — la rétro-compatibilité est totale.

35.5.2. La syntaxe ?::clause pour l'ignorance totale

Une variante syntaxique est essentielle : `?::clause` déclare que la clause est *à apprendre*. La probabilité prior est alors fixée à Beta(1, 1) — uniforme parfait — quelle que soit la source. C'est la manière de dire au module *« je ne connais pas le poids de cette clause, déduis-le des observations »*.

```
# Clause a apprendre
[bfacts "bayes_demo" "?::verglas."]

# Equivalent strict via [bannotate]
[bannotate "bayes_demo" "verglas." 0.5 "user"]
#                                  ^^^   ^^^^^^
#                                  moyenne 0.5, source par defaut
#                                  -- mais avec concentration faible
#
# La syntaxe ?:: est en realite un raccourci pour Beta(1,1),
# qui est plus ignorant que [bannotate cl 0.5 "user"] (Beta(10,10)).
```

L'usage typique de `?::clause` est de déclarer une **cause latente** qu'on veut faire émerger à partir d'observations indirectes. Dans le scénario canonique du tutoriel 02, `?::pluie.` est déclarée ignorante, et c'est l'observation répétée de `glissant` (qui dépend de `pluie` par une règle déterministe) qui permettra à Gibbs de récupérer le taux réel de pluie. Sans la syntaxe `?::`, il faudrait passer par `[bannotate ... "unknown"]`, plus verbeux.

35.5.3. Annotation fine : [bannotate]

Pour un contrôle plus précis — moyenne, source, et éventuellement label —, le builtin `[bannotate]` expose la signature complète.

```
# Signature : [bannotate kb "clause" p source? label?]

# Annotation avec source forte (concentration ~40)
[bannotate "bayes_demo" "capteur_hs." 0.1 "fact"]

# Annotation avec label : corrige la cause latente partagee
[bannotate "bayes_demo" "capteur_a_hs." 0.2 "user" "coupure"]
[bannotate "bayes_demo" "capteur_b_hs." 0.2 "user" "coupure"]
#                                                    ^^^^^^^
#  meme label => meme variable aleatoire (cf. 35.10)
```

Les paramètres optionnels `source` et `label` permettent respectivement de spécifier la **provenance** (qui détermine la concentration du prior selon le tableau de la section 35.3) et le **label de corrélation** (qui permet à plusieurs clauses de partager une même variable aléatoire). Si la source est omise, elle vaut `user` par défaut. Si le label est omis, la clause a sa propre variable aléatoire indépendante.

35.6. Inspection des croyances

Une fois les clauses annotées, deux builtins permettent d'inspecter les croyances enregistrées dans la base : `[bposterior]` pour une clause particulière, `[bbeliefs]` pour la vue complète.

35.6.1. [bposterior] : la croyance d'une clause

`[bposterior kb "clause."]` retourne, sous forme d'un objet JSON, l'intégralité de la croyance associée à une clause précise. La structure du retour est standardisée et expose les informations essentielles pour comprendre ou exploiter la croyance.

```
$croyance = [bposterior "bayes_demo" "pluie."]
[log $croyance]

# => {
#      "annotated":     true,
#      "mean":          0.8,
#      "alpha":         16.0,
#      "beta":          4.0,
#      "concentration": 20.0,
#      "ci":            [0.62, 0.93],
#      "source":        "user",
#      "label":         null
#    }
```

Les champs sont explicites. `mean` est la moyenne de la Beta — c'est le scalaire que l'utilisateur a fourni. `alpha` et `beta` sont les paramètres de la distribution. `concentration` est leur somme. `ci` est l'intervalle de crédibilité à 95 % — la fourchette dans laquelle le *vrai* taux se trouve avec 95 % de probabilité selon notre croyance courante. `source` rappelle la provenance ; `label` est le label de corrélation s'il y en a un.

Le champ `annotated` vaut `true` pour une clause qui a effectivement une croyance enregistrée, et `false` pour une clause **ordinaire non annotée**. Dans ce dernier cas, les autres champs sont absents — le retour est minimal :

```
# Clause deterministe ordinaire
$r = [bposterior "bayes_demo" "route(r1)."]
[log $r]
# => {"annotated": false}
#
# Une clause non annotee se comporte comme du Prolog ordinaire :
# probabilite implicite 1.0, aucune croyance stockee.
```

35.6.2. [bbeliefs] : la vue complète

`[bbeliefs kb]` retourne la liste complète des croyances enregistrées dans la base, sous forme d'un tableau d'objets. C'est l'outil de vue d'ensemble qu'on appelle typiquement après une session d'annotation, ou à intervalle régulier pour suivre l'évolution des croyances pendant un apprentissage.

```
[log [bbeliefs "bayes_demo"]]

# => [
#      {"clause": "pluie.",       "mean": 0.8, "concentration": 20.0, ...},
#      {"clause": "brouillard.",  "mean": 0.3, "concentration": 20.0, ...},
#      {"clause": "0.9::glissant(R) :- pluie, route(R).",
#                                  "mean": 0.9, "concentration": 20.0, ...},
#      {"clause": "verglas.",     "mean": 0.5, "concentration": 2.0, ...},
#      {"clause": "capteur_hs.",  "mean": 0.1, "concentration": 40.0, ...}
#    ]
```

Cette vue n'inclut **que** les clauses annotées — les clauses ordinaires en sont absentes. Une base qui n'utiliserait aucun builtin `b*` retournerait simplement un tableau vide à `[bbeliefs]`, ce qui confirme une fois de plus la rétro-compatibilité du mécanisme.

35.7. L'apprentissage par échantillonnage de Gibbs

Avec les annotations et l'inspection en place, abordons le cœur du module : l'**apprentissage** des croyances à partir d'observations. Le builtin pivot est `[blearn]`, qui implémente un algorithme d'échantillonnage de Gibbs par augmentation de données. La figure ci-dessous présente le schéma général.

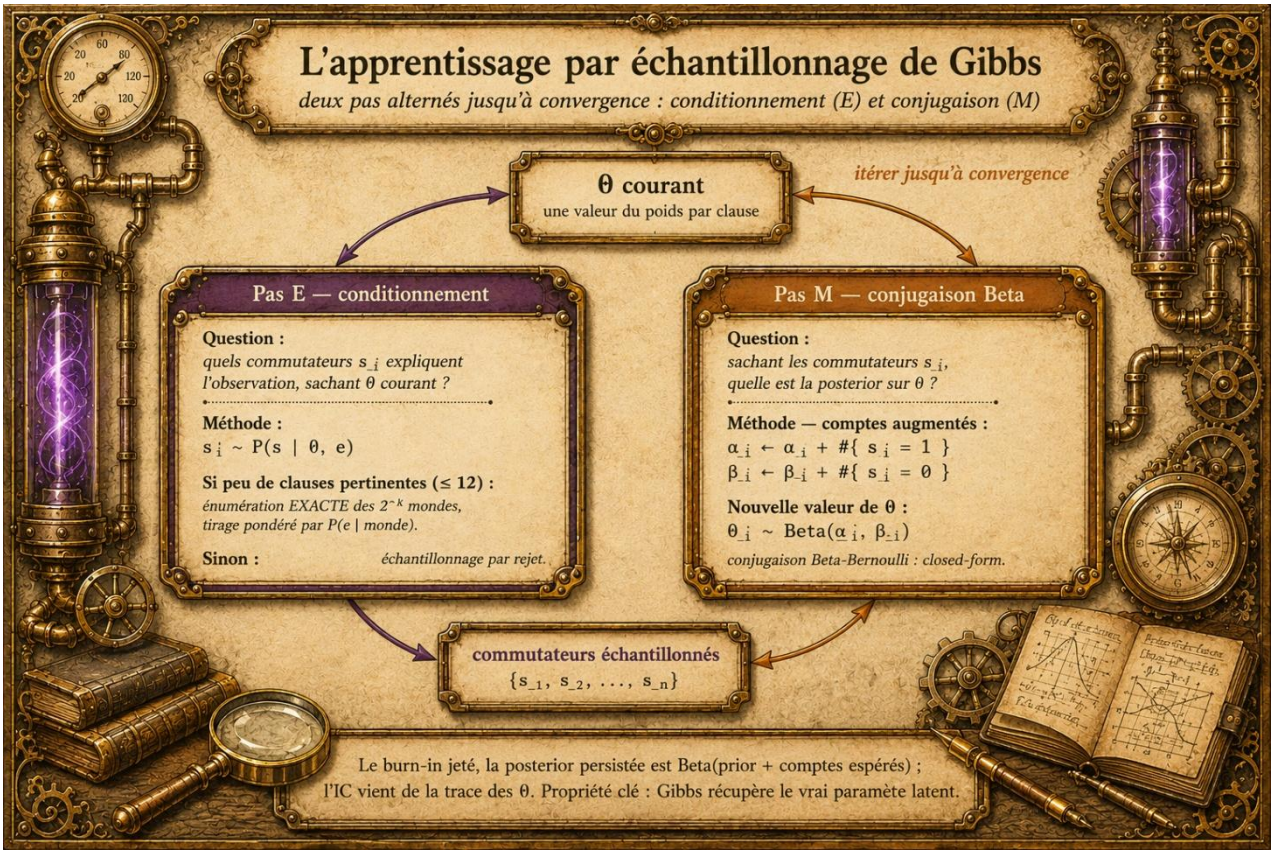


Figure 35.3 — *L'apprentissage par Gibbs alterne deux pas : un pas E qui échantillonne les commutateurs latents sachant les poids courants, et un pas M qui met à jour les paramètres Beta par conjugaison. La boucle itère jusqu'à convergence.*

35.7.1. Signature et flux global

Le builtin se prête à un usage très simple :

```
# Signature : [blearn kb OBS_JSON iters? seed?]
#
#   OBS_JSON   tableau d'observations
#              format : [{"goal": "<atome ground>", "obs": true|false}, ...]
#   iters?     nombre d'iterations (defaut : depend du moteur)
#   seed?      graine d'aleatoire (pour reproductibilite)

$obs = [JSON '[
  {"goal":"glissant","obs":true},
  {"goal":"glissant","obs":true},
  {"goal":"glissant","obs":true},
  {"goal":"glissant","obs":false}
]']

$rapport = [blearn "bayes_02" $obs 200 42]
[log "rapport :" $rapport]
#
# => {
#      "iterations":      200,
#      "burn_in":         50,
#      "n_observations":  4,
```

```
#      "posterior_mean":  {"pluie.": 0.74},
#      "ci":              {"pluie.": [0.45, 0.95]},
#      ...
#    }
```

Une fois `[blearn]` exécuté, les croyances des clauses concernées sont **mises à jour dans la base** : leur source devient `learned`, leur α et leur β reflètent le posterior calculé, leur concentration augmente proportionnellement au volume d'évidence. Le rapport retourné permet d'inspecter immédiatement le résultat de l'apprentissage sans avoir à rappeler `[bposterior]`.

35.7.2. Le pas E : conditionnement

Le **pas E** (pour *expectation*, par analogie avec l'algorithme EM) résout la question suivante : *étant données les valeurs courantes des poids θ, et étant donnée une observation, quels commutateurs s_i ont raisonnablement produit cette observation ?* Formellement, on tire `s_i ~ P(s | θ, e)`.

Pour résoudre concrètement ce tirage, le module distingue deux cas selon le nombre de clauses **pertinentes** pour l'observation — c'est-à-dire les clauses dont le commutateur peut influencer le résultat du but visé.

Cas favorable — peu de clauses pertinentes (≤ 12). Le module énumère **exactement** les `2^k` mondes possibles (où `k` est le nombre de clauses pertinentes), évalue le but dans chaque monde, et calcule pour chacun la probabilité conjointe `P(monde) × P(observation | monde)`. Le tirage se fait alors par pondération exacte parmi les mondes compatibles avec l'observation. Cette énumération **exacte** garantit l'absence d'erreur d'approximation tant que le nombre de clauses pertinentes reste modeste.

Cas défavorable — plus de 12 clauses pertinentes. L'énumération deviendrait prohibitive (`2^13 = 8192`, `2^20 ≈ 10^6`…) ; le module bascule sur un échantillonnage par **rejet** : on tire un monde au hasard selon le prior, on évalue le but, et on accepte le tirage uniquement s'il est compatible avec l'observation. Cette méthode reste correcte mais devient lente quand l'observation est rare. Le seuil de 12 a été choisi empiriquement comme le point de bascule où les deux méthodes ont des coûts comparables.

35.7.3. Le pas M : conjugaison Beta

Le **pas M** (pour *maximization*, dans le vocabulaire EM, bien qu'on parle ici d'un pas de **rééchantillonnage** plus que de maximisation stricte) effectue la mise à jour bayésienne des paramètres Beta. À chaque clause `i`, on agrège les commutateurs tirés au pas E pour cette clause :

```
pour chaque clause i pertinente :
    n1 = nombre de mondes ou s_i = 1
    n0 = nombre de mondes ou s_i = 0

    alpha_i  <-  alpha_i + n1     # ajout des succes observes
    beta_i   <-  beta_i  + n0     # ajout des echecs observes

    theta_i  ~   Beta(alpha_i, beta_i)   # nouvelle valeur tiree
```

Cette mise à jour est ce que la conjugaison Beta-Bernoulli rend si économique : pas d'intégration numérique, pas d'optimisation, juste une **addition de comptages** suivie d'un tirage. Le coût total de 200 itérations sur une base de quelques dizaines de clauses se compte en dizaines de millisecondes, ce qui rend l'apprentissage **utilisable en ligne** plutôt que comme étape de pré-traitement lourd.

35.7.4. La pertinence : ne créditer que ce qui a influé

Une subtilité importante de l'algorithme mérite un développement propre : le pas M ne met à jour que les clauses dont le commutateur a **réellement influé** sur l'observation. Une clause dont le commutateur n'a aucun effet sur la valeur du but observé reste *indifférente* à cette observation et son α/β ne sont pas modifiés.

Cette discipline évite ce que la littérature bayésienne appelle la **sur-confiance par crédit non justifié** : si une clause est réellement indifférente, l'apprendre depuis des observations qui ne la concernent pas conduirait à artificiellement piquer sa Beta autour de sa moyenne prior, comme si elle avait été *confirmée* alors qu'elle n'a fait que ne pas être *contredite*. Le module évite ce piège en filtrant explicitement la pertinence à chaque pas M.

35.7.5. La propriété clé : récupérer le paramètre latent

L'algorithme de Gibbs ainsi conçu possède une **propriété remarquable** : il récupère le vrai paramètre latent quand celui-ci est identifiable depuis les observations. Cette propriété est vérifiée par un test de régression sur le scénario canonique du tutoriel 02.

Scénario. Base : `?::pluie.` (probabilité de pluie inconnue, prior Beta(1, 1)) ; règle déterministe : `glissant :- pluie.`. Observations : 12 fois `glissant = vrai`, 3 fois `glissant = faux` — soit un taux empirique de 0.8.

Résultat de Gibbs. Après 200 itérations, `[bposterior "pluie."]` retourne une moyenne de l'ordre de 0.8. Le module a *récupéré* la probabilité de pluie alors qu'elle n'a **jamais été observée directement** — seul l'observable `glissant` était fourni. La règle déterministe a fait le pont entre l'observable et la cause latente, et Gibbs a remonté ce pont.

Cette propriété est l'argument le plus fort en faveur du module : elle montre qu'on peut **inférer des probabilités sur des causes que l'on n'observe pas directement**, pourvu qu'on observe des effets qui en dépendent. C'est précisément ce qu'on demande à un système d'inférence bayésienne digne du nom.

35.8. La négation est exacte

Un point technique qui distingue radicalement le module `bayes/` de la plupart des langages probabilistes mérite un développement dédié : **la négation y est traitée exactement**, sans approximation. Cette propriété, qui peut sembler anodine au premier abord, ouvre des possibilités d'apprentissage qui sont structurellement hors d'atteinte des approches concurrentes.

35.8.1. Pourquoi la négation est-elle un problème

Dans les langages probabilistes qui utilisent une représentation compilée — circuit booléen pondéré, diagramme de décision binaire, réseau bayésien marginalisé —, la négation pose un problème fondamental. La probabilité d'une conjonction `P(A ∧ B)` se calcule facilement quand A et B sont indépendants : `P(A) × P(B)`. La probabilité d'une disjonction `P(A ∨ B)` aussi : `P(A) + P(B) - P(A ∧`

`B)`. Mais `P(¬A)` n'est trivial *que* si on a une représentation complète de A — sinon, la négation devient une opération coûteuse, et plusieurs langages probabilistes la *approchent* par des techniques d'inférence approchée (échantillonnage MCMC, propagation de croyance approximative).

L'approche du module `bayes/` court-circuite entièrement ce problème en exploitant la structure du modèle génératif : chaque monde possible est, par construction, un **programme Prolog booléen ordinaire**, fixé. Dans un tel programme, la négation est celle de Prolog standard — la négation-par-l'échec — qui est *exacte par construction*. Le module n'introduit donc **aucune approximation supplémentaire** sur la négation.

35.8.2. Apprendre à travers la négation

La conséquence pratique de cette propriété est puissante : on peut **apprendre le poids d'une clause à travers une négation**. Le tutoriel 03 illustre ce scénario.

```
001  [kb set "bayes_03"]
002
003  # Modele : pluie inconnue + regle a NEGATION
004  #   sec est vrai SSI il ne pleut pas
005  [bfacts "bayes_03" "?::pluie."]
006  [brules "bayes_03" "sec :- not(pluie)."]
007
008  # Observations : sec vrai 5 fois sur 16 (~31%)
009  #   => pluie devrait approcher 0.69 (= 1 - 0.31)
010  $obs = [JSON '[
011    {"goal":"sec","obs":true},  {"goal":"sec","obs":true},
012    {"goal":"sec","obs":true},  {"goal":"sec","obs":true},
013    {"goal":"sec","obs":true},
014    {"goal":"sec","obs":false}, {"goal":"sec","obs":false},
015    {"goal":"sec","obs":false}, {"goal":"sec","obs":false},
016    {"goal":"sec","obs":false}, {"goal":"sec","obs":false},
017    {"goal":"sec","obs":false}, {"goal":"sec","obs":false},
018    {"goal":"sec","obs":false}, {"goal":"sec","obs":false},
019    {"goal":"sec","obs":false}
020  ]']
021
022  [blearn "bayes_03" $obs 200 7]
023  [log [bposterior "bayes_03" "pluie."]]
024  # => mean ~ 0.7
```

Le résultat est remarquable. La cause latente `pluie` n'est **jamais observée** ; seul `sec` l'est. Le lien entre les deux est une règle `sec :- not(pluie)` qui inverse logiquement la première. Pourtant, Gibbs récupère correctement que `pluie` a un taux d'environ 0.7 — *à travers la négation*. Aucune approximation n'est introduite ; la sémantique de la négation Prolog est respectée monde par monde.

35.8.3. Limite : programmes stratifiés

Une réserve technique mérite d'être nommée. La sémantique de la négation-par-l'échec est rigoureusement définie pour les programmes **stratifiés** — c'est-à-dire ceux dont les négations ne construisent pas de cycle. Pour les programmes avec récursion négative non stratifiée (par exemple `a`

`:- not(b). b :- not(a).`), la sémantique elle-même devient ambiguë et plusieurs interprétations concurrentes existent dans la littérature (sémantique des modèles stables, sémantique well-founded, etc.).

Le module `bayes/` n'opère **proprement** que sur des programmes stratifiés. Cette restriction n'est pas spécifique au module — elle vaut déjà pour le moteur Prolog ordinaire — mais elle mérite d'être rappelée dans le contexte probabiliste, où la tentation est plus grande de construire des modèles cycliques pour modéliser des rétroactions. Pour de tels modèles, des outils dédiés (PLP well-founded, ASP probabiliste) sont mieux adaptés.

35.9. Prédiction avec propagation d'incertitude

L'apprentissage produit des croyances posterior ; encore faut-il savoir les **utiliser** pour prédire la probabilité de buts qu'on n'a pas encore observés. C'est le rôle des deux builtins `[bprob]` et `[bapprox]`. La première subtilité à comprendre — et la plus importante du chapitre, peut-être — est que ces builtins **ne pluggent pas la moyenne** : ils propagent la *distribution* entière.

35.9.1. [bprob] : propagation Beta et IC épistémique

`[bprob]` calcule la probabilité prédictive d'un but en propageant la croyance Beta des clauses pertinentes, sans réduire ces croyances à leur moyenne. Le retour est un objet structuré qui expose à la fois la valeur centrale et la mesure de l'incertitude.

```
# Signature : [bprob kb "goal" n_outer? n_inner? seed?]
#
#   n_outer    nombre de tirages externes (sur theta)
#   n_inner    nombre de tirages internes (sur les commutateurs)
#   seed       graine d'aleatoire

$p = [bprob "bayes_04" "glissant" 150 8 2]
[log $p]

# => {
#      "value":      0.74,                  # P(glissant)
#      "ci":         [0.58, 0.88],          # IC EPISTEMIQUE
#      "range":      [0.58, 0.88],
#      "confidence": "moderate",
#      "hypotheses": ["pluie."]
#    }
```

Quatre champs essentiels structurent le retour. **`value`** est la probabilité moyenne du but — la valeur centrale qu'un système naïf aurait été le seul à fournir. **`ci`** est l'intervalle de crédibilité **épistémique** : la fourchette dans laquelle la *vraie* probabilité se trouve, étant donnée notre ignorance des poids exacts. Cette information est précieuse : un système prédictif qui annonce 0.74 ± 0.15 est très différent de l'un qui annonce 0.74 ± 0.02, alors que la valeur centrale est la même. **`range`** donne la même information sous forme de fourchette. **`confidence`** est un label catégoriel — `exact`, `high`, `moderate`, `low` — qui permet de brancher la prédiction sur l'échelle de confiance VVL standardisée par `[approximation:*]`.

35.9.2. Le calcul interne : double échantillonnage

Pour calculer ces quantités, `[bprob]` effectue une **double-échantillonnage** qui sépare proprement les deux incertitudes. Au tirage *externe*, le module échantillonne `n_outer` valeurs des poids θ depuis leurs Beta posterior. Pour chaque valeur de θ ainsi obtenue, il calcule au tirage *interne* la probabilité exacte de l'observation, soit par énumération des `n_inner` mondes pertinents (si peu de clauses), soit par échantillonnage. La valeur centrale est la moyenne des probabilités obtenues ; l'IC épistémique vient de leur dispersion.

Cette séparation est ce qui permet d'**isoler l'incertitude épistémique** (la variance entre tirages externes) du **bruit d'échantillonnage interne** (l'erreur Monte-Carlo sur le calcul de P(goal | θ)). Une approche qui plombe directement la moyenne dans le calcul de P(goal | θ) confondrait les deux et serait incapable de distinguer une prédiction *moyennement sûre* d'une prédiction *très sûre en moyenne mais avec peu de confiance dans la moyenne elle-même*.

35.9.3. [bapprox] : intégration arithmétique via VVL_Approximation

`[bapprox]` encapsule le même calcul dans une valeur de type **VVL_Approximation** — une primitive du langage qui supporte l'arithmétique avec propagation d'incertitude. La probabilité devient alors un *opérande* qui peut être multiplié, ajouté, comparé, et dont l'incertitude se propage automatiquement à travers les calculs.

```
001  # Probabilite emballee en VVL_Approximation
002  $risque = [bapprox "bayes_04" "glissant" 120 8 3]
003
004  # Acces explicite aux champs via la suite [approximation:*]
005  [log "  P(glissant)   =" [approximation:value      $risque]]
006  [log "  confiance     =" [approximation:confidence $risque]]
007  [log "  fourchette    =" [approximation:range      $risque]]
008
009  # Et — c'est le point clef — arithmetique propagee
010  $cout_salage = 1000 * $risque
011  [log "  cout attendu  =" [approximation:value $cout_salage]]
012  [log "  fourchette    =" [approximation:range $cout_salage]]
013  #
014  # La multiplication a propage l'incertitude epistemique
015  # au cout : on connait le cout MOYEN attendu, et on connait
016  # aussi l'INCERTITUDE sur ce cout, sans calcul supplementaire.
```

Ce mécanisme est, à proprement parler, le **pont** entre le module bayésien et le reste du langage. Une probabilité apprise par Gibbs devient, via `[bapprox]`, une grandeur arithmétique de plein droit — qui peut être combinée avec d'autres grandeurs (coûts, durées, scores) sans aucune ligne de code supplémentaire. Le résultat de la combinaison reste une `VVL_Approximation` propre, qui peut elle-même être passée à d'autres builtins, comparée à des seuils, ou exposée dans une UI.

35.9.4. Confidence : exact / high / moderate / low

Le label de confiance mérite un dernier mot. Plutôt que d'imposer à l'utilisateur d'interpréter lui-même la largeur de l'IC, le module la traduit en une étiquette catégorielle standardisée selon le tableau suivant.

Label	Critère typique	Lecture pratique
exact	IC quasi-nul, concentration très forte	la valeur est connue avec certitude
high	IC étroit, concentration forte	la valeur est bien établie, on peut s'y fier
moderate	IC modéré, concentration moyenne	la valeur est plausible mais incertaine
low	IC large, concentration faible	on en sait trop peu, à raffiner par plus d'observations

Cette étiquette permet à un script consommateur de prendre des **décisions différenciées** sans inspecter les paramètres bruts. Par exemple, un agent qui se base sur une prédiction `low` peut déclencher une collecte d'observations supplémentaires avant d'agir, tandis qu'une prédiction `high` autorise une action immédiate.

35.10. Mise à jour séquentielle et labels

Deux mécanismes complémentaires du module mettent en valeur la richesse de la représentation Beta : la **mise à jour séquentielle** (qui exploite la conjugaison pour apprendre en plusieurs lots) et les **labels de corrélation** (qui permettent à plusieurs clauses de partager une variable aléatoire). Les deux sont illustrés par le tutoriel 05.

35.10.1. Posterior → prior : la mise à jour séquentielle

La conjugaison Beta-Bernoulli a une conséquence pratique très agréable : la posterior d'un lot d'observations peut servir **directement** de prior pour le lot suivant, par simple réutilisation des α et β mis à jour. Aucune intégration spéciale, aucune recalibration — il suffit d'appeler `[blearn]` à nouveau sur le second lot.

```
[kb set "bayes_seq"]
[bfacts "bayes_seq" "?::pluie."]
[brules "bayes_seq" "glissant :- pluie."]

# Lot 1 : 8 vrais / 1 faux
$lot1 = [JSON '[ ... 9 observations ... ]']
[blearn "bayes_seq" $lot1 120 5]
[log "apres lot 1 :" [bposterior "bayes_seq" "pluie."]]
#
# => {"mean": 0.83, "concentration": 11.0, ...}

# Lot 2 : 8 vrais / 1 faux (les memes observations, donnees a part)
$lot2 = [JSON '[ ... 9 observations ... ]']
[blearn "bayes_seq" $lot2 120 6]
[log "apres lot 2 :" [bposterior "bayes_seq" "pluie."]]
#
# => {"mean": 0.85, "concentration": 20.0, ...}
```

Trois effets sont à observer dans cet exemple. **Effet 1** : la moyenne se stabilise (0.83 → 0.85) à mesure que l'évidence s'accumule. **Effet 2** : la concentration **augmente significativement** (11 → 20), ce qui traduit le resserrement de la croyance. **Effet 3** : l'intervalle de crédibilité se rétrécit proportionnellement à la concentration croissante, ce qui veut dire que la prédiction `[bprob]` qui s'appuie sur cette croyance deviendra de plus en plus précise.

Ce comportement est exactement ce qu'on attend d'un système **bayésien correct** : plus on observe, plus on est sûr. Le mécanisme est si naturel qu'il s'utilise sans cérémonie — on appelle simplement `[blearn]` autant de fois qu'on a de lots d'observations, et le module accumule l'évidence sans intervention supplémentaire.

35.10.2. Labels : variables aléatoires partagées

Le second mécanisme — les **labels** — permet à plusieurs clauses de partager **une seule variable aléatoire**. Concrètement, deux clauses annotées avec le même label tirent **un seul commutateur par monde** au pas E, et partagent **une seule croyance Beta**. Cette corrélation est utile chaque fois que plusieurs clauses dépendent de la même cause latente.

```
001  # Scenario : deux capteurs qui tombent en panne pour la MEME cause
002  #  (une coupure d'alimentation commune)
003
004  [bannotate "bayes_demo" "capteur_a_hs." 0.2 "user" "coupure"]
005  [bannotate "bayes_demo" "capteur_b_hs." 0.2 "user" "coupure"]
006
007  # Une SEULE entree dans bbeliefs : le label "coupure"
008  [log [bbeliefs "bayes_demo"]]
009  # => [{ "clause": "coupure",  // le label, pas la clause
010  #       "mean":   0.2,
011  #       "concentration": 20.0,
012  #       "clauses": ["capteur_a_hs.", "capteur_b_hs."]
013  #     }]
```

Sans le label, le module enregistrerait **deux croyances indépendantes** — l'une pour `capteur_a_hs.`, l'autre pour `capteur_b_hs.` — et l'apprentissage les mettrait à jour séparément. Avec le label, le module enregistre **une seule** croyance, sous le nom du label, et son apprentissage utilise les observations qui concernent l'**une ou l'autre** des deux clauses pour réviser un poids unique. La modélisation devient *causalement honnête* : la coupure d'alimentation est une cause unique qui explique les deux pannes, et on en mesure le taux à travers les deux observables qui en dépendent.

Sans cette mécanique, on aurait du **double comptage** : chaque panne de capteur serait considérée comme un événement indépendant, alors qu'elles ont en réalité une corrélation parfaite. Les labels résolvent ce problème d'identifiabilité sans introduire de modélisation graphique lourde.

35.11. Articulation avec les autres paradigmes d'apprentissage

Le module `bayes/` ne remplace aucun des paradigmes d'apprentissage exposés précédemment dans l'ouvrage — il les **complète**. Cette section précise l'articulation avec les trois voies du chapitre 34 (LLM, ILP, GA) et avec le mécanisme de tests intégrés du chapitre 32. C'est le passage qui justifie de parler d'une **quatrième voie**.

35.11.1. vs ILP (chapitre 27) : forme vs poids

L'induction logique inductive du chapitre 27 et l'annotation bayésienne du présent chapitre apprennent toutes deux des choses depuis des exemples, mais elles n'apprennent **pas la même chose**. ILP apprend la **forme** des clauses — quelles règles, avec quelles variables, quels arguments — sous garantie sound-and-complete. Bayes, à l'inverse, suppose les clauses **déjà données** et apprend leur **poids** — leur probabilité de déclenchement — avec mesure d'incertitude.

Cette différence n'est pas une opposition mais une **complémentarité**. Un système réaliste peut très bien : (1) utiliser ILP pour synthétiser la structure des règles à partir d'exemples étiquetés, puis (2) utiliser bayes pour calibrer le poids de ces règles synthétisées à partir d'observations en production. Les deux mécanismes opèrent à des niveaux différents de la chaîne d'apprentissage et n'entrent en conflit à aucun moment.

35.11.2. vs LLM (chapitre 17) : bayes calibre les règles synthétisées

L'induction par LLM produit des règles qui *plausiblement* couvrent un domaine. Mais leur poids — la confiance qu'on doit leur accorder — reste qualitatif : le logprob du modèle donne une indication, mais pas une probabilité opérationnelle. Le module bayes complète cette induction en transformant les règles LLM en clauses annotées dont le **prior reflète le logprob**, et dont l'apprentissage par Gibbs peut ensuite **affiner** ce prior à partir d'observations réelles.

Le pipeline canonique devient alors : (1) LLM produit des règles candidates ; (2) les règles sont ingérées via `[brules]` avec `source="LLM"` et prior calibré par logprob ; (3) une suite de tests d'audit (chapitre 32) écarte les règles manifestement incorrectes ; (4) les règles restantes sont calibrées par `[blearn]` sur des observations historiques ; (5) la prédiction en production utilise `[bprob]` avec IC épistémique pour propager la confiance résiduelle. C'est l'incarnation la plus complète à ce jour du **pont neuro-symbolique** que l'ouvrage défend depuis le chapitre 17.

35.11.3. vs tests (chapitre 32) : oracle binaire vs oracle gradué

Le système de tests intégrés du chapitre 32 — `[test:assert]`, `[test:equals]`, `[test:suite]` — fournit un **oracle binaire** : un test passe ou échoue, sans nuance intermédiaire. Cette discipline est exactement ce qu'il faut pour vérifier le comportement d'un programme déterministe. Mais elle ne convient pas, en l'état, pour auditer un système qui produit des **prédictions probabilistes** : un test qui vérifierait `[bprob ... "glissant"]` ne peut pas raisonnablement exiger une valeur exacte, sous peine d'être fragile aux variations d'échantillonnage.

Le module bayes complète l'arsenal en fournissant un **oracle gradué** : on peut comparer une prédiction à un intervalle acceptable, vérifier que l'IC contient la valeur cible, ou mesurer la log-vraisemblance d'observations de validation sous le modèle appris. Ces vérifications graduelles s'expriment naturellement en composant `[bprob]` ou `[bposterior]` avec `[test:assert]` :

```
[test:suite "calibration-bayes"
    $p = [bposterior "bayes_demo" "pluie."]
    [test:assert (and ($p.mean > 0.7) ($p.mean < 0.9))
        "moyenne de pluie dans l'intervalle attendu"]
    [test:assert ($p.concentration > 10)
        "concentration suffisante apres apprentissage"]
]
```

35.11.4. Les quatre paradigmes d'apprentissage

Le chapitre 34 récapitulait trois paradigmes d'**écriture** des règles. Avec ce chapitre 35, on peut désormais récapituler les quatre paradigmes d'**apprentissage**, qui ne sont pas exactement la même question.

Paradigme	Apprend...	Méthode	Chapitre
Manuel	rien (écriture explicite)	auteur humain	11-15
LLM	forme + plausibilité	génération neurale	17, 22
ILP	forme (sound-and-complete)	recherche logique	27
GA	stratégies, candidats	évolution simulée	26
Bayes	**poids + incertitude**	**Gibbs sur Beta-Bernoulli**	**35**

Les quatre paradigmes apprenants apprennent **des choses différentes** et se composent naturellement. Une chaîne complète d'apprentissage pourrait utiliser : ILP pour la forme, LLM pour compléter une couverture incomplète, GA pour explorer un espace de stratégies discret, bayes pour calibrer les poids et la confiance. Chacun apporte une dimension propre. La plateforme VVL, en hébergeant les quatre, ouvre l'horizon de la programmation neuro-symbolique au-delà de ce que chaque paradigme isolé permettrait.

35.12. Architecture interne, rétro-compatibilité, limites

Pour clore l'examen technique, cette section parcourt rapidement l'architecture interne du module, les détails de son intégration rétro-compatible avec le runtime existant, et les limites connues du MVP.

35.12.1. Cinq modules Python en table latérale

L'implémentation se répartit en cinq fichiers Python sous `src/vails/bayes/`, dont les responsabilités sont strictement séparées.

Fichier	Responsabilité
beliefs.py	la classe BetaBelief, l'élicitation du prior, le BayesStore
worlds.py	l'énumération des mondes possibles, la pertinence, la négation exacte
gibbs.py	l'algorithme d'apprentissage (pas E / pas M), le LearnReport
predict.py	la prédiction avec propagation d'incertitude épistémique
builtins.py	la surface VVL — les huit builtins b*

Le `BayesStore` est la **table latérale** mentionnée plus haut : une structure indexée par la signature de chaque clause (déjà calculée par le moteur Prolog pour sa propre déduplication) qui associe à chaque signature sa croyance. Cette table est attachée à l'engine sous l'attribut `eng.bayes`. Le moteur Prolog

lui-même n'est **pas modifié** ; il continue de fonctionner sur ses structures booléennes ordinaires, et les builtins `b*` consultent la table latérale lorsqu'ils ont besoin de poids probabilistes.

35.12.2. Rétro-compatibilité : le contrat

Trois garanties explicites assurent que la couche bayésienne ne casse rien de l'existant.

Garantie 1. Une base de connaissance `.vkb` sauvegardée avant la disponibilité du module se recharge sans erreur. `eng.bayes` est alors une table vide et la base se comporte exactement comme du Prolog booléen pur.

Garantie 2. Une clause ajoutée par `[facts]` ou `[rules]` ordinaires reste **déterministe** : sa probabilité implicite est 1.0, et `[bposterior]` la marque comme `annotated: false`. Aucun auto-attachement de croyance ne se produit.

Garantie 3. L'ajout du module dans le runtime se réduit à une **unique ligne** dans `vvl_builtins.py` : l'import de la surface exposée par `bayes/builtins.py`. Aucune modification ailleurs. Cette discipline rend le module désactivable trivialement — il suffit de commenter l'import.

35.12.3. Le piège STEP_COUNTER

Un détail d'implémentation mérite d'être mentionné parce qu'il a donné lieu à des bugs subtils pendant le développement. Le moteur Prolog `prolog.py` utilise un compteur global `STEP_COUNTER` pour limiter le coût des résolutions individuelles. Ce compteur n'est **jamais remis à zéro** entre deux requêtes successives, ce qui n'a aucune importance dans l'usage ordinaire — quelques résolutions à l'occasion — mais devient catastrophique pour l'apprentissage Gibbs, qui lance **plusieurs milliers** de résolutions par itération.

Le module `bayes/` contourne ce piège en réinitialisant explicitement `STEP_COUNTER` avant chaque résolution effectuée par `worlds.eval_goal`. Cette discipline est invisible pour l'utilisateur du module mais cruciale pour le bon fonctionnement de l'apprentissage sur des observations nombreuses. Tout code futur qui lancerait beaucoup de requêtes Prolog en boucle devra prendre la même précaution.

35.12.4. Limites et hypothèses

Quatre limites explicites du MVP doivent être nommées pour situer honnêtement les usages.

Limite 1 — observations ground. Les buts observés et les buts interrogés doivent être *ground* — c'est-à-dire sans variables libres. Cette restriction permet l'évaluation directe par le solveur Prolog ; les buts paramétrés (`P(glissant(X)) = ?` sur tous les `X` possibles) nécessiteraient une généralisation du moteur qui n'est pas dans le MVP.

Limite 2 — indépendance par défaut. Sauf si elles partagent un `label`, les clauses sont supposées **indépendantes** dans le modèle génératif. Pour des dépendances structurelles plus complexes — corrélations partielles, mélanges, dépendances temporelles — il faut soit factoriser le modèle par labels, soit envisager des outils externes.

Limite 3 — programmes stratifiés. Comme mentionné en 35.8.3, la sémantique de la négation impose des programmes stratifiés. Les programmes cycliques avec négation non-stratifiée ne sont pas couverts.

Limite 4 — bascule à 2^12 mondes. L'énumération exacte des mondes est faisable jusqu'à 12 clauses pertinentes (`2^12 = 4096` mondes). Au-delà, le module bascule en échantillonnage par rejet, ce qui

reste correct mais peut devenir lent pour des modèles denses. Pour des modèles très grands (centaines de clauses annotées), une refonte algorithmique vers des méthodes d'inférence approchée structurées (variational inference, particle filters) serait nécessaire — c'est un chantier ouvert au-delà du MVP.

35.13. Synthèse — le quatrième paradigme et le mot final

Ce chapitre additionnel a introduit une **quatrième voie d'apprentissage** pour la programmation neuro-symbolique en VVL : l'annotation stochastique bayésienne. Le mécanisme est techniquement modeste — huit builtins, un modèle à quatre niveaux, un algorithme à deux pas — mais son apport conceptuel est substantiel.

35.13.1. Ce que le module ajoute à l'écosystème

Trois apports principaux, qu'on peut résumer en un mot chacun. **Calibration** : le module sait *combien* on est sûr d'une règle, pas seulement *si* elle se déclenche. **Apprentissage** : il sait réviser cette croyance depuis des observations, par un algorithme exact (Gibbs sur conjugaison Beta-Bernoulli). **Propagation** : il sait transmettre l'incertitude résiduelle à travers des chaînes d'inférence et des compositions arithmétiques, sans la fondre dans une moyenne aveugle.

Ces trois apports adressent un manque réel dans l'arsenal du chapitre 27. ILP donne la forme exacte des règles ; bayes donne la confiance graduée qu'on doit leur accorder. Les deux mécanismes se composent harmonieusement : on peut induire la forme par ILP, puis calibrer le poids par bayes, et l'audit par tests intégrés (chapitre 32) valide le comportement final. La chaîne complète, désormais, ne laisse aucune dimension de l'incertitude sans réponse — ni l'incertitude structurelle (forme), ni l'incertitude paramétrique (poids), ni l'incertitude comportementale (tests).

35.13.2. Statut et intégration

Le module présenté fait partie du serveur courant, où il cohabite avec les autres couches additives de l'ouvrage. Sa conception le rend inoffensif pour l'existant : les croyances sont stockées dans une table latérale indexée par la signature des clauses, le moteur Prolog n'est pas modifié, et un programme qui n'emploie pas les builtins b* se comporte exactement comme avant. C'est cette discipline d'extension — additive plutôt que substitutive — qui a permis d'ajouter quatre sous-systèmes majeurs au langage sans casser une seule ligne de code utilisateur.

La trajectoire d'intégration se poursuit sur deux axes. Le premier est la couverture documentaire : la doctrine bayésienne doit rejoindre le corps principal de la documentation, ce qui suppose à terme une réécriture partielle du chapitre 34 pour intégrer le quatrième paradigme au tableau central plutôt qu'en prolongement. Le second est l'articulation avec les couches voisines : le pont déjà établi avec la logique relationnelle — une croyance Beta sert directement de poids à une branche [any-weighted], comme le montre le chapitre 38 — indique la direction, et d'autres jonctions du même ordre restent à établir avec la preuve formelle et l'approximation.

35.13.3. Le mot final, à présent ici

Le chapitre 34 s'était conclu sur une triple invitation au lecteur. Ce chapitre 35 ouvre la Partie IX et n'a pas à reprendre cette conclusion à son compte : la clôture de l'ouvrage se trouve désormais après le chapitre 40, où elle referme la totalité du parcours.

Aux trois invitations initiales — prendre un problème réel et le reformuler en VVL ; explorer un paradigme nouveau ; participer à la discussion publique — il faut en ajouter une quatrième : éprouver la couche probabiliste sur vos propres données. C'est le sous-système le plus jeune de l'ouvrage, donc le plus malléable, et celui pour lequel un retour d'usage a le plus de chances d'infléchir la trajectoire du langage. Cette quatrième invitation est plus exigeante que les trois autres, mais elle est aussi celle qui aura le plus d'impact.

L'auteur de ce livre a passé plusieurs années à concevoir et à éprouver VVL avant de pouvoir l'exposer dans la forme stabilisée que cet ouvrage présente. Il a, comme tous les concepteurs de langage, fait des choix qu'il regrette parfois, des choix qu'il défendra toujours, et des choix qu'il reverra peut-être un jour. Le résultat — la plateforme telle qu'elle est aujourd'hui — n'est pas le meilleur des résultats possibles ; c'est un résultat qui essaie d'être **cohérent**, qui prend au sérieux la programmation neuro-symbolique comme **discipline industrielle**, et qui offre au lecteur **un outil** plutôt qu'un manifeste.

Le module bayésien est le plus jeune des sous-systèmes de la Partie IX, et celui dont la trajectoire dépend le plus des retours d'usage. Le chapitre suivant aborde une couche d'une tout autre nature — le calcul symbolique exact — qui ne cherche pas à apprendre mais à calculer juste, et dont la preuve formelle du chapitre 37 fera plus tard son prouveur réel.

Les deux couches se rejoindront au chapitre 38, où une croyance Beta apprise ici sert directement de poids à une branche de la logique relationnelle.

36. Le calcul symbolique comme citoyen de première classe

Mots-clés : *CAS, calcul symbolique, SymPy, VVL_Sym, expression exacte, résolution, dérivation, intégration, équations différentielles, algèbre linéaire, rationnel, radical, liaison paresseuse, pont approximation, [resolve], neuro-symbolique, sandbox AST, additif*

Périmètre. La couche de calcul symbolique exposée dans ce chapitre fait partie du serveur courant. Le paquet src/vails/symbolic/ apporte une vingtaine de builtins préfixés symbolic:* ainsi que deux sucres syntaxiques, [$ …] et [? …]. Comme le module bayésien du chapitre précédent, il s'agit d'une couche strictement additive : si SymPy n'est pas installé, le serveur démarre et les builtins ne sont simplement pas enregistrés. Le chapitre s'adresse aux praticiens qui veulent l'exploiter, et aux lecteurs qui souhaitent comprendre comment un système de calcul algébrique formel s'intègre dans un langage qui n'a pas été conçu autour de lui.

Contrairement au chapitre 35, ce chapitre **ne décrit pas une voie d'apprentissage**. Le calcul symbolique exact ne calibre rien, n'apprend rien depuis des observations — il *calcule* exactement, déterministiquement, des expressions algébriques. C'est un outil **orthogonal** aux quatre paradigmes du chapitre 34. Sa place est celle d'un nouvel **étage** du système numérique de VVL, qui prend place entre les nombres concrets (`VVL_Num`) et le système d'approximation calibrée (`VVL_Approximation`).

36.1. Position du module : couche CAS additive

Le module `symbolic/` suit la même discipline d'extension que le module `bayes/` du chapitre 35 : couche **strictement additive**, qui ne modifie aucune fonctionnalité existante. Si SymPy n'est pas installé sur l'environnement, le serveur démarre quand même et les builtins `[symbolic:*]` retournent une erreur claire qui indique la dépendance manquante. Le reste du langage continue de fonctionner intégralement. Cette dégradation gracieuse est la marque, dans l'écosystème VVL, des modules additifs qu'on peut adopter ou ignorer sans conséquence sur la base de code existante.

36.1.1. Trois utilisations qui justifient un CAS

VVL disposait, avant l'introduction du module `symbolic/`, de trois façons distinctes de représenter une quantité : les **nombres concrets** (`VVL_Num`, qui sont des entiers ou des flottants ordinaires) ; les **objets abstraits** modélisés par l'algèbre Zalta ; et les **approximations** (`VVL_Approximation`) pilotées par LLM, qui portent une valeur centrale, une fourchette, et un label de confiance. Chacune de ces représentations a son rôle propre, mais aucune ne couvre le cas du **calcul exact non-approché**.

Trois cas d'usage typiques exposent la lacune. **Premier cas — la préservation de la précision algébrique** : $\sqrt{2}$ est un nombre parfaitement bien défini, mais sa représentation flottante (1.4142…) introduit une erreur qui se propage dans toute la suite des calculs. Sur trois opérations imbriquées, l'erreur cumulée peut atteindre des fractions de pourcent — acceptable pour de l'estimation, inadmissible pour de la démonstration. **Deuxième cas — la résolution exacte** : `x² − 5x + 6 = 0` a pour solutions exactes `{2, 3}`, pas `{1.9999…, 3.0000…}`. Un solveur numérique peut approcher ces racines mais ne peut pas certifier leur exactitude. **Troisième cas — le calcul paramétrique** : la formule quadratique `x = (−b ± √(b² − 4ac)) / 2a` est une *expression*, pas un *nombre* ; elle se

manipule, se substitue, se simplifie, mais ne se calcule qu'au moment où l'on fournit les valeurs des paramètres.

Le module `symbolic/` adresse ces trois cas en exposant un véritable **système de calcul algébrique** (Computer Algebra System, ou CAS) intégré au cœur du langage. La technologie sous-jacente est **SymPy** — la bibliothèque Python de référence pour ce type de calcul.

36.1.2. Pourquoi SymPy spécifiquement

Le choix de SymPy comme moteur sous-jacent mérite quelques mots d'explication, parce qu'il conditionne certaines propriétés importantes du module. SymPy est une bibliothèque **purement Python** : sa seule dépendance externe est `mpmath`, qui est aussi pure Python. Cette propriété rend l'installation possible sur n'importe quel environnement Python sans étape de compilation native, y compris hors-ligne, sur des images Docker minimales, ou sur des systèmes embarqués qui n'ont pas de chaîne de compilation. C'est exactement la propriété qu'on souhaite pour un module qui peut être absent — le coût d'ajout est minimal, et le coût d'absence est nul.

Le périmètre fonctionnel de SymPy couvre tout ce qu'on attend d'un CAS moderne : résolution d'équations polynomiales et transcendantales, calcul différentiel et intégral, algèbre linéaire symbolique et numérique, équations différentielles ordinaires, rationnels exacts et radicaux, génération de rendu LaTeX, manipulation arbitraire d'arbres syntaxiques. La sérialisation round-trip des expressions vers une représentation textuelle permet en outre un transport propre des résultats à travers les frontières REST du serveur VVL — un détail qui simplifie considérablement l'intégration.

36.2. La lacune comblée : √2 reste √2

Pour fixer concrètement ce que le module apporte, voici quelques comparaisons directes entre l'arithmétique numérique ordinaire et l'arithmétique symbolique.

36.2.1. Précision préservée par construction

```
001  # Comparaison : numerique vs symbolique
002
003  # Numerique ordinaire (VVL_Num) :
004  [log [sqrt 2]]              # => 1.4142135623730951  (approximation)
005  [log (* [sqrt 2] [sqrt 2])] # => 2.0000000000000004  (erreur de 4e-16)
006
007  # Symbolique exact (VVL_Sym) :
008  $s = [symbolic:parse "sqrt(2)"]
009  [log $s]                    # => <symbolic: sqrt(2)>     (exact)
010  [log (* $s $s)]             # => <symbolic: 2>           (exact)
011
012  # Le rationnel 1/3 sous les deux formes :
013  [log (/ 1 3)]                       # => 0.3333333333333333
014  [log [symbolic:rational 1 3]]       # => <symbolic: 1/3>  (exact)
```

La différence n'est pas seulement esthétique. Quand `√2 × √2` produit `2.0000000000000004` au lieu de `2`, on perd la possibilité de tester une égalité par comparaison directe ; on doit recourir à une tolérance

numérique, ce qui ouvre la porte à des bugs subtils où une « égalité » dépend du seuil choisi. La version symbolique produit `2` strict — le test d'égalité est sans ambiguïté.

36.2.2. Résolution exacte, pas seulement numérique

```
001  # Quatre exemples qui exhibent l'eventail de l'exactitude
002
003  # Racines entieres :
004  [? [$ x^2 - 5*x + 6] x]
005  # => [2 3]
006
007  # Racines rationnelles :
008  [? [$ 2*x - 1] x]
009  # => [<symbolic: 1/2>]
010
011  # Racines complexes :
012  [? [$ x^2 + 1] x]
013  # => [<symbolic: -I> <symbolic: I>]
014
015  # Racines parametriques (formule litterale) :
016  [? [$ a*x^2 + b*x + c] x]
017  # => [<symbolic: (-b - sqrt(b^2 - 4*a*c)) / (2*a)>
018  #     <symbolic: (-b + sqrt(b^2 - 4*a*c)) / (2*a)>]
```

Le quatrième exemple est particulièrement parlant : sans valeur numérique pour `a`, `b`, `c`, un solveur numérique serait incapable de produire quoi que ce soit. Le solveur symbolique, lui, produit la **formule littérale** — la même qu'on apprend en lycée pour résoudre les trinômes du second degré. Cette formule est ensuite manipulable : on peut la substituer avec des valeurs spécifiques plus tard, la simplifier, la dériver, la sérialiser, la transmettre à un autre module.

36.3. Le type VVL_Sym, citoyen de première classe

Toutes les expressions symboliques manipulées par le module sont des instances d'un type unique nommé `VVL_Sym`. Ce type n'est pas un objet à part qu'il faudrait convertir explicitement pour interagir avec le reste du langage — c'est un **citoyen de première classe** qui se comporte, à quelques nuances près, exactement comme les autres types numériques. La figure ci-dessous présente sa position dans le système numérique étendu.

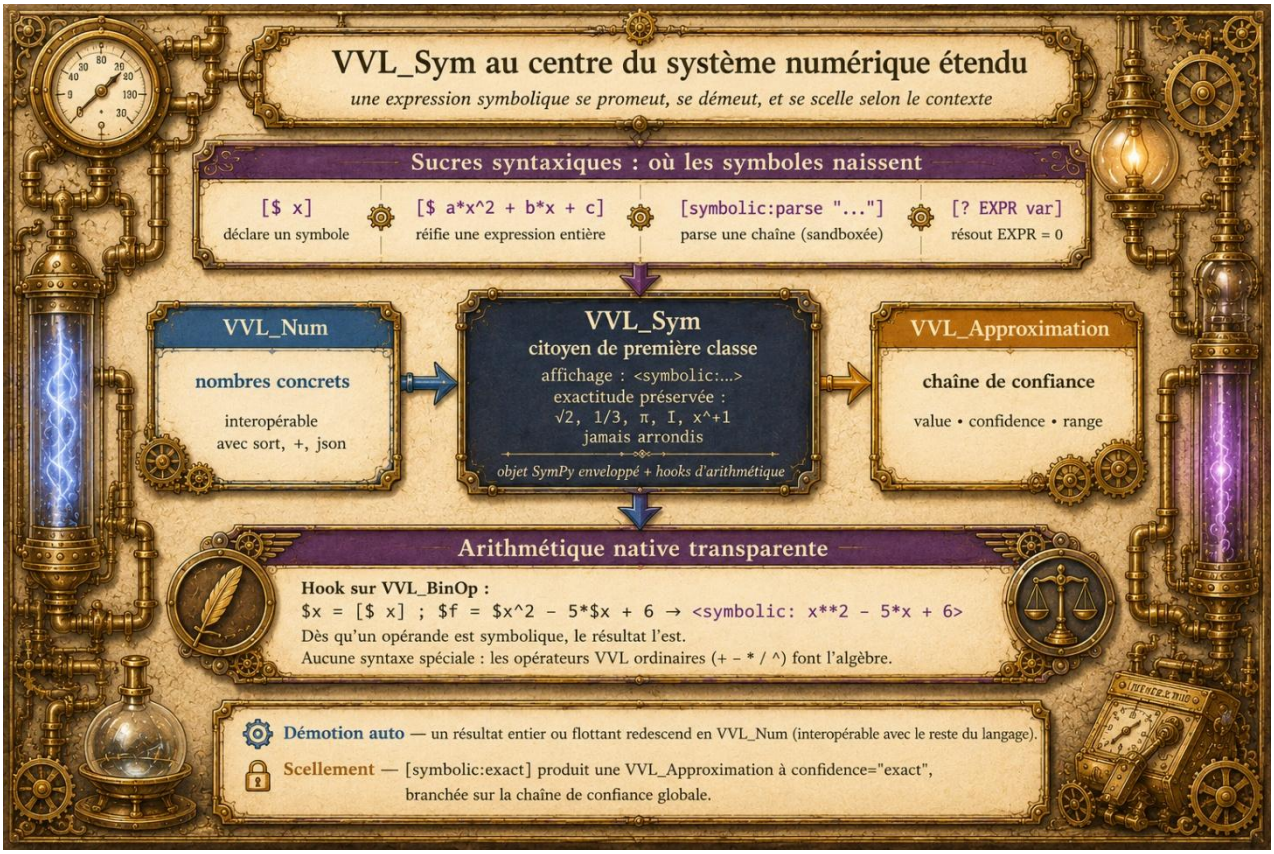


Figure 36.1 — *VVL_Sym occupe une position centrale entre VVL_Num (nombres concrets) et VVL_Approximation (chaîne de confiance). Les sucres syntaxiques créent les expressions ; l'arithmétique native les compose ; le runtime gère automatiquement la frontière exact/numérique.*

36.3.1. Affichage et représentation

Une valeur de type `VVL_Sym` s'affiche sous la forme `<symbolic: ...>`, où ... est la représentation textuelle de l'expression sympy sous-jacente. Cette représentation est intentionnellement reconnaissable : elle permet à un lecteur humain de voir au premier coup d'œil si une valeur est symbolique ou numérique, sans avoir à inspecter explicitement son type.

```
001  $a = 3                              # VVL_Num
002  [log $a]                           # => 3
003
004  $b = [symbolic:parse "sqrt(2)"]     # VVL_Sym
005  [log $b]                           # => <symbolic: sqrt(2)>
006
007  $c = [symbolic:exact $b]            # VVL_Approximation
008  [log $c.confidence]                # => "exact"
```

36.3.2. La frontière automatique entre les types

Une propriété essentielle du type `VVL_Sym` est qu'il **migre automatiquement** vers les autres types selon le contexte. Cette migration suit deux règles simples qui valent la peine d'être explicitées.

Règle 1 — démotion vers VVL_Num. Quand un calcul symbolique produit un résultat qui est *entier* ou *flottant explicite* (sans radical, rationnel, π, I ou expression libre), le résultat redescend automatiquement en `VVL_Num`. Cette démotion garantit l'interopérabilité avec tous les builtins qui

attendent des nombres ordinaires — `sort`, opérateurs arithmétiques natifs, sérialisation JSON, comparaisons. Le programmeur n'a rien à faire pour bénéficier de cette propagation.

Règle 2 — promotion vers VVL_Approximation. Le builtin `[symbolic:exact]` enveloppe explicitement une expression symbolique dans le système d'approximation, ce qui lui ajoute les métadonnées de confiance (`value`, `confidence`, `range`, `hypotheses`). Comme l'expression est exacte, la confiance produite est `"exact"` — la valeur la plus forte de l'échelle, qu'on ne peut obtenir autrement que par un calcul certifié.

Ce qui *reste* symbolique : les radicaux ($\sqrt{2}$, $\sqrt[3]{5}$), les rationnels non entiers (1/3, 22/7), les constantes mathématiques (π, e, I), les expressions à variables libres ($x^2 + 1$ avec `x` non lié). Ce qui *redescend* en numérique : les résultats entiers issus de simplifications, les flottants explicites produits par `[symbolic:numeric]`, les opérations arithmétiques qui n'introduisent aucun terme symbolique non-réductible.

36.4. Les sucres syntaxiques [$ ...] et [? ...]

Pour rendre l'usage du module agréable, le parser VVL a été étendu avec deux sucres syntaxiques dédiés. Sans ces sucres, déclarer un symbole exigerait `[symbolic:symbol "x"]` à chaque occurrence — verbeux et étranger au reste du langage. Avec les sucres, l'écriture devient `[$ x]`, qui se lit immédiatement comme *« un symbole nommé x »*. Cette section examine les cinq formes que prennent ces sucres.

36.4.1. Forme [$ x] : déclarer un symbole

La forme la plus simple. `[$ x]` produit un symbole nommé `x` qu'on peut ensuite assigner à une variable VVL pour l'utiliser dans des expressions. L'**espace** entre `$` et `x` est syntaxiquement significatif : `[$ x]` produit un symbole, mais `[$x]` (sans espace) reste une variable VVL ordinaire qui se résout par lookup. Cette distinction est essentielle, et nous y reviendrons en 36.13.2 dans la section des pièges.

```
001  $x = [$ x]                    # declaration d'un symbole
002  [log $x]                     # => <symbolic: x>
003
004  # A NE PAS CONFONDRE avec :
005  $x = 5
006  [log [$x]]                   # ERREUR : [$x] n'est pas un appel
007  #                              # la variable $x vaut 5, pas un symbole
```

36.4.2. Forme [$ x y z] : déclarer plusieurs symboles

Quand on a besoin de plusieurs symboles d'un coup — par exemple pour une équation à plusieurs inconnues — la forme `[$ x y z]` produit une **liste** de symboles dans l'ordre donné. Elle peut être déstructurée ou indexée comme une liste VVL ordinaire.

```
001  $syms = [$ a b c]             # => [<symbolic: a> <symbolic: b> <symbolic: c>]
002  [log $syms.0]                # => <symbolic: a>
003
004  # Souvent plus pratique avec une destructuration explicite :
005  $a = $syms.0
006  $b = $syms.1
```

```
007  $c = $syms.2
```

36.4.3. Forme [$ a*x^2 + b*x + c] : réifier une expression

C'est ici que la magie commence à opérer. `[$ a*x^2 + b*x + c]` ne se contente pas de déclarer des symboles individuels : il **réifie une expression entière**, où chaque mot nu (`a`, `x`, `b`, `c`) devient automatiquement un symbole. Le résultat est une expression symbolique composée, manipulable comme une seule valeur.

```
001  $trinome = [$ a*x^2 + b*x + c]
002  [log $trinome]
003  # => <symbolic: a*x**2 + b*x + c>
004
005  # On peut deriver par rapport a x :
006  [log [symbolic:diff $trinome x]]
007  # => <symbolic: 2*a*x + b>
008
009  # Ou resoudre par rapport a x :
010  [log [? $trinome x]]
011  # => les deux racines symboliques de la formule quadratique
```

Cette forme est la plus utilisée en pratique parce qu'elle évite complètement de déclarer les symboles à la main. Le moteur du module détecte les noms nus dans l'expression et les transforme en symboles **paresseusement** — c'est-à-dire qu'ils restent libres tant qu'aucune variable VVL homonyme ne reçoit une valeur. Cette liaison paresseuse est le sujet de la section 36.6.

36.4.4. Forme [? EXPR var?] : la résolution terse

Le sucre `[? …]` est un raccourci pour la résolution d'équations. L'expression `[? EXPR var]` est strictement équivalente à `[symbolic:solve EXPR var]`, mais nettement plus concise — assez concise pour rester lisible même quand on l'utilise en chaîne. La convention sémantique est que `EXPR` est interprétée comme *« EXPR = 0 »* : pour résoudre `x² − 5x + 6 = 5`, on écrit `[? [$ x^2 - 5*x + 6 - 5] x]` (réécrit comme `… − 5 = 0`).

```
001  # Forme courte
002  [? [$ x^2 - 5*x + 6] x]              # => [2 3]
003
004  # Equivalent verbeux
005  [symbolic:solve [$ x^2 - 5*x + 6] x]   # => [2 3]
006
007  # La variable est optionnelle si l'expression a UNE seule inconnue :
008  [? [$ x^2 - 1]]                      # => [-1 1]
```

36.4.5. Forme [$f arg] : appliquer une lambda symbolique

La dernière forme `[$f arg]` permet d'**appliquer** une fonction symbolique préalablement créée par `[symbolic:lambda …]`. Cette syntaxe imite l'application de fonction des langages fonctionnels, et permet d'utiliser les fonctions symboliques comme on utiliserait des keywords VVL.

```
# Definition de la fonction
$f = [symbolic:lambda x [$ x^2 + 1]]
[log $f]                  # => <symbolic: Lambda(x, x**2 + 1)>

# Application avec une valeur numerique
[log [$f 1]]              # => 2  (demote en VVL_Num)
[log [$f 3]]              # => 10

# Application avec une autre expression symbolique
[log [$f [$ y]]]          # => <symbolic: y**2 + 1>
```

Cette double-vocation — *appliquer à un nombre concret* (qui produit un nombre concret par évaluation) ou *appliquer à un autre symbole* (qui produit une nouvelle expression symbolique composée) — est l'une des plus jolies propriétés du système. Une fonction symbolique peut servir indifféremment à *calculer* ou à *composer*, selon ce qu'on lui passe en argument.

36.5. L'arithmétique native transparente

La propriété qui distingue le plus radicalement le module `symbolic/` des CAS « bibliothèque externe » habituels est l'**arithmétique native transparente**. Dans la plupart des langages qui intègrent SymPy ou un équivalent, l'utilisateur doit appeler explicitement les opérations CAS : `sympy.Add(a, b)` ou `a.__add__(b)` plutôt que `a + b`. Cette verbosité décourage l'usage et fragmente le code en deux mondes — celui des nombres natifs et celui des expressions symboliques.

VVL adopte la voie opposée : **les opérateurs natifs `+`, `-`, `*`, `/`, `^` produisent directement du symbolique** dès qu'au moins un opérande est de type `VVL_Sym`. L'utilisateur écrit ses calculs comme s'il manipulait des nombres ordinaires, et le runtime fait l'algèbre symbolique automatiquement.

```
# La syntaxe est strictement celle de l'arithmetique VVL ordinaire :
$x = [$ x]
$f = $x^2 - 5*$x + 6
[log $f]
# => <symbolic: x**2 - 5*x + 6>

# Toutes les operations sont disponibles :
$g = ($f * $x) + 7
[log $g]
# => <symbolic: x*(x**2 - 5*x + 6) + 7>

# La factorisation simplifie automatiquement quand c'est possible :
[log [symbolic:simplify $g]]
# => <symbolic: x**3 - 5*x**2 + 6*x + 7>
```

36.5.1. Le mécanisme interne : hook sur VVL_BinOp

Sous le capot, l'évaluation d'une opération binaire VVL passe par le nœud d'arbre `VVL_BinOp`, qui exécute l'opérateur sur les deux valeurs après leur résolution. Quand au moins une des deux valeurs

est de type `VVL_Sym`, un **hook d'enveloppement** intercepte le résultat : la valeur sympy native (de type `sympy.Basic`) produite par l'opération est ré-emballée en `VVL_Sym` avant d'être propagée au reste de l'évaluation. C'est ce hook qui rend l'arithmétique transparente — l'utilisateur n'a pas à s'occuper de la conversion.

Ce mécanisme est rendu possible par une propriété de SymPy : l'objet enveloppé par `VVL_Sym` est le `sympy.Basic` lui-même, qui supporte nativement tous les opérateurs Python (`__add__`, `__sub__`, etc.) sur des opérandes hétérogènes. Quand le runtime VVL délègue l'opération à Python, sympy fait son travail tout seul : `sympy_expr + 1` produit un nouveau `sympy.Basic`, et le hook le détecte pour le ré-emballer. La chaîne est complètement automatique.

36.5.2. Conséquence : le code reste lisible

La conséquence pratique de cette transparence est qu'**un programme VVL qui utilise du symbolique reste, syntaxiquement, un programme VVL ordinaire**. La règle physique `Ec = ½ m v²` se traduit en VVL par `$ec = [$ m * v^2 / 2]` ou, après déclaration des variables, par `$ec = $m * $v^2 / 2`. Aucune syntaxe nouvelle, aucun mot-clé exotique, aucune chaîne à parser. Cette discipline rend le module **utilisable** dans des bases de code existantes sans réécriture en cascade : il suffit d'introduire un `[$ …]` au bon endroit pour qu'une expression devienne symbolique, et tout le reste s'adapte.

36.6. La liaison paresseuse

L'une des propriétés les plus élégantes du module est ce qu'on appelle la **liaison paresseuse** (*lazy binding*) : un symbole nommé `x` se résout vers la variable VVL homonyme `$x` **uniquement quand celle-ci reçoit une valeur**. Tant que `$x` n'est pas définie, le symbole reste libre. Dès que `$x = qqch`, le symbole se *réveille* et toutes les expressions qui le contiennent se résolvent automatiquement.

36.6.1. Le mécanisme en action

```
001  # Etat initial : x est un symbole libre
002  $x = [$ x]
003  $g = [$ x + 1]              # ou $x + 1 par arithmetique native
004  [log $g]                   # => <symbolic: x + 1>     (libre)
005
006  # On donne une valeur a $x :
007  $x = 1
008  [log $g]                   # => 2                     (resolu)
009
010  # On peut rebinder :
011  $x = 5
012  [log $g]                   # => 6                     (re-resolu)
013
014  # Et on peut revenir au symbolique en redeclaration :
015  $x = [$ x]
016  [log $g]                   # => <symbolic: x + 1>     (libre a nouveau)
```

Cette propriété transforme le module en **système de calcul paramétrique** sans aucun mécanisme supplémentaire. On définit une formule abstraite (par exemple l'énergie cinétique `Ec = m v² / 2`), on inspecte la formule symbolique tant qu'elle reste abstraite, puis on fournit les paramètres au fur et à

mesure pour obtenir une valeur numérique. La même expression est tour à tour *formule* et *résultat*, selon l'état des variables.

36.6.2. Pourquoi la liaison paresseuse est précieuse

L'avantage le plus immédiat est l'**économie cognitive**. Sans liaison paresseuse, l'utilisateur devrait appeler explicitement `[symbolic:subs]` à chaque substitution. Avec elle, il suffit d'affecter la variable VVL et le moteur fait la substitution automatiquement à la prochaine évaluation.

Un avantage moins évident mais plus profond est qu'**une formule et un calcul partagent le même nom dans le code**. Considérons le bloc suivant :

```
# Une seule formule pour deux usages
$ec = [$ m * v^2 / 2]

# Usage 1 : inspecter la formule (m, v non lies)
[log [symbolic:latex $ec]]
# => "\\frac{m v^{2}}{2}"

# Usage 2 : calculer pour des valeurs concretes
$m = 2
$v = 3
[log $ec]
# => 9
```

La même variable `$ec` joue deux rôles successifs dans le même fil d'exécution : *expression à inspecter* tant que les paramètres sont libres, puis *valeur numérique* dès qu'ils sont fixés. Cette dualité simplifie énormément les patrons d'utilisation paramétrique, qui sont fréquents dans les modèles physiques, économiques, ou en simulation.

36.7. Catalogue des builtins — calcul élémentaire et algèbre linéaire

Au-delà des sucres syntaxiques, le module expose une vingtaine de builtins préfixés `[symbolic:*]`. La figure ci-dessous les regroupe en cinq familles fonctionnelles ; cette section les examine en détail.



Figure 36.2 — *Les builtins du module symbolic, organisés en cinq familles : sucres syntaxiques et parse ; calcul élémentaire ; algèbre linéaire ; fonctions et ODE ; pont vers les autres types VVL.*

36.7.1. Famille « calcul élémentaire »

Cette famille couvre la majorité des usages quotidiens d'un CAS : résolution d'équations, calcul différentiel et intégral, manipulation algébrique. Les builtins principaux sont les suivants.

Builtin	Effet
[symbolic:solve eq var?]	résout eq = 0, retourne la liste des solutions
[symbolic:solveset eq var]	idem mais retourne un ensemble-solution (domaine réel)
[symbolic:diff e var n?]	dérivée n-ième de e par rapport à var
[symbolic:integrate e var (a b)?]	primitive ou intégrale définie sur [a, b]
[symbolic:limit e var to]	limite de e quand var → to
[symbolic:simplify e]	réécriture canonique de l'expression
[symbolic:expand e]	développement des produits et puissances
[symbolic:factor e]	factorisation polynomiale
[symbolic:subs e v1 x1 …]	substitution (une ou plusieurs paires simultanées)
[symbolic:eq lhs rhs]	construit une équation Eq(lhs, rhs)
[symbolic:rational p q]	rationnel exact p/q
[symbolic:numeric e digits?]	évaluation numérique avec digits chiffres

Quelques exemples concrets qui montrent comment ces builtins se composent.

```
# Calcul differentiel et integral
[symbolic:diff [$ x^2 - 5*x + 6] x]
# => <symbolic: 2*x - 5>

[symbolic:integrate [symbolic:parse "1/(1+x**2)"] x]
# => <symbolic: atan(x)>

[symbolic:limit [symbolic:parse "sin(x)/x"] x 0]
# => 1

# Identites trigonometriques
[symbolic:simplify [symbolic:parse "sin(x)**2 + cos(x)**2"]]
# => 1

# Substitution multi-variables simultanees
$ec = [$ m * v^2 / 2]
[symbolic:subs $ec m 2 v 3]
# => 9

# Substitution partielle
[symbolic:subs $ec v 10]
# => <symbolic: 50*m>     (m reste libre)
```

36.7.2. Famille « algèbre linéaire »

Les matrices symboliques ouvrent un champ de manipulation que les matrices numériques ne couvrent pas : on peut calculer un déterminant ou résoudre un système avec des coefficients littéraux, ce qui produit des **formules** plutôt que des nombres.

Builtin	Effet
[symbolic:matrix <[[…]]>]	construit une matrice depuis un littéral JSON
[symbolic:det M]	déterminant
[symbolic:inv M]	inverse
[symbolic:transpose M]	transposée
[symbolic:solve-linear A b]	résolution du système linéaire A·x = b

```
# Determinant d'une matrice numerique
[symbolic:det <[[1,2],[3,4]]>]
# => -2

# Determinant d'une matrice litterale (resultat symbolique)
$a = [$ a]   $b = [$ b]   $c = [$ c]   $d = [$ d]
[symbolic:det [list [list $a $b] [list $c $d]]]
# => <symbolic: a*d - b*c>

# Resolution d'un systeme : x + y = 10, x - y = 2
[symbolic:solve-linear <[[1,1],[1,-1]]> <[10,2]>]
# => [6 4]
```

Le contraste entre les deux usages du `det` est éclairant. Avec des coefficients entiers, on obtient un nombre — le déterminant calculé. Avec des coefficients littéraux, on obtient la **formule du déterminant** — `ad - bc` pour une matrice 2×2. Cette dernière forme est précieuse pour la démonstration ou l'enseignement, et n'est accessible qu'à un système symbolique.

36.8. Catalogue des builtins — fonctions, ODE et pont

Les deux dernières familles du catalogue couvrent des usages plus avancés mais structurellement importants : la déclaration de fonctions (définies ou indéfinies), la résolution d'équations différentielles, et le pont entre `VVL_Sym` et les autres types du système.

36.8.1. Famille « fonctions et ODE »

Builtin	Effet
[symbolic:lambda x BODY]	définit une fonction d'argument x ; appliquée par [$f arg]
[symbolic:fn f x [y …]]	fonction **indéfinie** f(x, …) (utile pour dériver, ODE)
[symbolic:assume x "…"]	symbole portant une hypothèse (positive, real, integer…)
[symbolic:dsolve eq func]	résout une équation différentielle ordinaire

La distinction entre `lambda` et `fn` mérite un commentaire. `lambda` crée une **fonction définie** : son corps est une expression connue, et l'application produit une valeur (numérique ou symbolique) par substitution. `fn`, à l'inverse, crée une **fonction indéfinie** — une fonction symbolique abstraite dont on ne connaît que le nom et la signature. Cette deuxième forme est essentielle pour exprimer des équations différentielles, où l'on cherche précisément à *trouver* la fonction qui satisfait l'équation.

```
# Fonction definie
$f = [symbolic:lambda x [$ x^2 + 1]]
[log [$f 1]]              # => 2
[log [$f 3]]              # => 10
[log [$f [$ y]]]          # => <symbolic: y**2 + 1>

# Fonction indefinie + equation differentielle f'(x) = f(x)
$fx  = [symbolic:fn f x]                  # <symbolic: f(x)>
[log [symbolic:diff $fx x]]             # => Derivative(f(x), x)

$ode = [symbolic:eq [symbolic:diff $fx x] $fx]
[log [symbolic:dsolve $ode $fx]]
# => <symbolic: Eq(f(x), C1*exp(x))>
```

Le dernier exemple — l'équation différentielle `f'(x) = f(x)` — produit la famille de toutes les fonctions exponentielles $f(x) = C_1 \cdot e^x$, où C_1 est une constante d'intégration libre. Ce résultat est mathématiquement exact, et il a été obtenu sans intervention numérique d'aucune sorte. Le solveur ODE de SymPy fait le travail entièrement en symbolique.

36.8.1. Famille « pont vers les autres types »

Builtin	Effet
[symbolic:numeric e digits?]	évalue numériquement (→ VVL_Num)
[symbolic:exact EXPR label?]	scelle en VVL_Approximation confidence="exact"
[symbolic:string e]	rendu textuel (chaîne)
[symbolic:latex e]	rendu LaTeX

`[symbolic:exact]` est le mécanisme essentiel qui sera examiné en détail à la section 36.10 : il permet d'intégrer un résultat symbolique exact dans la chaîne de confiance globale du langage. Le résultat reçoit automatiquement le label `confidence = "exact"`, qui se propage à travers les calculs ultérieurs dans le système d'approximation.

36.8.2. Le builtin sous-jacent [symbolic:parse]

Le builtin `[symbolic:parse "…"]` mérite une mention dédiée, parce qu'il joue un rôle particulier. Il prend une chaîne de caractères et la parse en expression symbolique, en utilisant un sous-ensemble strictement contrôlé de la syntaxe Python/SymPy. Sont disponibles : les opérateurs arithmétiques `+ - * / ** %`, les fonctions standard `sin cos tan exp log sqrt`, les constantes `pi` et `E`, l'unité imaginaire `I`.

```
# Exemple typique : parsing d'une expression complexe
[symbolic:parse "sin(x)*exp(x) - cos(x)"]
# => <symbolic: -cos(x) + sin(x)*exp(x)>

# Avec sqrt et pi
[symbolic:parse "sqrt(2) * pi / 4"]
# => <symbolic: sqrt(2)*pi/4>

# ATTENTION : la puissance s'ecrit ** dans le parse, pas ^
[symbolic:parse "x**2 + 1"]    # OK
[symbolic:parse "x^2 + 1"]     # ERREUR : ^ est XOR en Python
```

L'asymétrie entre `^` (en code VVL) et `**` (dans une chaîne `parse`) est l'une des petites bizarreries du module : elle découle directement de la convention Python que SymPy hérite. On peut la voir comme un défaut, mais elle est conservée plutôt que masquée parce qu'elle rend les chaînes `parse` directement copiables vers et depuis un shell Python — propriété utile pour le débogage.

36.9. Propriétés accessibles sur VVL_Sym

Au-delà des builtins, une expression symbolique expose un ensemble de **propriétés** accessibles par la notation pointée habituelle. Ces propriétés permettent un usage plus fluide quand on souhaite enchaîner plusieurs opérations sur la même valeur sans passer par des builtins intermédiaires.

Propriété	Effet
.simplify	forme simplifiée canonique
.expand	développement des produits
.factor	factorisation polynomiale
.numeric	évaluation numérique (si pas de variable libre)
.latex	rendu LaTeX
.string	rendu textuel
.free	liste des variables libres restantes
.json	sérialisation {sym, latex, free, srepr}
.is_number	booléen : vrai si évaluable à un nombre concret

L'usage typique combine les propriétés pour produire une transformation directe d'une expression vers la forme attendue :

```
$h = [symbolic:parse "(x+1)**2"]

[log $h.expand]              # => <symbolic: x**2 + 2*x + 1>
[log $h.factor]              # => <symbolic: (x + 1)**2>
[log $h.free]                # => ["x"]
[log $h.latex]               # => "\\left(x + 1\\right)^{2}"
[log $h.is_number]           # => 0  (faux, x est libre)

# Substituer puis tester is_number :
$k = [symbolic:subs $h x 5]
[log $k.is_number]           # => 1  (36 est un nombre)
[log $k.numeric]             # => 36
```

Le champ `.free` est particulièrement utile pour la programmation défensive : avant de tenter une évaluation numérique, on peut vérifier que l'expression n'a pas de variable libre résiduelle. Cette vérification évite les erreurs *en aval* qui seraient déclenchées par un appel à `.numeric` sur une expression encore paramétrique.

Le champ `.json` est précieux pour la **sérialisation** : il produit une représentation portable qui contient à la fois la chaîne sympy (`srepr`, qui peut être reparsée pour reconstruire l'objet), la chaîne LaTeX pour le rendu humain, et la liste des variables libres. Cette structure peut être transmise via REST, stockée dans une base de connaissance, ou échangée entre instances VVL sans perte d'information.

36.10. Le pont avec VVL_Approximation : la confiance « exact »

Le module ne vit pas isolé du reste du langage. Sa pièce d'intégration la plus importante est le **pont avec le système d'approximation** : le builtin `[symbolic:exact]` permet à un résultat symbolique de circuler dans la chaîne de confiance globale de VVL, avec la garantie qu'il sera marqué comme **exact**.

36.10.1. La sémantique de [symbolic:exact]

`[symbolic:exact EXPR]` enveloppe une expression symbolique dans un objet `VVL_Approximation` qui expose les champs standard de ce type : `value` (la valeur centrale), `confidence` (l'étiquette de confiance), `range` (la fourchette), et `hypotheses` (la liste des hypothèses qui ont contribué au résultat). La nuance essentielle est que la **confidence** est calculée d'après le poids des hypothèses contributrices, et qu'un résultat purement symbolique a, par construction, **zéro hypothèse de source « LLM »**.

```
# Resolution exacte enveloppee en VVL_Approximation
$x = [$ x]
$r = [symbolic:exact [? [$ a*x^2 + b*x + c] $x]]

[log $r.value]
# => les deux racines symboliques de la formule quadratique

[log $r.confidence]
# => "exact"

# Tant que les parametres a, b, c restent libres, le resultat
# est une formule. Une fois substitues, il devient un nombre :
$f = $r.value.0    # premiere racine
[log [symbolic:subs $f a 1 b -5 c 6]]
# => 2  (numerique, demote en VVL_Num)
```

Cette propriété — *un résultat CAS est exact gratuitement et honnêtement* — est l'une des conséquences les plus précieuses du design en couches du système. La règle est simple à énoncer : **la confiance ne dépend que du poids des hypothèses LLM contributrices**. Si aucun appel LLM n'a été nécessaire pour produire la valeur, la confiance est `"exact"`. Si certaines quantités ont été estimées par LLM, la confiance descend proportionnellement à leur poids relatif.

36.10.2. Pourquoi cette discipline est honnête

Le mot *honnête* dans la formule ci-dessus mérite d'être déballé. Une approche naïve aurait fait du CAS un système *toujours exact* — ce qui est techniquement vrai au sens algébrique, mais pratiquement faux dès qu'on l'utilise dans un calcul qui mélange l'exact et l'approché. Considérons l'exemple suivant : on veut calculer le coût annuel d'un trajet en voiture de 12 000 km avec une consommation de 6 L/100 km et un prix de carburant estimé.

```
# La formule est exacte (litres × prix unitaire) :
$prix_litre = [concept:approx [@ "carburant"] "prix_moyen_en_france"]
# Cette valeur vient d'un LLM qui estime ~1.85€/L
# avec confidence="moderate"

$conso = 6                    # L/100km (donnee)
$dist  = 12000                # km (donnee)
# Calcul mixte symbolique + estimation
$cout = ($dist * $conso / 100 * $prix_litre)

[log $cout.confidence]
# => "moderate"

# La confiance est dégradée par la SEULE hypothèse LLM
# (le prix du carburant). La structure du calcul est exacte,
# mais le résultat n'est pas plus sûr que la quantité la moins sûre.
```

Cette propagation est ce qu'on appelle, dans la littérature de calcul incertain, un **chainage de confiance honnête** : la confiance d'un résultat ne peut pas dépasser celle de son intrant le moins certain. Le système refuse de produire une réponse plus confiante que la moins confiante de ses entrées. Cette discipline empêche les abus typiques où un calcul « précis » à 8 décimales masquerait une estimation grossière à 10 % près quelque part en amont.

36.11. Intégration au langage naturel via [resolve]

L'autre pièce d'intégration majeure est le pont avec le langage naturel via le builtin `[resolve]`. Ce builtin — décrit en détail dans des chapitres antérieurs de l'ouvrage — prend une question en français et produit un programme VVL équivalent qu'il évalue. Avec l'arrivée du module symbolic, `[resolve]` est devenu capable de produire des programmes **symboliques par défaut**, ce qui change qualitativement la nature des réponses possibles. La figure ci-dessous présente la boucle.



Figure 36.3 — *La boucle neuro-symbolique de [resolve] : le LLM traduit la question en programme VVL symbolique ; SymPy calcule exactement ; le résultat porte une confiance honnête, dégradée seulement par le poids des estimations LLM intervenues.*

36.11.1. La politique « symbolique par défaut »

Le compilateur NL → VVL embarqué dans `[resolve]` suit une règle explicite, encodée dans son prompt système : *quand une quantité n'est pas connue numériquement, elle reste un symbole libre*. Cette règle empêche le LLM de combler les inconnues avec des estimations spontanées — ce qui produirait des réponses fausses à confiance faussée. À la place, la question est traduite en expression symbolique que l'utilisateur pourra substituer plus tard avec ses propres valeurs.

```
# Question 1 : maths pures (aucune quantite inconnue)
[resolve "résous x^2 - 5x + 6 = 0"]
# => [2 3]   ; confidence = "exact"

# Question 2 : formule physique (m et v non donnes)
[resolve "énergie cinétique de masse m et vitesse v"]
# => <symbolic: m*v**2/2>   ; confidence = "exact"

# Question 3 : application de fonction
[resolve "soit f(x) = x^2 + 1, combien vaut f(3) ?"]
# => 10   ; confidence = "exact"
```

Les trois exemples illustrent les trois cas typiques de résolution exacte. Dans le premier, la question est mathématique pure — il n'y a rien à estimer, SymPy calcule directement. Dans le deuxième, la formule contient des paramètres inconnus, donc le résultat est une **expression** (la formule symbolique elle-même) que l'utilisateur pourra évaluer en substituant `m` et `v` plus tard. Dans le troisième, une fonction est définie et appliquée, le résultat est numérique.

36.11.2. Le mode estimation explicite

Quand l'utilisateur **demande explicitement** une estimation — par exemple en utilisant les mots « estime », « environ », « ordre de grandeur » dans sa question —, le compilateur autorise le LLM à réifier les quantités inconnues comme des hypothèses LLM. La confiance du résultat est alors dégradée en conséquence.

```
# Question avec intention d'estimation
[resolve "estime le coût carburant d'une Clio sur 12000 km"]

# Le LLM reifie les quantites necessaires :
#   prix_litre   ~ 1.85 €/L   (hypothese LLM, source="LLM")
#   consommation ~ 6 L/100km  (hypothese LLM, source="LLM")
#
# Calcul symbolique : 12000 / 100 * 6 * 1.85
#
# => 1332.0   ; confidence = "moderate"
#   value     1332
#   range     [1100, 1600]
#   confidence "moderate"
```

Cette double politique — *symbolique par défaut, estimation sur intention explicite* — est ce qui rend le couplage LLM + CAS **honnête** au sens où on l'a déballé en 36.10.2. L'utilisateur obtient des réponses calibrées à ce qu'il a vraiment demandé : des formules exactes quand la question est mathématique, des estimations honnêtement labellisées quand il sollicite des approximations.

36.12. Sécurité du sandbox AST

Le builtin `[symbolic:parse]` accepte une chaîne arbitraire en entrée et la parse en expression sympy. Cette fonctionnalité — indispensable pour permettre la sérialisation et l'échange d'expressions — ouvrirait une faille de sécurité majeure si elle était implémentée naïvement. Cette section explique la menace et la défense.

36.12.1. La menace : exécution de code arbitraire

La fonction Python `parse_expr` de SymPy, utilisée par défaut pour transformer une chaîne en expression symbolique, **évalue l'AST Python** au moment du parse. Cette évaluation est ce qui permet à `parse_expr("sin(x)")` de retourner directement un objet `sin(Symbol('x'))` plutôt qu'un AST brut. Mais cette évaluation laisse passer **tout ce qui est valide en Python** — y compris des constructions qui n'ont rien d'algébrique.

L'attaque la plus connue contre `parse_expr` est ce qu'on appelle le *gadget d'évasion par introspection*, qui exploite la chaîne d'héritage des classes Python pour atteindre des fonctions système :

```
# UNE ATTAQUE ILLUSTRATIVE — JAMAIS A FAIRE EN PRODUCTION
#
# Une chaine apparemment innocente :
# "().__class__.__bases__[0].__subclasses__()[XXX]( ... )"
#
# Avec un index XXX bien choisi, on peut atteindre
# subprocess.Popen, os.system, eval, exec...
# soit un RCE (Remote Code Execution) trivial.

# Si on appliquait parse_expr nu sur une telle chaine,
# SymPy evaluerait l'AST Python et l'attaque reussirait.
```

Cette classe d'attaque est documentée publiquement depuis plusieurs années et touche tous les langages qui exposent une interface `parse_expr` sans précaution. Pour un serveur VVL qui exposerait `[symbolic:parse]` via un endpoint REST, c'est une vulnérabilité critique.

36.12.2. La défense : validation AST par liste blanche

Le module `symbolic/` applique une **validation préalable** de l'AST Python avant d'appeler `parse_expr`. Cette validation suit une approche par **liste blanche stricte** : seuls les nœuds AST explicitement autorisés sont acceptés ; tout le reste est rejeté avec une erreur claire.

Concrètement, les nœuds autorisés sont : les littéraux numériques (entiers, flottants), les noms (limités à une liste blanche incluant les symboles utilisateurs et les fonctions standard comme `sin`, `cos`, `exp`, `log`, `sqrt`, plus `pi`, `E`, `I`), les opérateurs arithmétiques (`+`, `-`, `*`, `/`, `**`, `%`, unaires), les appels de fonctions whitelistées, et les structures de regroupement (parenthèses, opérateurs ternaires si nécessaire).

Sont **explicitement rejetés** : tous les accès d'attribut (`.foo`), toutes les références dunder (`__bases__`, `__class__`, etc.), l'indexation (`[…]`), les lambdas et compréhensions, les appels à des fonctions non-whitelistées, et tout ce qui ressemble à du contrôle de flux Python (`if`, `for`, `while`). Cette liste blanche est volontairement très restrictive — il est plus sûr d'élargir progressivement la liste au fil des besoins que de tenter de la rendre exhaustive d'emblée et d'introduire accidentellement une faille.

Le placement de la validation **avant** `parse_expr` est essentiel. Une validation a posteriori — qui inspecterait l'expression résultante pour détecter d'éventuelles aberrations — serait trop tard : `parse_expr` aurait déjà exécuté le code malveillant. La discipline est donc *valider la structure syntaxique, puis exécuter le parsing une fois la structure validée*.

36.13. Notes de syntaxe, pièges, et articulation finale

Avant de conclure le chapitre, deux sections de portée pratique : les pièges syntaxiques à connaître, et l'articulation du module avec les autres composantes de l'ouvrage.

36.13.1. Pièges syntaxiques connus

Trois pièges récurrents méritent d'être nommés explicitement.

Piège 1 — l'espace dans `[$ x]`. L'espace entre `$` et `x` est syntaxiquement significatif. `[$ x]` est le sucre symbolique qui crée un symbole. `[$x]` (sans espace) reste une expression VVL ordinaire où `$x` est une **variable**. Cette distinction prend tout le monde au dépourvu au moins une fois ; elle est causée par la nécessité de distinguer le sucre `$` de la sigile de variable habituelle.

Piège 2 — la puissance dans `parse`. En code VVL natif, la puissance s'écrit avec le caret : `$v^2`. Mais à l'intérieur d'une chaîne passée à `[symbolic:parse "…"]`, la syntaxe suit la convention Python/SymPy et utilise le double astérisque : `"x**2"`. Écrire `"x^2"` dans une chaîne `parse` provoque une erreur (`^` est l'opérateur XOR en Python). C'est désagréable mais cohérent — la chaîne est *du Python*, le reste est *du VVL*.

Piège 3 — les fonctions indéfinies via parse. On pourrait être tenté d'écrire `[symbolic:parse "f(x)"]` pour créer une fonction indéfinie `f` d'argument `x`. Le sandbox AST rejette cette construction (l'appel à `f` n'est pas dans la whitelist et `f` n'est pas un symbole ordinaire). Le builtin dédié pour ce cas est `[symbolic:fn f x]`, qui produit explicitement la fonction indéfinie attendue.

36.13.2. Articulation avec les autres modules

Le module `symbolic/` s'articule naturellement avec plusieurs autres composantes de l'ouvrage, sans se confondre avec aucune.

Module	Différence avec symbolic/
Logique floue (ch. 25)	incertitude graduée par appartenance ; pas de manipulation algébrique exacte
Algorithmes génétiques (ch. 26)	recherche heuristique dans un espace de candidats ; pas de calcul algébrique
ILP (ch. 27)	induction de la forme de règles depuis des exemples ; pas de calcul algébrique
Système d'approximation	calibration de l'incertitude par LLM ; complète, ne remplace pas
Bayes (ch. 35)	calibration de poids par évidence ; orthogonal

Le placement précis de `symbolic/` est celui d'un **étage du système numérique** plutôt que d'une *voie d'apprentissage*. C'est ce qui le distingue du chapitre 35 sur le bayésien : ce dernier apprend des poids depuis des observations, tandis que le CAS calcule exactement des expressions algébriques. Les deux modules adoptent la même discipline d'extension additive — mais ils opèrent dans des dimensions complètement différentes du langage.

36.13.1. Synthèse et clôture

Ce chapitre additionnel a introduit le **calcul symbolique exact** comme nouvelle dimension du système numérique de VVL. Le module `symbolic/` apporte trois choses qui n'existaient pas auparavant : la **préservation de la précision algébrique** ($\sqrt{2}$ reste $\sqrt{2}$), la **résolution exacte** d'équations (entières, rationnelles, complexes, paramétriques), et le **calcul paramétrique** où les formules sont des citoyens de première classe.

Sur le plan technique, l'intégration est rendue économique par trois choix de design qui méritent d'être retenus. **Premier choix** : un type unique `VVL_Sym` qui se promeut et se démeut automatiquement vers les autres types selon le contexte, sans intervention de l'utilisateur. **Deuxième choix** : une arithmétique native transparente, où les opérateurs `+ - * / ^` ordinaires produisent du symbolique dès qu'au moins un opérande est symbolique — aucun appel CAS explicite n'est nécessaire. **Troisième choix** : une liaison paresseuse, qui permet à une même expression d'être tour à tour formule abstraite et résultat concret, selon l'état des variables qu'elle contient.

L'intégration avec le reste du langage se fait à deux niveaux. Vers le système d'approximation, `[symbolic:exact]` scelle un résultat CAS comme `confidence="exact"` — la valeur maximale de l'échelle de confiance, qu'on ne peut obtenir autrement. Vers le langage naturel, `[resolve]` adopte une politique *symbolique par défaut, estimation sur intention explicite* qui produit des réponses calibrées à ce que l'utilisateur a vraiment demandé.

Le module est intégré au serveur courant, aux côtés de la couche bayésienne du chapitre 35. Sa surface est stabilisée par l'usage qu'en font les autres sous-systèmes : le solveur de contraintes symboliques de la logique relationnelle (chapitre 38) et le prouveur réel de la preuve formelle (chapitre 37) s'appuient tous deux sur ce paquet. Cette dépendance croisée est le meilleur garde-fou contre une dérive d'API — une modification de surface casserait immédiatement deux moteurs, ce que la suite de tests détecte.

Le **mot final** strict du parcours d'apprentissage — l'auteur leur souhaite beaucoup de plaisir — clôt le chapitre 35 plutôt que celui-ci. Cette discipline a été expliquée à la fin du chapitre 34 et tient en une phrase : le chapitre 35 ferme le parcours des quatre paradigmes d'écriture des règles, tandis que le présent chapitre 36 ferme celui du système numérique. Les deux clôtures se complètent sans se subordonner. Les chapitres 37 à 40, qui suivent, relèvent d'un autre registre : ils décrivent l'infrastructure — vérification, résolution, exécution native, distribution — sur laquelle tout ce qui précède s'appuie désormais.

Pour la suite — de nouveaux sous-systèmes, une consolidation de ceux que ces chapitres décrivent, ou des évolutions encore non anticipées — l'auteur préfère laisser le mot à ceux qui prendront la plateforme en main. La discipline qui a guidé ce chapitre, comme celle de tous les autres, reste la même : la **cohérence** prime sur la complétude ; les **outils** priment sur les manifestes ; et les **conventions additives** priment sur les ruptures architecturales. Cette discipline est, au fond, la promesse que ce livre essaie de tenir d'un bout à l'autre.

37. La preuve formelle native : vérifier ce que le LLM propose

Mots-clés : preuve formelle, critère de de Bruijn, noyau de confiance, soundness, proof:auto, proof:check, contrat, invariant, provenance graduée, SAT, CAS

Le chapitre 1 a posé un diagnostic en trois termes : les grands modèles de langage hallucinent, ils sont opaques, et rien en eux ne distingue une réponse vraie d'une réponse plausible. Les chapitres suivants ont répondu aux deux premiers termes. À l'hallucination, VVL oppose l'ancrage symbolique : une règle Prolog inférée est vraie ou fausse, pas vraisemblable. À l'opacité, il oppose la trace — le protocole d'atomes du chapitre 23 rend chaque valeur introspectable, et l'on peut demander à n'importe quel résultat d'où il vient. Le troisième terme, la non-vérifiabilité, est resté sans réponse jusqu'ici.

C'est l'objet de ce chapitre. VVL embarque un système de preuve formelle natif, inspiré des assistants de preuve — Lean, Coq, Isabelle — mais taillé pour un usage précis : vérifier du code et des connaissances produits par un LLM. Il ne s'agit pas de formaliser les mathématiques ; il s'agit de répondre, sur une propriété donnée d'un programme donné, à la question « est-ce que cela tient ? » — et de répondre honnêtement, y compris quand la réponse est « je ne sais pas ».

Le chapitre suit la construction du système en six couches, mais dans un ordre pédagogique plutôt que numérique : les propositions comme données (37.3), la décision automatique qui est la porte d'entrée pratique (37.4), le noyau de confiance qui en est le fondement théorique (37.5), la boucle qui referme le dispositif sur le LLM (37.6), puis l'entailment relationnel (37.7), les contrats et invariants (37.8), et enfin la provenance graduée (37.9) qui est le véritable différenciateur de l'approche.

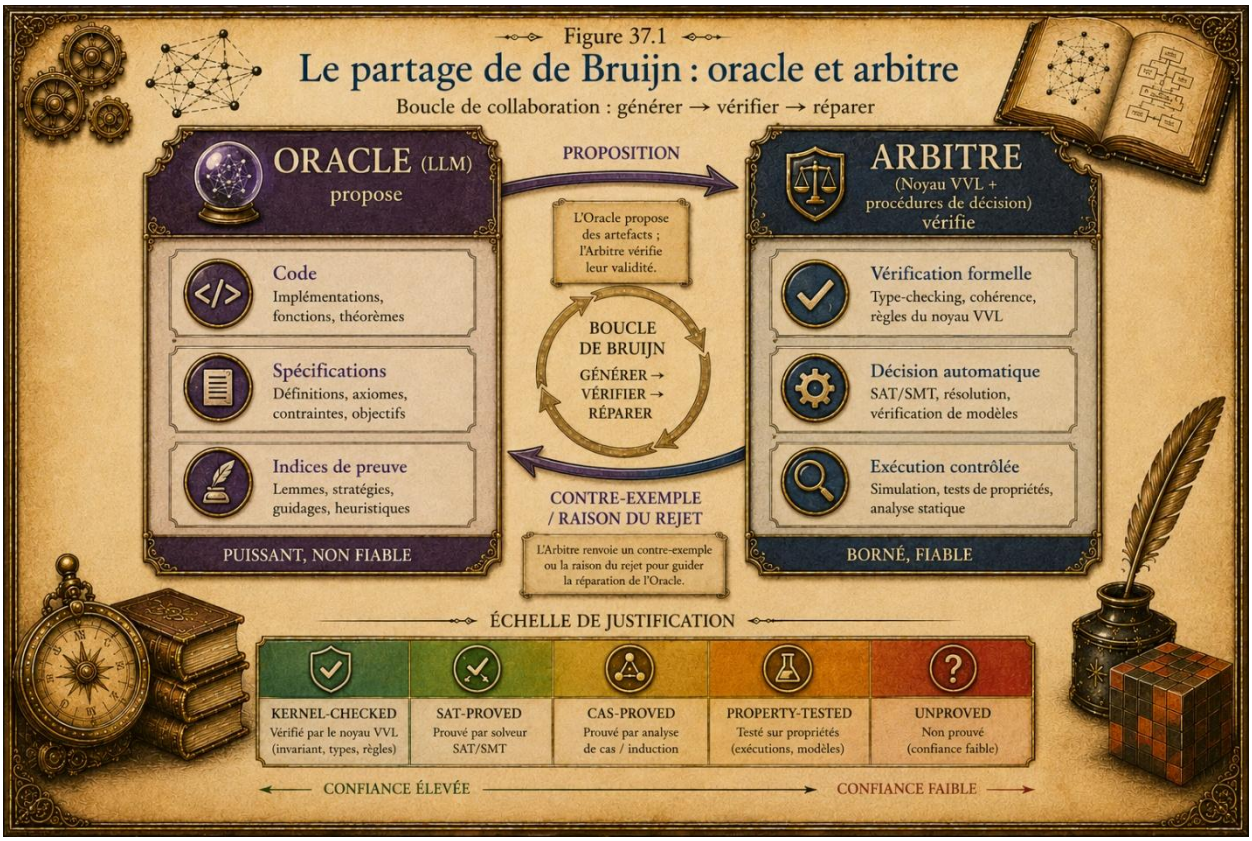


Figure 37.1 — Qui trouve, qui vérifie : la répartition des rôles héritée de de Bruijn

37.1. Le critère de de Bruijn : trouver n'est pas vérifier

L'enseignement transposable des assistants de preuve n'est pas la théorie des types dépendants. C'est une observation d'ingénierie, formulée par Nicolaas Govert de Bruijn dans le cadre du projet Automath : trouver une preuve est difficile et peu fiable, mais la vérifier est facile et sûr — à condition de disposer d'un petit noyau de confiance auditable. Toute la crédibilité du système repose alors sur ce noyau, et sur lui seul : le reste de la machinerie peut être aussi gros, aussi heuristique et aussi faillible qu'on veut, puisque rien n'est admis sans passer par le vérificateur.

Cette structure décrit exactement la situation d'un LLM qui génère du code. Le modèle est un oracle : il propose une implémentation, une spécification, parfois même un schéma de preuve, et il le fait avec une aisance qu'aucune méthode symbolique n'égale. Mais il ne garantit rien. En face, un vérificateur borné — quelques centaines de lignes de code et des procédures de décision — ne propose rien mais ne se trompe pas. Le système VVL organise cette division du travail au lieu de la subir.

La conséquence pratique est la boucle générer → spécifier → vérifier → réparer. Quand le vérificateur rejette, il ne se contente pas de dire non : il rend un contre-exemple concret ou la raison précise du rejet. Cette information est du feedback exploitable, réinjectable dans le prompt du tour suivant. Le LLM n'apprend pas à être fiable ; il apprend, à chaque itération, ce qui n'a pas marché. C'est ce mécanisme que les couches 1bis et 4 industrialisent.

37.2. Un choix d'ingénierie : pur Python plutôt que Z3

La contrainte de conception était explicite : pur Python, multi-plateforme, aucune dépendance binaire. Z3, le solveur SMT de Microsoft Research, serait le choix évident sur le plan des capacités — mais c'est un binaire C++, dont la distribution multi-OS contredit la promesse d'un serveur qui s'installe par un simple pip. Le système s'appuie donc sur des moteurs déjà présents dans VAILS et intégralement écrits en Python.

Le fragment propositionnel est confié à sympy.logic.satisfiable, une implémentation DPLL. L'arithmétique et l'algèbre passent par le CAS SymPy déjà exploité au chapitre 36 — simplify, solve, reduce_inequalities. Le fragment relationnel est traité par le moteur Prolog du chapitre 11. Et la vérification d'un terme de preuve revient à un noyau maison d'environ cent vingt lignes. Quatre moteurs, aucun binaire, et surtout : aucun d'eux n'a été écrit pour l'occasion.

Le compromis est assumé et il faut le nommer. Un SAT pur Python ne passe pas à l'échelle de millions de clauses. Il n'y a pas de SMT combiné, donc pas de raisonnement mêlant plusieurs théories. Le prouveur réel est univarié. Pour des fonctions de taille petite à moyenne — précisément ce qu'un LLM génère — cela suffit. Au-delà, le système ne bluffe pas : il répond unknown et bascule sur du property-test. C'est ce refus du faux positif qui fait la valeur de l'ensemble, et la section 37.9 montrera qu'il est possible d'en tirer une information exploitable plutôt qu'un simple aveu d'échec.

37.3. Couche 0 — la proposition comme donnée

Une proposition, dans VVL, n'est pas un type nouveau. C'est un VVL_Sym — l'expression symbolique du chapitre 36 — dont le contenu se trouve être booléen ou relationnel. Ce choix n'est pas une économie de code : il signifie qu'une proposition traverse gratuitement l'arithmétique native et le pont VVL ↔ SymPy. On peut la stocker dans une variable, la passer à une lambda, la construire par composition, la faire produire par un LLM.

Les connecteurs s'écrivent en préfixe — proposition:and, proposition:or, proposition:not, proposition:implies, proposition:iff — et les comparaisons de même. Mais écrire une formule de cette façon devient vite illisible, d'où un compilateur infixe qui accepte la notation usuelle. Le lexer VVL ne connaissant ni & ni ->, la formule passe par une chaîne :

```
$p = [proposition:symbolic p]
$q = [proposition:symbolic q]

[$$ "p & q -> r"]                          # => <symbolic: Implies(p & q, r)>
[log [proposition:valid "p | ~p"]]         # => 1   (tiers exclu)
[log [proposition:entails [$p [proposition:implies $p $q]] $q]]   # => 1  (modus ponens)
```

Le sucre [$$ "..."] est à la proposition ce que [$ ...] est au symbole : un alias de tête. La précédence des opérateurs va du moins au plus prioritaire : équivalence, implication (associative à droite), disjonction, conjonction, négation, comparaisons, arithmétique, application de fonction, parenthèses. Un identifiant nu devient un symbole propositionnel. Trois builtins de décision complètent la couche : proposition:sat rend la satisfiabilité et un modèle, proposition:valid teste la tautologie, proposition:entails vérifie qu'un ensemble de prémisses implique un but.

37.4. Couche 2 — la décision automatique et le routeur de fragment

[proof:auto P] décide si P est toujours vraie. C'est la porte d'entrée pratique du système : un seul appel, une chaîne infixe acceptée directement, et un verdict structuré en retour. Le statut vaut proved, refuted — accompagné d'un contre-exemple — ou unknown.

```
[log [proof:auto "(p & (p -> q)) -> q"]]   # proved   (by: sat,      justification: sat-proved)
[log [proof:auto "x^2 >= 0"]]              # proved   (by: cas-real, justification: cas-proved)
[log [proof:auto "x^2 >= 1"]]              # refuted  (contre-exemple {x: 0})
[log [proof:auto "x^2 + y^2 >= 0"]]        # unknown  (multivarié, hors prouveur réel)
```

Derrière l'appel unique se trouve un routeur de fragment, et c'est lui qui porte la soundness. Une proposition purement propositionnelle part au SAT, qui en donne une décision complète. Une proposition arithmétique ou relationnelle passe d'abord par le CAS, puis par une recherche de contre-exemple entier borné, puis par le prouveur réel — et si aucun de ces trois n'aboutit, la réponse est unknown.

Le détail qui mérite l'attention est le refus d'appliquer le SAT à $x > 0$. Un solveur propositionnel accepterait cette formule : il la traiterait comme un atome booléen opaque, et conclurait qu'elle est satisfiable — ce qui est vrai mais sans rapport avec la question posée. Le routeur reconnaît le fragment arithmétique et l'envoie au CAS. C'est exactement le genre de garde-fou qui distingue un système sound d'un système qui répond vite. La quatrième ligne de l'exemple ci-dessus est du même ordre : $x^2 + y^2 \geq 0$ est évidemment vraie, mais le prouveur réel étant univarié, le système préfère avouer son ignorance plutôt que de généraliser un cas univarié.

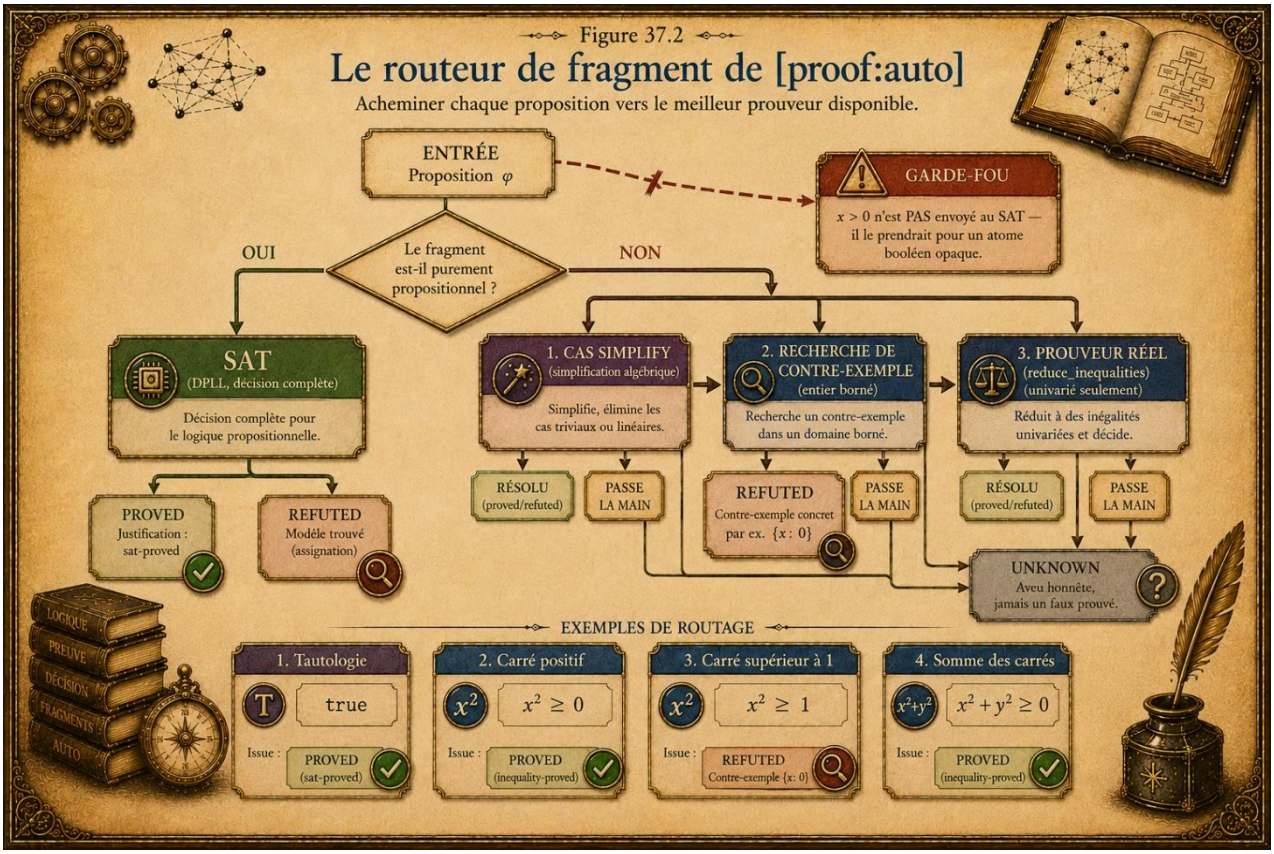


Figure 37.2 — Router avant de décider : c'est le routeur qui porte la soundness

37.5. Couche 1 — le noyau de confiance

[proof:auto] décide seul, mais son périmètre est celui de ses procédures de décision. Pour tout ce qui les dépasse, il faut pouvoir vérifier une preuve fournie de l'extérieur. C'est le rôle de [proof:check GOAL TERME], qui prend un terme de preuve — un arbre de règles d'inférence en déduction naturelle — et vérifie qu'il établit bien le but.

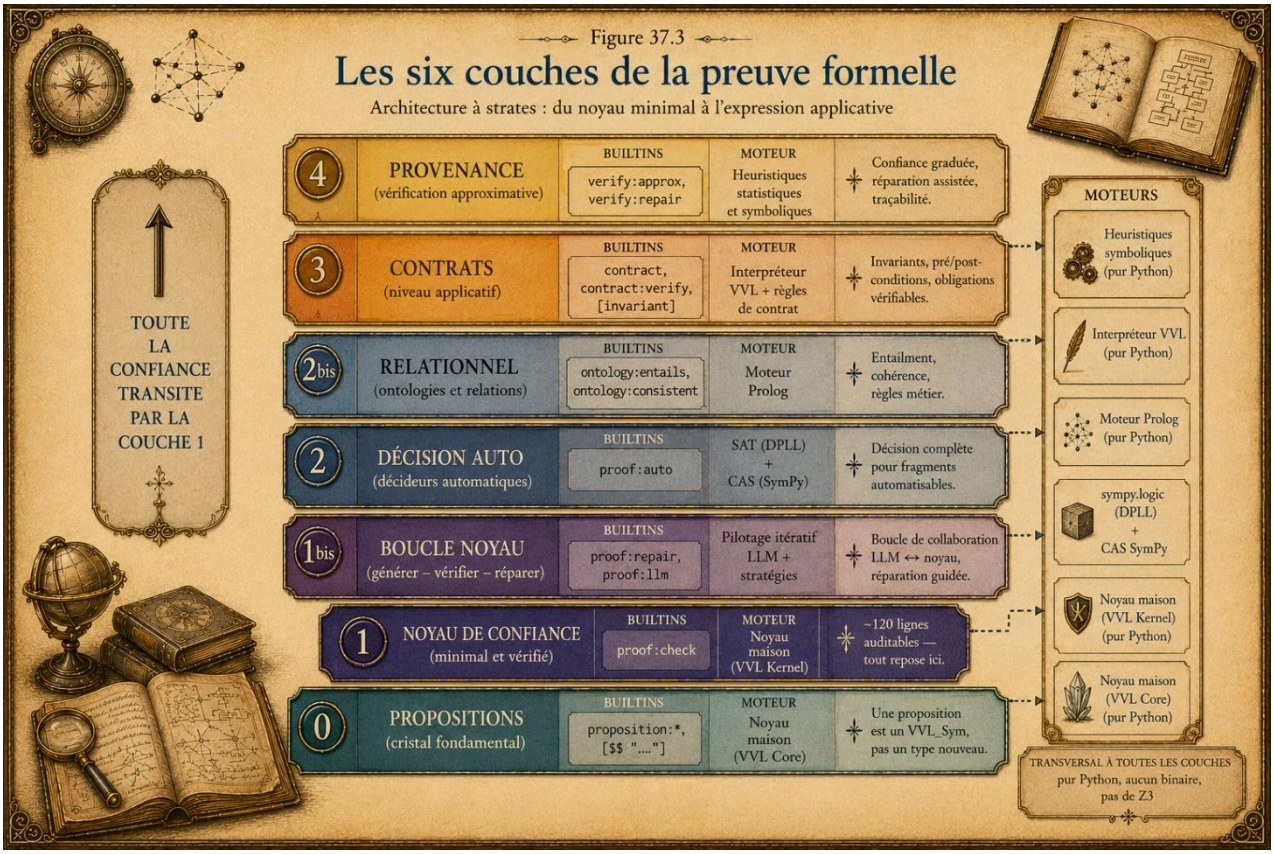


Figure 37.3 — L'architecture en six couches et les quatre moteurs qui les portent

Sept règles suffisent : la feuille automatique [auto P], déchargée par proof:auto ; l'hypothèse [assume H], qu'il faudra décharger ; le modus ponens [mp] ; l'introduction et l'élimination de la conjonction ; l'introduction de la disjonction ; et l'introduction de l'implication [imp-intro], qui décharge une hypothèse. Une preuve est valide si et seulement si sa conclusion est équivalente au but et qu'aucune hypothèse ne reste non déchargée.

```
[proof:check "(p & q) -> (q & p)"
  [imp-intro "p & q"
     [and-intro [and-elim "q" [assume "p & q"]]
                [and-elim "p" [assume "p & q"]]]]]
# => valid: true, justification: kernel-checked
```

Une subtilité d'implémentation mérite d'être signalée, parce qu'elle illustre le prix à payer pour s'appuyer sur une bibliothèque externe. SymPy réordonne les arguments des conjonctions et des disjonctions selon son propre ordre canonique. Les règles and-elim et or-intro ne peuvent donc pas désigner un conjoint par sa position : elles sont dirigées par cible, et l'on nomme explicitement le conjoint ou le disjoint visé. C'est le genre de détail qui n'apparaît que lorsqu'on écrit réellement le noyau.

Ce noyau est le point de confiance unique du système. Il tient en une centaine de lignes, il est relisible par un humain en une soirée, et tout ce qui porte la justification kernel-checked est passé par lui. C'est peu de code pour beaucoup de garantie — ce qui est précisément l'argument de de Bruijn.

37.6. Couche 1bis — la boucle noyau : le LLM propose, le noyau dispose

Les deux couches précédentes posent un vérificateur. Il reste à lui donner un fournisseur de preuves. [proof:repair] et [proof:llm] ferment la boucle : un proposer soumet un terme, le noyau le vérifie, et en cas de rejet la raison précise — antécédent qui ne colle pas, hypothèses non déchargées, conclusion différente du but — est réinjectée au tour suivant.

Le proposer de [proof:repair] est une lambda enfichable, ce qui rend la boucle testable sans appel réseau. L'exemple ci-dessous échoue volontairement au premier tour, en oubliant de décharger ses hypothèses, puis se corrige :

```
$n = 0
$proposer = [lambda [feedback]
    [do
        ($n = $n + 1)
        [if ($n == 1)
            '[mp [assume "p -> q"] [assume "p"]]'
            '[imp-intro "(p -> q) & p" [mp [and-elim "p -> q" [assume "(p -> q) & p"]]
                                           [and-elim "p" [assume "(p -> q) & p"]]]]'
        ]
    ]
]
[log [proof:repair "((p -> q) & p) -> q" $proposer 5]]
# => proved: true, iterations: 2, justification: kernel-checked
```

[proof:llm] remplace la lambda par le modèle : le prompt contient la grammaire des règles du noyau et le but, et l'erreur du noyau est réinjectée dans le prompt suivant. Le point essentiel est que le proposer ne porte aucune confiance. Que le terme vienne d'une lambda déterministe, d'un LLM ou d'un fichier, un succès est kernel-checked et un échec reste un échec honnête. À aucun moment le modèle ne certifie quoi que ce soit — il ne fait que proposer des candidats à un arbitre qui, lui, ne se trompe pas.

C'est la formulation la plus nette de la thèse neuro-symbolique de cet ouvrage. Le neuronal fournit la recherche dans un espace immense ; le symbolique fournit le critère d'acceptation. Ni l'un ni l'autre ne suffit : sans le LLM, il faudrait écrire les preuves à la main ; sans le noyau, il faudrait croire le LLM sur parole.

37.7. Couche 2bis — entailment relationnel et audit de cohérence

Le fragment relationnel ne relève ni du SAT ni du CAS mais du moteur Prolog. Deux builtins l'exposent. [ontology:entails $schema "but"] demande si la base de connaissances d'un schéma prouve un but donné. [ontology:consistent $schema] fait l'audit inverse : il cherche les contradictions.

Cet audit est intégralement symbolique — aucun appel LLM. Il détecte les attributs required manquants, les violations de bornes min et max, les valeurs hors énumération, les contraintes logiques posées par [assert] et [validate] qui sont bloquantes, celles posées par [check] qui sont de simples avertissements, et les dépassements de cardinalité maximale sur les relations.

```
$O = [ontology:define org [ontology:class Agent
        [attributes [nom String [required true]] [age Integer [min 18]]]]]
$a1 = [new Agent]
$a1.age = 15
[ontology:commit $a1]

[log [ontology:consistent $O]]   # consistent: false — required(nom) + range(age 15 < 18)
[log [ontology:entails $O "instance(X, agent)"]]   # => 1
```

Le lecteur attentif remarquera que le builtin s'appelle consistent et non consistent? : le point d'interrogation est un token du langage, il ne peut pas terminer un nom. Détail anecdotique, mais qui rappelle que ce système vit dans un langage réel avec ses contraintes lexicales.

La portée pratique de cette couche est considérable pour les pipelines des chapitres 15 et 16. Une ontologie peuplée par extraction LLM peut être auditée sans rappeler le modèle : les contradictions se lisent dans le schéma. C'est de la vérification à coût nul, sur des données dont la production, elle, a coûté des appels réseau.

37.8. Couche 3 — contrats de fonction et invariants de classe

Les couches précédentes raisonnent sur des propositions. La couche 3 les attache à du code. Un contrat associe à une fonction des préconditions [require] et des postconditions [ensure] :

```
$safe_div = [contract [$a $b]
    [require "b != 0"]
    [ensure  "result * b = a"]
    ($a / $b)]

[log [$safe_div 6 2]]            # => 3
[log [$safe_div 6 0]]            # ERREUR : précondition violée: Ne(b, 0)
[log [contract:verify $safe_div]]
# => status: proved, method: static-symbolic, result: a/b
```

Deux régimes de vérification cohabitent, et c'est le point intéressant. À l'exécution, les conditions sont évaluées et une violation lève. Statiquement, [contract:verify] exécute symboliquement le corps de la fonction et soumet l'implication requires ⟹ ensures à [proof:auto] — le contrat est alors prouvé une fois pour toutes, sans exécution. Quand le corps n'est pas symbolisable, le système ne renonce pas : il se replie sur du property-test, et la justification descend d'un cran sur l'échelle de la section suivante.

Une convention de nommage mérite d'être retenue car elle surprend au premier usage. Dans le corps de la fonction, un paramètre s'écrit $a — c'est une valeur. Dans les conditions, qui sont des propositions infixes, il s'écrit a — c'est un symbole. Le mot-clé result désigne la valeur de retour dans les postconditions.

Le pendant orienté objet du contrat est l'invariant de classe. La clause [invariant] contient une expression booléenne sur $self et les champs, vérifiée après construction et après chaque mutation. Une mutation qui violerait l'invariant est annulée :

```
[class Compte [field solde 0] [invariant ($solde >= 0)]
    [method retrait [m] ($self.solde = $self.solde - $m)]]

$c = [new Compte]
$c.solde = 50
[$c.retrait 100]     # ERREUR : [invariant] violé (état: solde = -50)
[log $c.solde]       # => 50   — rollback effectué
```

Le rollback est ce qui distingue l'invariant d'une simple assertion. Une assertion signale que l'objet est devenu incohérent ; l'invariant garantit qu'il ne l'est jamais devenu. Pour un système où des objets sont construits par extraction LLM — le pipeline du chapitre 15 — cette différence est structurante : un objet à demi peuplé par une extraction ratée n'entre pas dans la base.

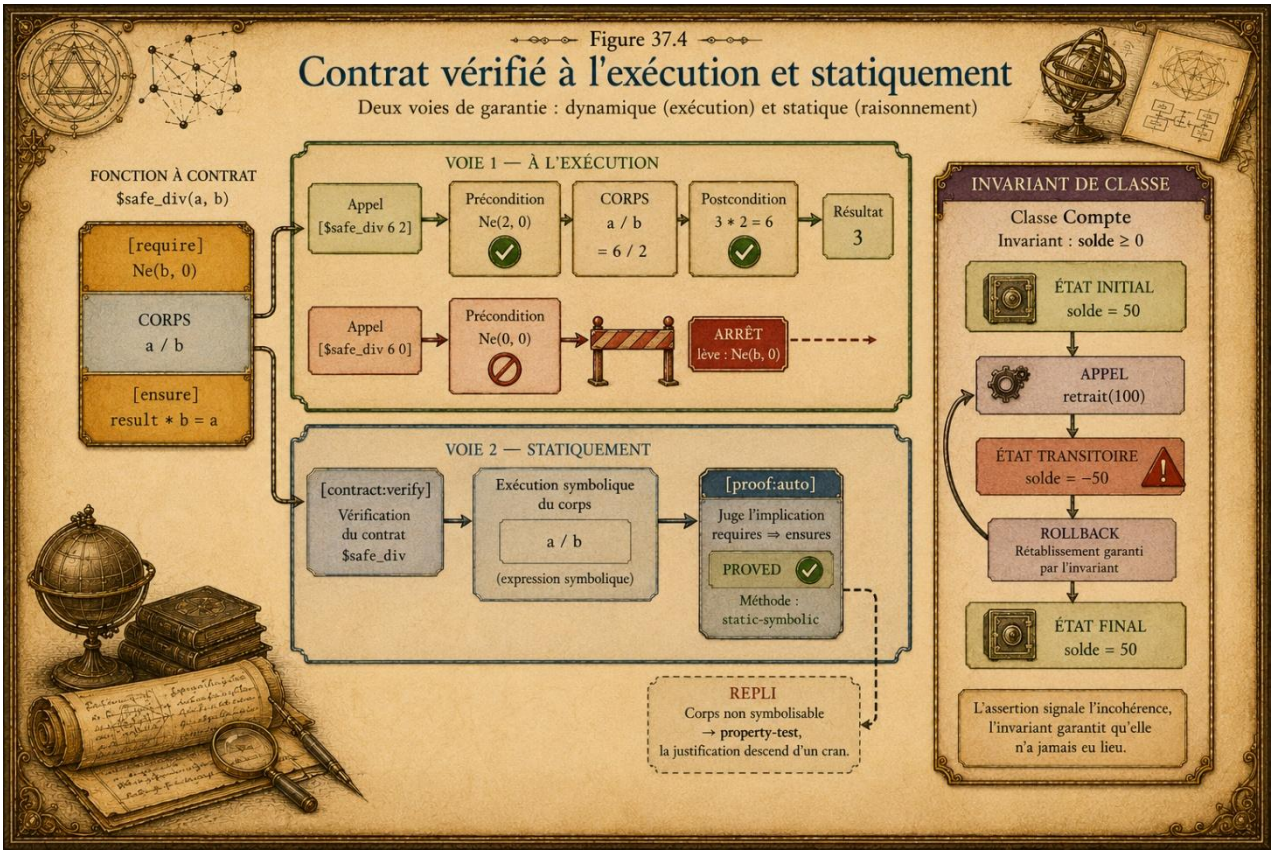


Figure 37.4 — Deux régimes de vérification pour un même contrat, et le rollback des invariants

37.9. Couche 4 — la provenance graduée : sortir du tout-ou-rien

Un prouveur classique rend un verdict binaire. Cette couche est le véritable différenciateur de l'approche VVL : chaque verdict porte une justification calibrée, ordonnée sur une échelle explicite.

kernel-checked > sat-proved > cas-proved > property-tested > unproved

À quoi s'ajoute refuted, qui n'est pas une faiblesse mais une réfutation certaine — savoir qu'une propriété est fausse vaut mieux que ne rien savoir. Le builtin [verify:approx] projette cette échelle sur le registre de confiance VVL_Approximation rencontré au chapitre 35 : une preuve machine devient une confiance exact, un résultat testé devient moderate, une propriété non prouvée devient low.

```
1  [log [verify:approx "x^2 >= 0"].confidence]        # => "exact"
2  [log [verify:approx "x^2 + y^2 >= 0"].confidence]  # => "low"
```

L'intérêt de ce raccordement est qu'il rend l'ignorance du prouveur exploitable en aval. Une propriété non prouvée ne bloque pas le pipeline : elle le traverse en portant une confiance faible, qui se propage dans l'arithmétique d'approximation et qui peut être filtrée par une policy du chapitre 23. On obtient un registre de confiance honnête plutôt qu'un mur — exactement ce dont a besoin un système où tout ne sera jamais prouvable.

Reste la boucle. [verify:repair] joue au niveau des propositions le rôle que [proof:repair] joue au niveau des termes : un proposer est rappelé avec le verdict raté jusqu'à produire une proposition prouvée.

```
$n = 0
$fix = [lambda [fb] [do ($n = $n + 1) [if ($n == 1) "x > 5" "x + 1 > x"]]]
[log [verify:repair $fix 5]]
# => repaired: true, iterations: 2, justification: cas-proved
#    history: [{1, "x > 5", refuted}, {2, "x + 1 > x", proved}]
```

Le proposer est ici déterministe pour que l'exemple soit reproductible, mais rien n'empêche d'y appeler [§§ ...] ou [resolve]. Le contre-exemple du tour raté est alors réinjecté comme feedback, et l'on obtient une auto-réparation pilotée par la vérification — le LLM corrige non pas au jugé, mais à partir d'un fait.

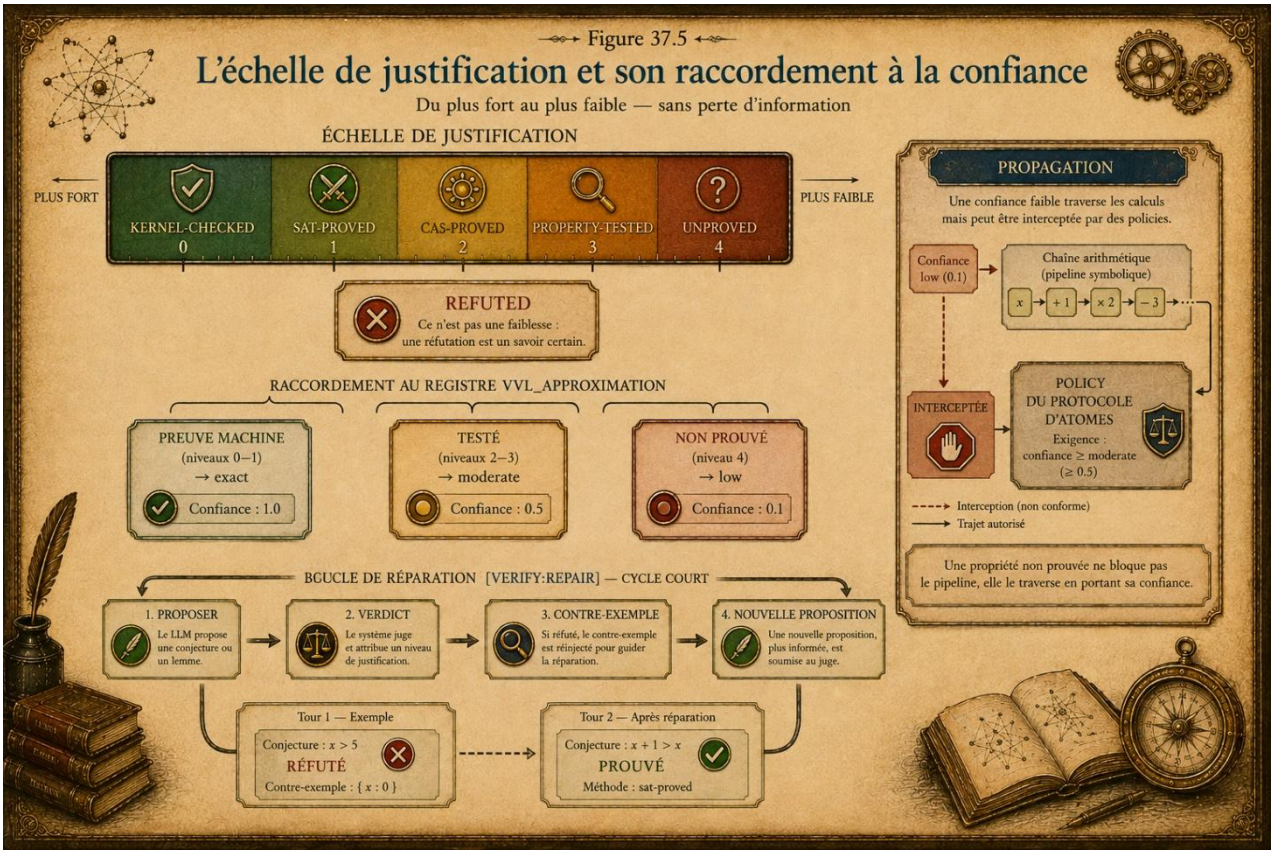


Figure 37.5 — Entre le prouvé et l'inconnu, une gradation exploitable en aval

37.10. Synthèse : la réponse au diagnostic du chapitre 1

Il faut mesurer le chemin parcouru depuis la section 1.5. Le reproche fait aux LLM y était qu'aucun mécanisme interne ne garantit qu'une réponse satisfait une contrainte donnée, et que pour s'en assurer il faut ajouter une couche externe. Ce chapitre décrit cette couche — à ceci près qu'elle n'est pas externe. Le vérificateur est dans le langage, il partage le runtime, le système de types et le registre de confiance des autres sous-systèmes.

Trois propriétés méritent d'être retenues. La première est la soundness : le système ne rend jamais un faux « prouvé ». Cette discipline coûte des unknown, et c'est le prix à payer pour qu'un proved veuille dire quelque chose. La deuxième est la petitesse du noyau : une centaine de lignes portent toute la confiance, le reste est faillible sans conséquence. La troisième est la gradation : entre le prouvé et

l'inconnu, le système ménage des degrés, ce qui lui permet de rester utile là où un prouveur binaire deviendrait bloquant.

Ce chapitre referme aussi une boucle interne à l'ouvrage. Le chapitre 17 fait induire des règles par le LLM, le chapitre 22 lui fait générer des programmes entiers, le chapitre 23 les rend introspectables et le chapitre 27 apprend des programmes logiques corrects par construction. Il manquait le juge. Avec la preuve formelle, la chaîne est complète : le modèle propose, le noyau tranche, la provenance enregistre le degré de certitude obtenu, et la policy décide de ce qui passe en production.

Le module fait partie du serveur courant. Ses limites sont connues et documentées — SAT non passant à l'échelle, prouveur réel univarié, pas de théories combinées — et elles définissent un périmètre plutôt qu'un provisoire. Dans ce périmètre, celui des fonctions et des ontologies de taille humaine, il tient sa promesse : distinguer, sans jamais mentir, ce que l'on sait de ce que l'on croit.

38. miniKanren embarqué : la logique relationnelle comme citoyen du langage

Mots-clés : miniKanren, microKanren, relation réversible, unification, entrelacement, complétude, cKanren, contrainte différée, absento, quasiquote, ProbLog, Active Record, eval relationnel, quine

Le chapitre 11 a doté VVL d'un moteur Prolog. On y déclare des faits, on écrit des règles, on interroge — et l'inférence remonte ce qui en découle. Ce chapitre introduit un second moteur logique, qui ne remplace pas le premier mais l'étend sur trois axes où Prolog montre ses limites : la complétude de la recherche, la réversibilité des relations, et la composition avec le reste du langage.

miniKanren est une famille de langages de programmation logique relationnelle née dans le monde Scheme, popularisée par The Reasoned Schemer de Friedman, Byrd et Kiselyov. Son implémentation minimale, microKanren, tient en une centaine de lignes — ce qui en fait un candidat naturel à l'embarquement. VVL en possède une écrite de zéro dans builtin/minikanren.py, sans aucune dépendance externe, développée de la version langage 7.0.0 à la 7.1.7.

L'idée tient en deux lignes. On décrit des relations — récursives, réversibles — on pose des buts, et [run-all …] résout : il énumère toutes les solutions par recherche complète à entrelacement. Le point qui change tout par rapport à un moteur logique classique est que le but n'est pas une construction d'un sous-langage : c'est une valeur VVL ordinaire, inerte et composable. Un but probabiliste, une contrainte symbolique, une requête Prolog et une requête sur fichier CSV se mélangent dans le même [run-all].

```
[predicate appendo [x y z]
    [any
        [all ($x == []) ($y == $z)]
        [exists [h t r]
            [all ($x == [cons $h $t]) ($z == [cons $h $r]) [appendo $t $y $r]]]]]

[run-all [a b] [appendo $a $b [1 2 3]]]
# => [[[] [1 2 3]] [[1] [2 3]] [[1 2] [3]] [[1 2 3] []]]   (toutes les découpes)
```

Ce chapitre suit la progression du plus simple au plus complexe : le résolveur et les premiers buts, l'unification et les termes, les combinateurs et la complétude, les relations, le contrôle et la négation, le store de contraintes, les contraintes symboliques, puis les trois ponts — KB Prolog, objets, fichiers CSV — la couche probabiliste, et enfin l'interpréteur relationnel qui synthétise des programmes.

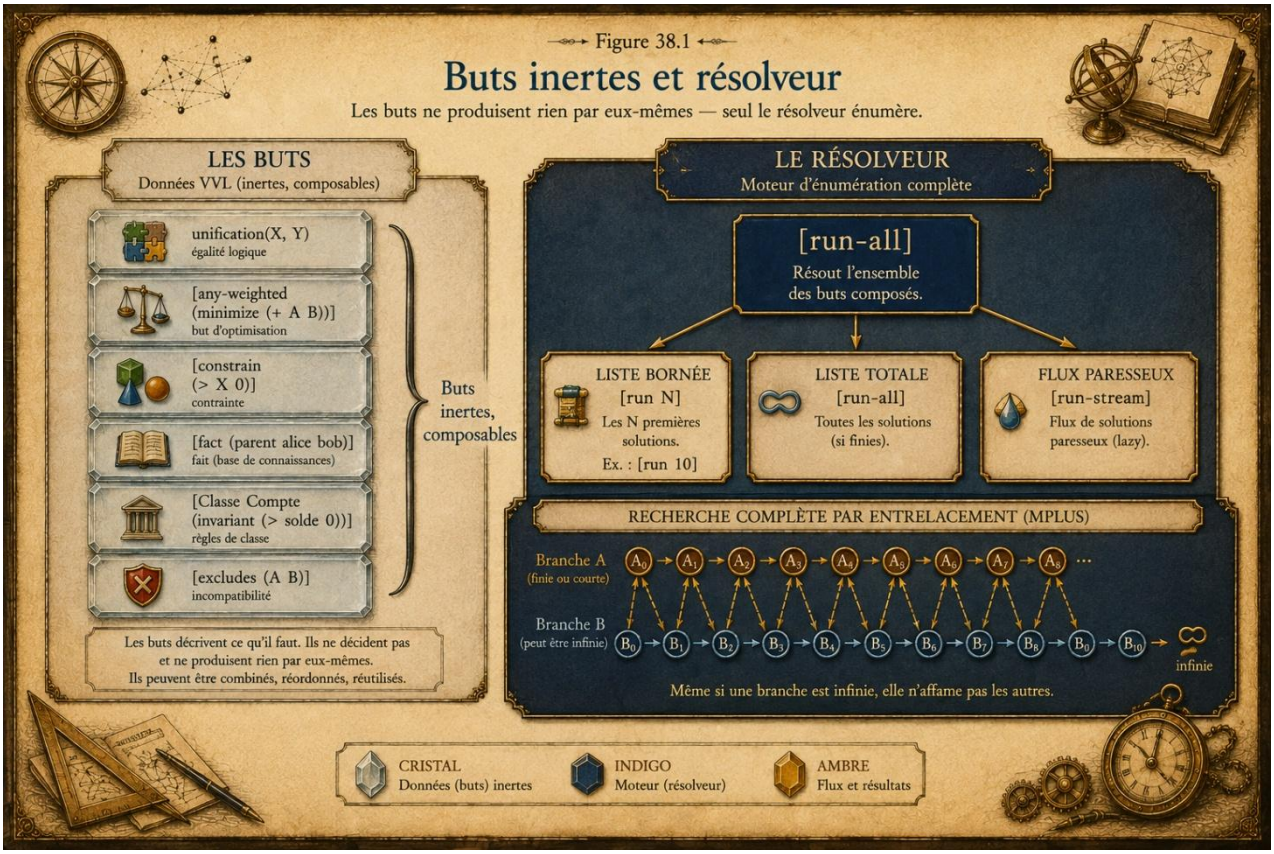


Figure 38.1 — Les buts sont des données ; seul le résolveur déclenche la recherche

38.1. Les buts sont inertes, le résolveur résout

C'est le point à comprendre avant tout le reste. Un but est une valeur VVL inerte. Il ne calcule rien tout seul. Écrire [unify $x 42] ne fait rien : c'est une description. Seul un résolveur déclenche la recherche et reifie — matérialise — les variables de requête.

Trois résolveurs coexistent, et leur choix dépend de la taille de l'espace de solutions. [run N [vars…] but…] rend une liste bornée aux N premières solutions : on veut un échantillon. [run-all [vars…] but…] rend la liste totale, entièrement matérialisée : l'espace est fini. [run-stream [vars…] but…] rend un flux paresseux : l'espace est infini, ou bien l'on souhaite composer avec les outils de flux du chapitre sur l'évaluation paresseuse — [take], [for], [map].

```
1  [run-all [x] [unify $x 42]]
2  # => [42]
```

La première liste, [x], déclare les variables de requête. Ce sont elles, et elles seules, que le résolveur reifie en sortie. Une seule variable déclarée donne une seule valeur par solution ; deux variables donnent des couples.

La disjonction [any g1 g2 …] réussit si l'une au moins de ses branches réussit, et le résolveur énumère toutes les branches satisfiables. La conjonction [all g1 g2 …] exige que tous les buts tiennent simultanément.

```
[run-all [x] [any [unify $x 1] [unify $x 2] [unify $x 3]]]
# => [1 2 3]

[run-all [x] [all [unify $x 1] [unify $x 2]]]
# => []      ($x ne peut valoir 1 ET 2)
```

Le second résultat mérite un commentaire. La liste vide n'est pas une erreur : c'est l'absence de solution, qui est une information. Une variable ne peut pas valoir à la fois 1 et 2 ; la conjonction échoue, et cet échec est un fait exploitable. Cette discipline — l'échec est un résultat, pas une exception — traverse tout le moteur.

Mises l'une dans l'autre, conjonction et disjonction expriment des requêtes riches. L'intersection de deux domaines s'écrit directement :

```
[run-all [x]
    [all [any ($x == 1) ($x == 2)]
         [any ($x == 2) ($x == 3)]]]
# => [2]
```

38.2. Unification, variables fraîches et termes

Le but le plus fondamental est l'unification. [unify a b], dont la forme infixe strictement équivalente en position de but est (a == b), rend a et b égaux. L'opération est bidirectionnelle et symétrique : peu importe quel côté porte la variable, le moteur la lie pour satisfaire l'égalité.

Il faut insister, car la confusion est fréquente : == n'est pas une affectation. Il ne dit pas « range b dans a ». Il pose la contrainte « a et b désignent la même chose », et le résolveur cherche à la satisfaire. C'est pourquoi l'ordre des opérandes est indifférent, et pourquoi deux unifications incompatibles sur la même variable n'ont aucune solution plutôt que d'écraser silencieusement la première.

Second point crucial : en position de but, $x est une variable logique du moteur miniKanren, isolée des variables VVL ordinaires de ctx.variables. Elle peut être non liée — libre — puis liée par unification. Elle ne fuit pas dans l'environnement VVL et n'est pas écrasée par une assignation VVL classique. Seul le résolveur lui donne une valeur, et seulement le temps d'énumérer les solutions. C'est cette étanchéité qui permet à [run-all [x] …] de reifier $x proprement, sans interférence avec le reste du programme.

[exists [a b …] g1 g2 …] introduit des variables logiques fraîches dont la portée est locale aux buts qui suivent. Sémantiquement, c'est le quantificateur existentiel : « il existe des a, b… telles que les buts sont vrais ». Ces variables auxiliaires ne figurent pas dans la liste de requête, donc ne sont pas reifiées en sortie — sauf si on les relie à une variable de requête. À l'intérieur d'un [exists], les buts sont en conjonction implicite.

```
[run-all [p]
    [exists [a b]
        [unify $a 1]
        [unify $b 2]
        [unify $p [cons $a $b]]]]
# => [[1 . 2]]
```

Le moteur manipule des termes : nombres, atomes, et structures construites. La brique de base est la paire [cons h t], faite d'une tête et d'une queue. La queue peut être n'importe quel terme, y compris

une variable logique encore non liée — et c'est de là que vient toute la réversibilité des relations sur listes. L'unification descend dans la structure : pour unifier deux cons, on unifie leurs têtes entre elles et leurs queues entre elles.

```
[run-all [x] [unify [cons 1 $x] [cons 1 2]]]
# => [2]
```

La liste littérale [1 2 3] est exactement une chaîne de cons terminée par nil, la liste vide [] — autrement dit [cons 1 [cons 2 [cons 3 []]]]. Les deux écritures s'unifient sans distinction. Comme la queue d'un cons peut rester variable, on peut décrire une liste dont on ne connaît pas encore la fin, ce qui est la base de la section suivante. L'unification pratique l'occurs-check, qui empêche de construire un terme cyclique en liant une variable à un terme qui la contient.

38.3. Conjonction, disjonction et complétude par entrelacement

Deux combinateurs forment le cœur de la logique relationnelle. [all g…] est le ET : l'intersection des solutions, les buts partageant leurs variables — ce qu'un but lie, le suivant doit le respecter. [any g…] est le OU : l'union des solutions. Mais c'est la manière dont [any] réalise cette union qui sépare miniKanren de Prolog, et le mot à retenir est entrelacement.

Un Prolog classique explore ses alternatives en profondeur d'abord. Si la première branche d'une disjonction décrit un espace infini, le moteur s'y enfonce et n'atteint jamais la seconde. miniKanren, lui, utilise mplus : au lieu de vider une branche avant de passer à la suivante, il alterne. Chaque branche reçoit équitablement la main, chaque branche progresse, et toute solution accessible finit donc par être trouvée. C'est la complétude de la recherche — une garantie que le backtracking en profondeur ne donne pas.

La démonstration se fait sur les entiers de Peano, dont l'énumération ne se termine jamais :

```
[predicate nat [n]
    [any
        [unify $n 0]
        [exists [m] [unify $n [s $m]] [nat $m]]]]

[take 5 [run-stream [n] [nat $n]]]
# => [0 [s 0] [s [s 0]] [s [s [s 0]]] [s [s [s [s 0]]]]]
```

Le test décisif de l'équité consiste à placer la relation infinie en première position d'un [any], avec une solution triviale en seconde. Un parcours en profondeur naïf s'enfoncerait dans nat pour toujours et n'atteindrait jamais done. Avec l'entrelacement, done finit par sortir :

```
[take 3 [run-stream [n] [any [nat $n] [unify $n done]]]]
# => contient "done"   malgré la branche infinie placée en tête
```

La conséquence pratique est qu'on peut raisonner sur des espaces infinis sans craindre la divergence, à condition de les consommer paresseusement par [run-stream] borné par [take]. C'est ce qui rend

praticables la synthèse de programmes de la section 38.12 et, plus généralement, toute génération relationnelle non bornée.

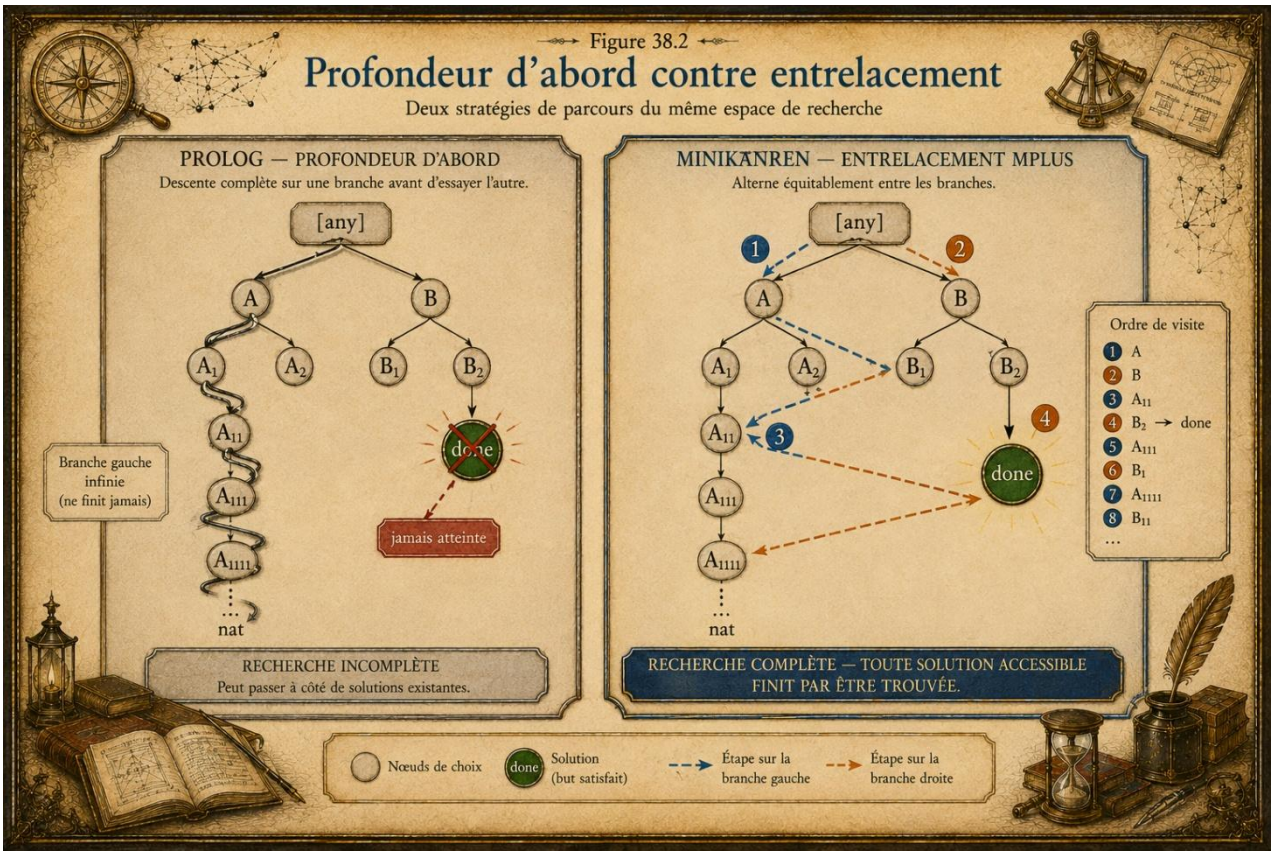


Figure 38.2 — L'entrelacement garantit qu'une branche infinie n'affame pas les autres

38.4. Relations récursives et réversibles

Une relation se définit avec [predicate nom [params] g...]. Elle nomme un ensemble de buts paramétrés. Contrairement à une fonction, une relation n'a ni entrées ni sorties : ses paramètres sont des termes logiques qui peuvent indifféremment être fournis ou laissés libres. C'est ce qui la rend réversible. Et comme elle peut s'appeler elle-même, elle est récursive.

[predicate] renvoie un objet de première classe décrivant la relation, introspectable :

```
$p = [predicate parento [a b] [unify $a $b]]
$p.dump     # => "<predicate: parento/2>"   (nom / arité)
$p.arity    # => 2
$p.name     # => "parento"
```

L'exemple canonique est appendo, présenté en ouverture de ce chapitre. La relation appendo x y z signifie « la concaténation de la liste x et de la liste y est la liste z ». Ses deux branches se lisent ainsi : cas de base, si x est la liste vide alors z est simplement y ; cas récursif, sinon x a une tête h et une queue t, z a la même tête h et une queue r, et r est la concaténation de t avec y.

La réversibilité se constate en interrogeant la même définition dans les deux directions. Dans le sens avant, on fournit x et y et l'on demande z — c'est la concaténation ordinaire. Dans le sens arrière, on

fournit le résultat z et l'on laisse x et y libres : la relation énumère alors toutes les façons de découper la liste.

```
[run 1 [z] [appendo [1 2] [3] $z]]
# => [[1 2 3]]                    (concaténation)

[run-all [a b] [appendo $a $b [1 2 3]]]
# => [[[] [1 2 3]] [[1] [2 3]] [[1 2] [3]] [[1 2 3] []]]
```

Quatre découpes, qu'aucune fonction append ordinaire ne sait produire. C'est tout l'intérêt d'une relation : on l'interroge dans la direction que l'on veut, sans écrire de code supplémentaire.

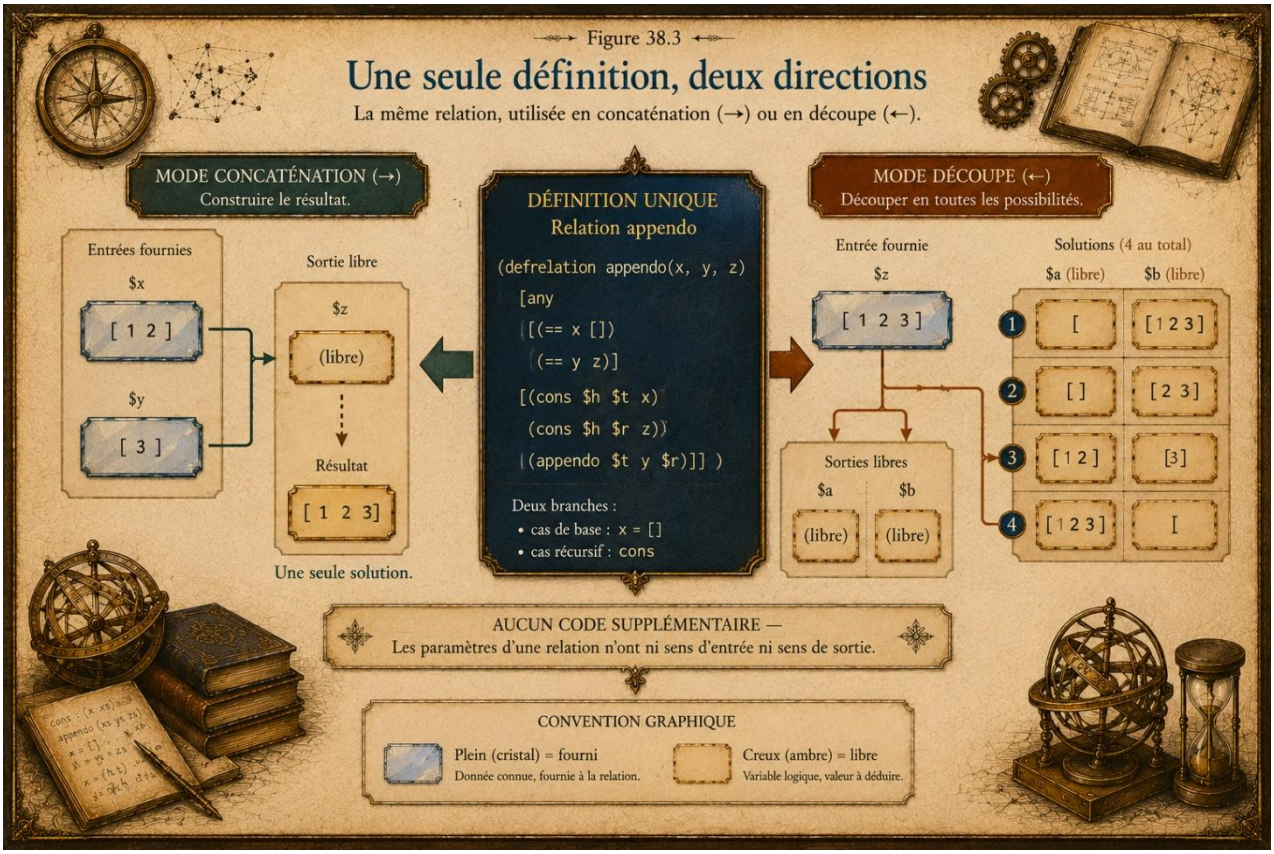


Figure 38.3 — La réversibilité : appendo concatène et découpe selon ce qu'on lui fournit

La seconde relation classique est membero, qui tient lorsque x est un élément de la liste l. Même structure : un cas de base où l'élément est en tête, un cas récursif qui cherche dans la queue. Interrogée avec l connue et x libre, elle énumère les éléments ; avec x connu, elle teste l'appartenance.

```
[predicate membero [x l]
    [exists [h t]
        [all ($l == [cons $h $t])
             [any ($x == $h)
                  [membero $x $t]]]]]
```

Un point mérite l'attention parce qu'il conditionne la terminaison : dans une conjonction, il faut contraindre avant de récurser. Le moteur traite les buts de gauche à droite ; placer l'appel récursif

avant les unifications qui le bornent peut faire explorer un espace infini inutilement. Dans appendo, l'unification ($z == [cons $h $r]) est posée avant l'appel [appendo $t $y $r]. Quand z est connue, cette unification fixe aussitôt h et la forme de r, ce qui réduit la liste à chaque appel récursif et garantit la terminaison. Si l'on récursait d'abord, rien ne bornerait la récursion. La règle pratique : mettre en tête les buts qui lient ou restreignent les variables, laisser les appels récursifs — les générateurs — en queue.

38.5. Choix engagés et négation : le cut sans l'impureté

En Prolog, le cut coupe le backtracking. C'est un outil indispensable en pratique, mais il est impur : son effet dépend de l'ordre de résolution, et il casse la réversibilité. miniKanren offre des équivalents purs — des buts qui restreignent l'énumération sans détruire la sémantique relationnelle. Ils sont disponibles depuis la version 7.0.0, la négation depuis la 7.0.15.

Deux buts constants servent de briques de base ou de marque-place. [succeed] réussit toujours sans rien lier — l'état passe inchangé. [fail] échoue toujours, produisant zéro solution.

```
[run-all [x] [all ($x == 1) [succeed]]]   # => [1]
[run-all [x] [all ($x == 1) [fail]]]      # => []
```

[once g] exécute g mais ne garde que sa première solution : l'énumération s'arrête là. C'est l'analogue pur du « je n'ai besoin que d'un témoin ». Sans [once], la disjonction ci-dessous donnerait [1 2].

```
[run-all [x] [once [any [unify $x 1] [unify $x 2]]]]
# => [1]
```

[commit [test g…] [test g…] …] est le choix engagé — le conda de la littérature, aussi appelé soft cut. Il essaie les clauses dans l'ordre, s'engage sur la première dont le test réussit, et ignore définitivement les suivantes, même si la suite de la clause engagée finit par échouer. Chaque clause est une liste dont le premier élément est le but-garde — le si — et le reste la suite exécutée si la garde a réussi — le alors.

```
[run-all [r]
    [exists [n]
        [all ($n == 75)
            [commit
                [($n >= 90) ($r == "excellent")]
                [($n >= 70) ($r == "bien")]
                [($n >= 50) ($r == "passable")]
                [[succeed]  ($r == "insuffisant")]]]]]
# => ["bien"]
```

La différence avec un simple [any] est nette : [any] énumérerait ["bien" "passable" "insuffisant"] — toutes les clauses dont la garde passe. [commit] n'en garde qu'une, la première, exactement comme un cut, mais sans impureté. C'est l'outil idiomatique de la classification mutuellement exclusive. Il conserve en revanche toutes les réponses du test engagé, celui-ci pouvant être non déterministe et lier des variables de plusieurs façons.

[commit-once], le condu, est identique mais ne garde qu'une seule réponse du test de la clause engagée, comme si chaque garde était enveloppée dans un [once]. Utile quand le test est non déterministe et qu'un seul témoin suffit.

```
[run-all [x]
    [commit-once
        [[any ($x == 1) ($x == 2)] [succeed]]]]
# => [1]      la garde réussirait pour 1 ET 2, un seul témoin est retenu
```

La négation s'écrit [absent g1 g2 ...], dont [without g...] est un alias strictement équivalent. Elle réussit si et seulement si la conjonction des buts n'a aucune solution : c'est la negation as failure, la sémantique du monde clos — ce qui n'est pas prouvable est faux. Deux propriétés en découlent. D'abord, [absent] est un test : il ne lie aucune variable, l'état passe inchangé en cas de succès. On l'utilise pour filtrer, jamais pour produire. Ensuite, la partie niée doit être suffisamment instanciée au moment où le but s'exécute ; on place donc [absent] après les buts qui lient ses variables.

```
[class Buyer [field name ""] [method init [n] $self.name = $n]]
[class Sale  [field ref ""] [field buyer ""]
    [method init [r b] $self.ref = $r $self.buyer = $b]]

$alice = [new Buyer "Alice"]
$carol = [new Buyer "Carol"]
[obj-assert $alice] [obj-assert $carol]
[obj-assert [new Sale "S-1" $alice]]      # seule Alice a acheté

# acheteurs SANS achat : on lie $name, puis on NIE l'existence d'une vente
[run-all [name]
    [all [Buyer [name $name]]
         [absent [Sale [buyer [Buyer [name $name]]]]]]]
# => ["Carol"]
```

La requête positive symétrique — les acheteurs avec achat — se passe de négation et rend ["Alice"]. L'ordre compte : [Buyer [name $name]] lie d'abord le nom, puis [absent ...] teste sur un nom instancié qu'aucune vente ne le référence.

38.6. Le store de contraintes : diségalité, ordre, types, domaines

Au-delà de l'unification, qui force l'égalité, le moteur porte un véritable store de contraintes façon cKanren. La diségalité est apparue en version langage 7.0.4, les comparaisons en 7.0.7, l'affichage des contraintes résiduelles en 7.0.18.

[distinct a b], de forme infixe (a != b), installe une contrainte : à aucun moment de la résolution a et b ne pourront devenir égaux. Ce n'est pas un test ponctuel mais une contrainte persistante, re-vérifiée après chaque liaison. Elle est donc correcte quel que soit l'ordre des buts.

```
[run-all [x] [all [distinct $x 1] ($x == 1)]]   # => []   contrainte posée AVANT
[run-all [x] [all ($x == 1) [distinct $x 1]]]   # => []   posée APRÈS : même verdict
[run-all [x] [all [distinct $x 1] ($x == 2)]]   # => [2]  valeur compatible

[run-all [a b]
    [all [any ($a == 1) ($a == 2)] ($b == 2) [distinct $a $b]]]
# => [[1 2]]      le couple [2 2] est éliminé par la contrainte
```

Les comparaisons d'ordre (a < b), (a > b), (a <= b) et (a >= b) sont un filtre numérique — le WHERE du SELECT relationnel. Depuis la 7.0.19 elles bénéficient de la propagation différée : si un opérande n'est pas encore lié à un nombre, la comparaison se suspend et se re-vérifie automatiquement à la liaison. L'ordre des buts n'importe donc plus. Une comparaison ne produit pas de valeurs, elle ne fait que les tester — mais elle attend patiemment.

```
[run-all [x] [all ($x > 5) ($x == 10)]]   # => [10]  comparaison AVANT la liaison
[run-all [x] [all ($x > 5) ($x == 3)]]    # => []    3 ne passe pas le filtre
```

Les contraintes de type — [a-number $x], [a-symbol $x], [a-string $x], alias des numbero, symbolo et stringo de cKanren — exigent qu'une variable soit d'un certain type. Comme les comparaisons, elles se suspendent tant que la variable est libre puis vérifient à la liaison. La contrainte de domaine fini [domain $x [v1 v2 ...]] contraint la variable à appartenir à un ensemble énuméré, avec le même comportement différé. Le mot-clé est domain et non in parce que in est déjà un opérateur infixe de VVL.

```
[run-all [x] [all [a-number $x] ($x == 5)]]        # => [5]
[run-all [x] [all [a-number $x] ($x == "hi")]]     # => []     viole le type
[run-all [x] [all [a-string $x] ($x == "hi")]]     # => ["hi"]
[run-all [x] [all [domain $x [1 2 3]] ($x == 2)]]  # => [2]
[run-all [x] [all [domain $x [1 2 3]] ($x == 5)]]  # => []     5 hors domaine
```

Reste une question que la plupart des moteurs éludent : que se passe-t-il si une variable de requête demeure libre mais contrainte à la fin de la résolution ? miniKanren ne triche pas. Il affiche la contrainte résiduelle, façon cKanren : la solution montre la variable reifiée sous la forme _.0, suivie de la liste des contraintes qui pèsent encore sur elle.

```
[run-all [x] [distinct $x 1]]
# => [[_.0 [[!= _.0 1]]]]

[run-all [x] [all [distinct $x 1] [distinct $x 2]]]
# => [[_.0 [[!= _.0 1] [!= _.0 2]]]]

[run-all [x] ($x > 5)]              # => [[_.0 [[> _.0 5]]]]
[run-all [x] [a-number $x]]         # => [[_.0 [[a-number _.0]]]]
[run-all [x] [domain $x [1 2 3]]]   # => [[_.0 [[domain _.0 [1 2 3]]]]]
```

À l'inverse, si la variable finit liée à une valeur concrète, le store est résolu : il n'y a plus de contrainte résiduelle et la sortie est la valeur nue. C'est la marque d'un vrai moteur cKanren — une variable libre mais contrainte n'est pas « sans réponse » : sa réponse est l'ensemble de ses contraintes. On notera que [constrain], objet de la section suivante, fait exception à cette règle : il est eager, résolu au premier passage du but.

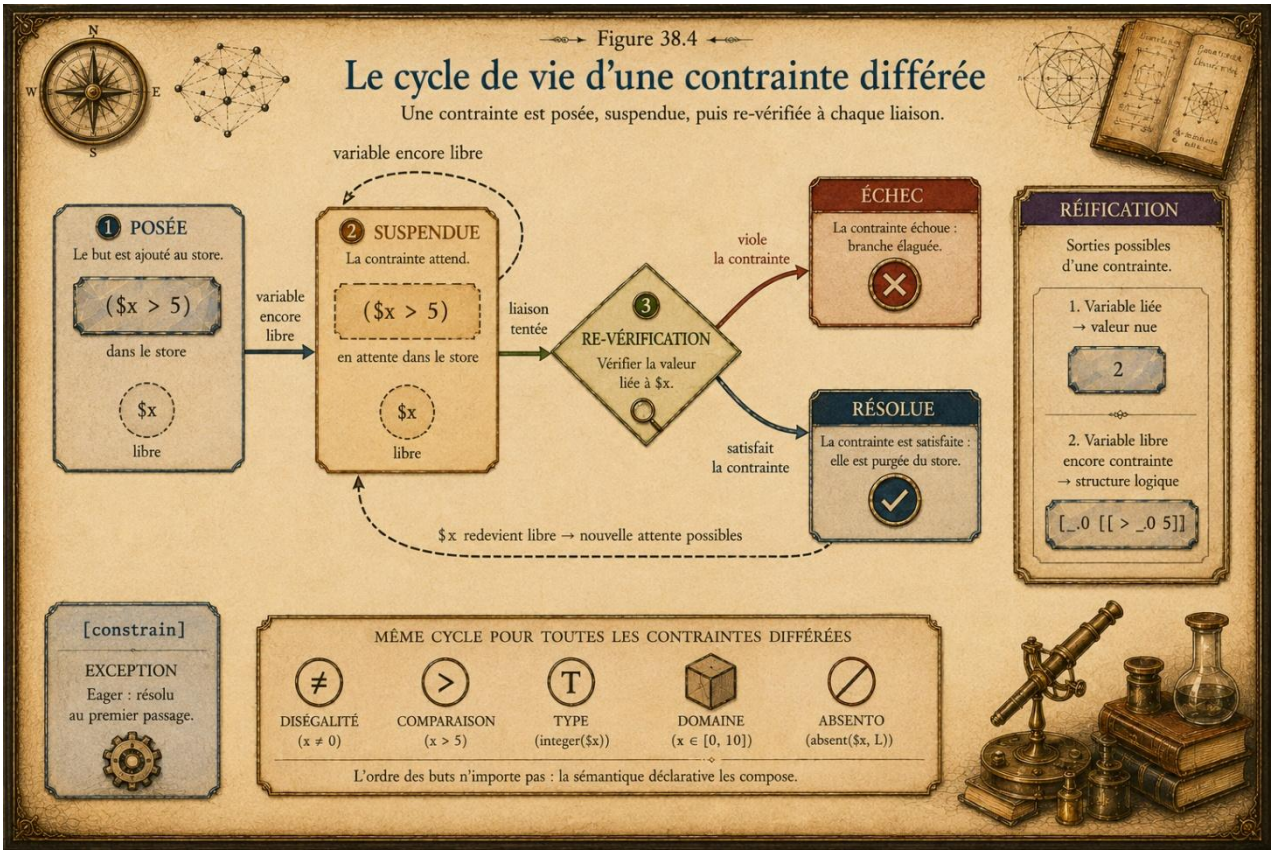


Figure 38.4 — Propagation différée : une contrainte suspend, attend, puis tranche à la liaison

38.7. Contraintes symboliques : la CLP par SymPy

[constrain "expr"] est un but qui pose la contrainte expr == 0 et la résout avec SymPy. C'est de la programmation logique par contraintes : on ne calcule pas une valeur, on énumère les valeurs qui annulent l'expression. Le chapitre 36 a introduit le calcul symbolique exact ; cette section en fait un citoyen de la logique relationnelle.

La règle d'or tient en une phrase : les symboles libres de l'expression correspondent aux variables logiques homonymes. Le symbole x dans la chaîne "x*x - 4" désigne la variable logique $x de la requête. Le pont se fait par le nom.

Quand la variable homonyme est libre, [constrain] résout l'équation et énumère ses racines réelles en unifiant la variable avec chacune. Une racine égale une solution. Et parce que chaque racine est posée par unification et non par affectation, [constrain] se compose naturellement avec tous les autres buts.

```
[run-all [x] [constrain "x*x - 4"]]
# => [-2 2]                                  les deux racines réelles

[run-all [x] [all [constrain "x*x - 4"] [distinct $x 2]]]
# => [-2]                                    on écarte une racine par diségalité

[run-all [x] [all [constrain "x*x - 4"] ($x == 2)]]
# => [2]                                     on vérifie une racine précise
```

Si un symbole homonyme est déjà lié à un nombre, [constrain] le substitue dans l'expression avant de résoudre pour le reste. L'ordre des buts n'a pas d'importance logique.

```
1  [run-all [x] [exists [y] ($y == 3) [constrain "x + y - 5"]]]
2  # => [2]                                  x + 3 - 5 = 0
3
4  [run-all [x] [all [constrain "x*x - 9"] [distinct $x -3]]]
5  # => [3]                                  système : racines ±3, puis filtrage
```

Quand une racine n'est pas rationnelle, le moteur ne l'approxime pas en flottant : il la conserve sous forme exacte, affichée <symbolic: ...>. L'exactitude algébrique est préservée de bout en bout, et ces valeurs restent de vrais termes réinjectables dans d'autres contraintes ou calculs symboliques.

```
1  [run-all [x] [constrain "x*x - 2"]]
2  # => [<symbolic: -sqrt(2)> <symbolic: sqrt(2)>]
```

Le comportement de [constrain] est polymorphe selon l'état de la variable. Liée, il vérifie : il substitue tout, teste si l'expression vaut zéro, et réussit ou échoue sans rien lier. Libre, il résout : il énumère les racines réelles en unifiant. Entre les deux — certains symboles liés, d'autres libres — la partie liée est substituée et l'on résout pour le reste. C'est le principe du is et du =:= de Prolog, poussé au rang de contrainte relationnelle : entièrement libre, c'est un générateur de solutions ; entièrement lié, c'est un test.

38.8. Le pont vers le KB Prolog

[fact "query"] est un but qui interroge le moteur Prolog du chapitre 11. Toute la base de faits et de règles devient directement requêtable depuis miniKanren, et les réponses se composent avec les autres buts. La forme [fact "kb" "query"] cible un KB nommé ; sans nom, c'est le KB "global" par défaut.

La convention de nommage est la clé du pont. Dans la requête, on écrit les variables en majuscule comme l'exige Prolog. Pour chaque solution, la variable Prolog Foo lie la variable logique homonyme $foo, en minuscule. Une variable Prolog sans homonyme déclaré est simplement ignorée : elle sert de variable de travail interne.

```
1  [run-all [p] [exists [i] [fact "prop(I, price, P)"]]]
2  # => [10 20 5]                            tous les prix du KB
```

Le KB se peuple de deux manières. Depuis l'objet, via [obj-assert ...] ou [$obj.assert] : chaque objet asserté produit un petit schéma de faits — instance(Id, Classe) déclare que l'objet est une instance de la classe, prop(Id, champ, Valeur) porte chaque champ scalaire, et rel(Id, champ, AutreId) porte les références vers d'autres objets. Ou directement en Prolog, via [facts "kb" "fait." ...] pour les faits bruts et [rules "kb" "tête :- corps." ...] pour les clauses.

Le mécanisme central des bases relationnelles s'obtient gratuitement : deux buts [fact] qui partagent une même variable logique sont joints sur cette variable. Ci-dessous, $i lie le nom et le prix du même objet — c'est une jointure sur la clé, exactement comme un JOIN SQL sur l'identité.

```
[run-all [name price]
    [exists [i]
        [all [fact "prop(I, name, NAME)"]
             [fact "prop(I, price, PRICE)"]]]]
# => [["Apple" 10] ["Banana" 5] ["Cherry" 20]]

# composition avec une diségalité
[run-all [p] [exists [i] [all [fact "prop(I, price, P)"] ($p != 5)]]]
# => [10 20]
```

Le point le plus important de cette section est que [fact] n'interroge pas seulement des faits bruts : l'inférence Prolog est mise à profit. Règles d'héritage, de transitivité, de subsomption — tout ce que le moteur sait déduire devient requêtable depuis miniKanren.

```
[facts "global"
    "parent(tom, bob)."
    "parent(bob, ann)."]
[rules "global"
    "grandparent(X, Z) :- parent(X, Y), parent(Y, Z)."]

[run-all [child] [fact "parent(tom, CHILD)"]]
# => ["bob"]
[run-all [gc] [fact "grandparent(tom, GC)"]]
# => ["ann"]        déduit par la règle, pas asserté tel quel
```

On a donc deux moteurs logiques qui coopèrent au lieu de se concurrencer : Prolog apporte son inférence en chaînage arrière et sa base persistante, miniKanren apporte la complétude, les contraintes et la composition. Chacun fait ce qu'il fait le mieux.

38.9. Le pont OO ↔ KB : la classe comme langage de requête

La section précédente supposait de connaître le schéma interne — les prédicats instance/2 et prop/3, les identifiants d'objets. Cette section supprime cette cérémonie : on écrit les objets via une méthode d'instance, on les lit en utilisant la classe comme une relation, et aucune connaissance du schéma Prolog n'est requise. Ces capacités couvrent les versions langage 7.0.9 à 7.0.14.

Côté écriture, [$obj.assert] persiste un objet dans le KB et renvoie le nombre de faits écrits — un instance/2 plus un prop/3 par champ. [$obj.assert "atelier"] cible un KB nommé, [$obj.retract] retire l'objet. La forme fonctionnelle [obj-assert $obj] reste disponible et équivalente ; la méthode d'instance est simplement la symétrie OO de la lecture par la classe.

Côté lecture, en position de but, [Classe [champ $v|valeur]…] est résolu comme une requête objet : la classe se comporte comme une relation unaire sur ses instances. Une variable en valeur de champ est une sortie — elle est liée et reifiée. Une valeur littérale est un filtre par égalité. Si une [predicate] porte le même nom que la classe, elle garde la priorité.

```
[class Gadget [field name ""] [field qty 0]
    [method init [n q] $self.name = $n $self.qty = $q]]

$g1 = [new Gadget "Hammer" 3]
$g2 = [new Gadget "Nail" 100]
[$g1.assert]        # => 3   nombre de faits écrits
[$g2.assert]

[run-all [name qty] [Gadget [name $name] [qty $qty]]]
# => [["Hammer" 3] ["Nail" 100]]
[run-all [name] [Gadget [name $name] [qty 3]]]
# => ["Hammer"]      filtre par égalité
```

Les champs liés se combinent avec les comparaisons d'ordre et la diségalité — c'est le SELECT … WHERE relationnel, la classe projetant et la comparaison filtrant. Pour lire dans un KB nommé, on passe son nom en chaîne, en tête ou en queue des specs de champ : le compilateur sépare les arguments, une liste étant une spec de champ et une chaîne le nom du KB.

```
[run-all [name]
    [exists [qty] [all [Gadget [name $name] [qty $qty]] ($qty > 5)]]]
# => ["Nail"]

[run-all [name] [Tool "atelier" [name $name]]]   # nom du KB en TÊTE
[run-all [name] [Tool [name $name] "atelier"]]   # ou en QUEUE
```

Il n'existe pas de syntaxe d'agrégat dédiée, et c'est délibéré : [run-all] produit la liste des valeurs, un builtin général la replie. sum, avg, count, min, max et length s'appliquent au résultat. La requête et l'agrégation restent deux briques séparées qui se composent.

```
[count [run-all [p] [Widget [price $p]]]]   # => 3
[sum   [run-all [p] [Widget [price $p]]]]   # => 35
[max   [run-all [p] [Widget [price $p]]]]   # => 20

# agrégat sur sous-population filtrée
[sum [run-all [p]
        [exists [n] [all [Widget [name $n] [price $p]] ($p > 5)]]]]
# => 30
```

Quand un champ contient une autre instance, [obj-assert] a créé un lien rel/3. On le suit dans la requête en imbriquant un patron de classe comme valeur de champ. La jointure est portée par la variable d'objet partagée, et l'imbrication est récursive à profondeur arbitraire.

```
$alice = [new Client "Alice" "Paris"]
$bob   = [new Client "Bob" "Lyon"]
[obj-assert [new Order "O-1" 100 $alice]]
[obj-assert [new Order "O-2" 50  $bob]]
[obj-assert [new Order "O-3" 200 $alice]]

[run-all [ref cname] [Order [ref $ref] [client [Client [name $cname]]]]]
# => [["O-1" "Alice"] ["O-2" "Bob"] ["O-3" "Alice"]]

# graphe + condition : client parisien, total > 80
[run-all [ref]
    [exists [t]
        [all [Order [ref $ref] [total $t] [client [Client [city "Paris"]]]]
             ($t > 80)]]]
# => ["O-1" "O-3"]
```

Enfin, interroger une super-classe remonte aussi les instances de ses sous-classes — le principe de substitution de Liskov appliqué à la requête. Le moteur énumère en disjonction sur la classe et ses sous-classes déclarées, et l'agrégation hérite du même comportement.

```
[class Item  [field name ""] [method init [n] $self.name = $n]]
[class Fruit : Item [field name ""] [method init [n] $self.name = $n]]
[class Veg   : Item [field name ""] [method init [n] $self.name = $n]]
[obj-assert [new Item "Generic"]]
[obj-assert [new Fruit "Apple"]]
[obj-assert [new Veg "Carrot"]]

[run-all [n] [Item  [name $n]]]   # => ["Apple" "Carrot" "Generic"]
[run-all [n] [Fruit [name $n]]]   # => ["Apple"]
```

La symétrie est totale : on écrit via l'instance, on lit via la classe, dans le même KB par défaut ou nommé. La lecture est pure — elle n'ajoute rien au KB.

38.10. Logique relationnelle probabiliste, façon ProbLog

Jusqu'ici une solution était soit présente, soit absente. Cette couche, apparue en version langage 7.0.17, ajoute une masse de probabilité : chaque solution porte un poids, et le moteur énumère tous les mondes possibles avec leur probabilité, de manière déterministe et sans aléa. C'est l'esprit de ProbLog — un programme logique augmenté de choix pondérés.

[any-weighted [poids but…] [poids but…] …] est une disjonction où chaque branche multiplie le poids de la solution. L'état porte un poids initialement égal à 1.0 ; prendre une branche de poids p multiplie ce poids par p. Le poids final d'un monde est donc le produit des poids des branches traversées. [run-weighted [vars] g…] résout comme [run-all] mais renvoie une liste de couples solution-probabilité.

```
[run-weighted [d] [any-weighted [0.7 ($d == "flu")] [0.3 ($d == "covid")]]]
# => [["flu" 0.7] ["covid" 0.3]]
```

Deux choix pondérés indépendants dans une même conjonction engendrent le produit cartésien des mondes, chaque masse étant le produit des poids. La masse totale vaut alors 1.0 — c'est une distribution propre.

```
[run-weighted [d f]
    [all [any-weighted [0.7 ($d == "flu")] [0.3 ($d == "covid")]]
         [any-weighted [0.9 ($f == "yes")] [0.1 ($f == "no")]]]]
# => 4 mondes :
#    flu   & yes : 0.7 * 0.9 = 0.63
#    flu   & no  : 0.7 * 0.1 = 0.07
#    covid & yes : 0.3 * 0.9 = 0.27
#    covid & no  : 0.3 * 0.1 = 0.03
```

[prob [vars] g...] renvoie la somme des poids des solutions satisfaisant les buts : c'est la probabilité marginale de la requête. On y ajoute simplement les contraintes qui définissent l'événement.

```
# P(d = flu)
[prob [d] [all [any-weighted [0.7 ($d == "flu")] [0.3 ($d == "covid")]]
               ($d == "flu")]]
# => 0.7

# P(f = yes), marginalisée sur d
[prob [d f]
    [all [any-weighted [0.7 ($d == "flu")] [0.3 ($d == "covid")]]
         [any-weighted [0.9 ($f == "yes")] [0.1 ($f == "no")]]
         ($f == "yes")]]
# => 0.9

# P(d = flu ET f = yes)
[prob [d f]
    [all [any-weighted [0.7 ($d == "flu")] [0.3 ($d == "covid")]]
         [any-weighted [0.9 ($f == "yes")] [0.1 ($f == "no")]]
         ($d == "flu") ($f == "yes")]]
# => 0.63
```

Comme tout but VVL, [any-weighted] se mélange à la requête objet. Mais il faut ici une vigilance particulière, car le moteur ne normalise pas. Avec un seul objet ground, le but objet réussit une fois avec une masse de 1 et la probabilité se réduit au poids. Si en revanche la variable de requête énumère plusieurs objets, [prob] somme le poids sur chaque solution : on obtient une valeur espérée — un nombre attendu d'objets — et non une probabilité bornée à 1.

```
# sans variable énumérante : une probabilité
[prob [] [all [Item [name "Widget"]]
              [any-weighted [0.8 [succeed]] [0.2 [fail]]]]]
# => 0.8

# avec variable énumérante : une espérance
[prob [name] [all [Item [name $name]]
                  [any-weighted [0.8 [succeed]] [0.2 [fail]]]]]
# => 0.8 * (nombre d'Items)
```

Depuis la 7.0.19, un poids de branche peut être un nombre ou une croyance Beta, écrite [beta α β]. Le poids utilisé est alors la moyenne de la croyance, α/(α+β), le moteur réutilisant la BetaBelief du module

bayésien du chapitre 35. C'est le pont avec la couche probabiliste : une croyance apprise par échantillonnage de Gibbs — α succès, β échecs — devient directement le poids d'une branche logique.

```
# Beta(7,3) -> moyenne 0.7   ;   Beta(3,7) -> 0.3
[run-weighted [d]
    [any-weighted [[beta 7 3] ($d == "flu")]
                  [[beta 3 7] ($d == "covid")]]]
# => [["flu" 0.7] ["covid" 0.3]]
```

Trois propriétés résument la couche : l'énumération est déterministe et exhaustive — aucun échantillonnage, le Monte-Carlo serait une couche ultérieure ; le poids d'un monde est le produit des poids pris ; et [prob] somme, donc rend une probabilité si la requête est ground et une espérance si elle énumère une population.

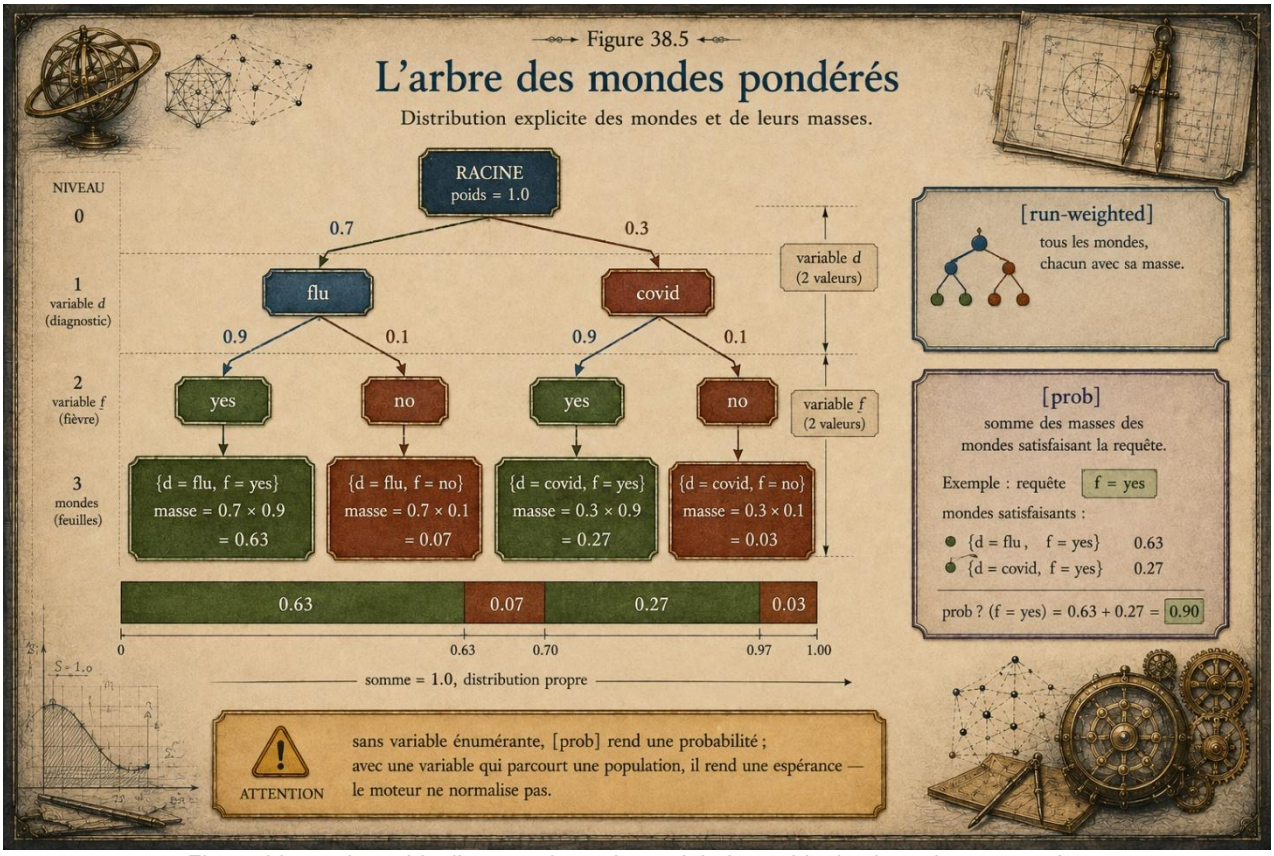


Figure 38.5 — Le poids d'un monde est le produit des poids des branches traversées

38.11. Le pont OO ↔ CSV : Active Record et requête sur table

Les faits venaient jusqu'ici du KB Prolog. Mais pour de grandes quantités de données tabulaires, une base en mémoire n'est pas le bon support. VVL dispose déjà d'un module CSV à stockage binaire .csvbin et index inversés : on adosse une classe à un fichier CSV et on le rend interrogeable en miniKanren, sans aucun autre moteur, en réutilisant ce stockage tel quel. Les versions concernées vont de 7.1.0 pour [bind-csv] à 7.1.4 pour la requête.

Le modèle est un Active Record dont la table est un CSV : la classe est dérivée du schéma du fichier, les instances sont des lignes vivantes, et [Classe …] interroge le CSV avec la même syntaxe que le pont KB.

```
$people = [CSV "people.csv"]      # charge le .csvbin binaire si présent
[bind-csv Person $people]          # classe générée : champs = colonnes du CSV
```

[bind-csv] lit le schéma — colonnes et types inférés — et génère la classe. Le choix de la clé primaire suit un ordre de préférence : argument explicite, puis pk déjà déclarée par le CSV, puis une colonne nommée id selon la convention des storages tabulaires, puis à défaut la première colonne. Cette clé sert d'identité d'instance pour la mise à jour et la suppression.

La requête distingue trois formes. Une valeur littérale dans le motif est un filtre d'égalité, résolu en temps constant par index inversé si [$csv.createIndex "colonne"] a été appelé, par scan sinon. Une variable est une projection. Et une comparaison posée avant le but de classe bénéficie du predicate pushdown : suspendue par la propagation différée de la section 38.6, elle est poussée dans le scan qui ne renvoie que les lignes utiles.

```
[run-all [n] [Person [city "Paris"] [name $n]]]
# => ["Alice" "Carol"]

# pushdown : comparaison posée AVANT le but de classe
[run-all [n a] [all ($a > 30) [Person [city "Paris"] [name $n] [age $a]]]]
# => [["Carol" 35]]

[sum [run-all [a] [Person [age $a]]]]          # => 90
[take 1 [run-stream [n] [Person [name $n]]]]   # => ["Alice"]
```

Les deux ordres sont corrects : après le but, le résultat est le même mais filtré ligne par ligne, sans l'optimisation. Agrégation et paresse composent naturellement, puisqu'il s'agit d'un but ordinaire.

Côté écriture, les instances persistent dans le CSV. La méthode .save est un upsert, .delete retire la ligne par sa clé primaire. La clé primaire est auto-assignée à .save si on ne l'a pas fournie — l'init généré exclut la pk, exactement comme [$obj.assert] génère un identifiant. La génération est inférée des valeurs existantes de la colonne : toutes numériques donne un auto-incrément par max+1, sinon un uuid4.

```
# CREATE sans id -> pk auto-assignée à .save
$d = [new Person "Dave" 40 "Nice"]
[$d.save]
$d.id                                  # => 4   (max+1)

# UPDATE : poser la pk explicitement -> upsert ciblé
$u = [new Person "Alice" 99 "Paris"]   $u.id = 1   [$u.save]

# DELETE par clé primaire
$x = [new Person "" 0 ""]   $x.id = 2   [$x.delete]
```

Les CSV étant des tables plates, une jointure se fait avec deux buts de classe qui partagent une variable logique, introduite par [exists] — l'idiome miniKanren déjà rencontré au pont KB.

```
# CA total des commandes d'Alice : Order.client_id == Person.id
[sum [run-all [t]
    [exists [cid]
        [all [Person [name "Alice"] [id $cid]]
             [Order [client_id $cid] [total $t]]]]]]
# => 300
```

Deux limites méritent d'être connues. Une variable de jointure doit impérativement être déclarée par [exists] ou être une variable de [run-all] : une variable non déclarée est traitée comme une valeur vide et le résultat est vide. Et le motif imbriqué [champ [Classe ...]], la jointure de graphe du pont KB, n'est pas supporté sur CSV — il faut passer par la variable partagée.

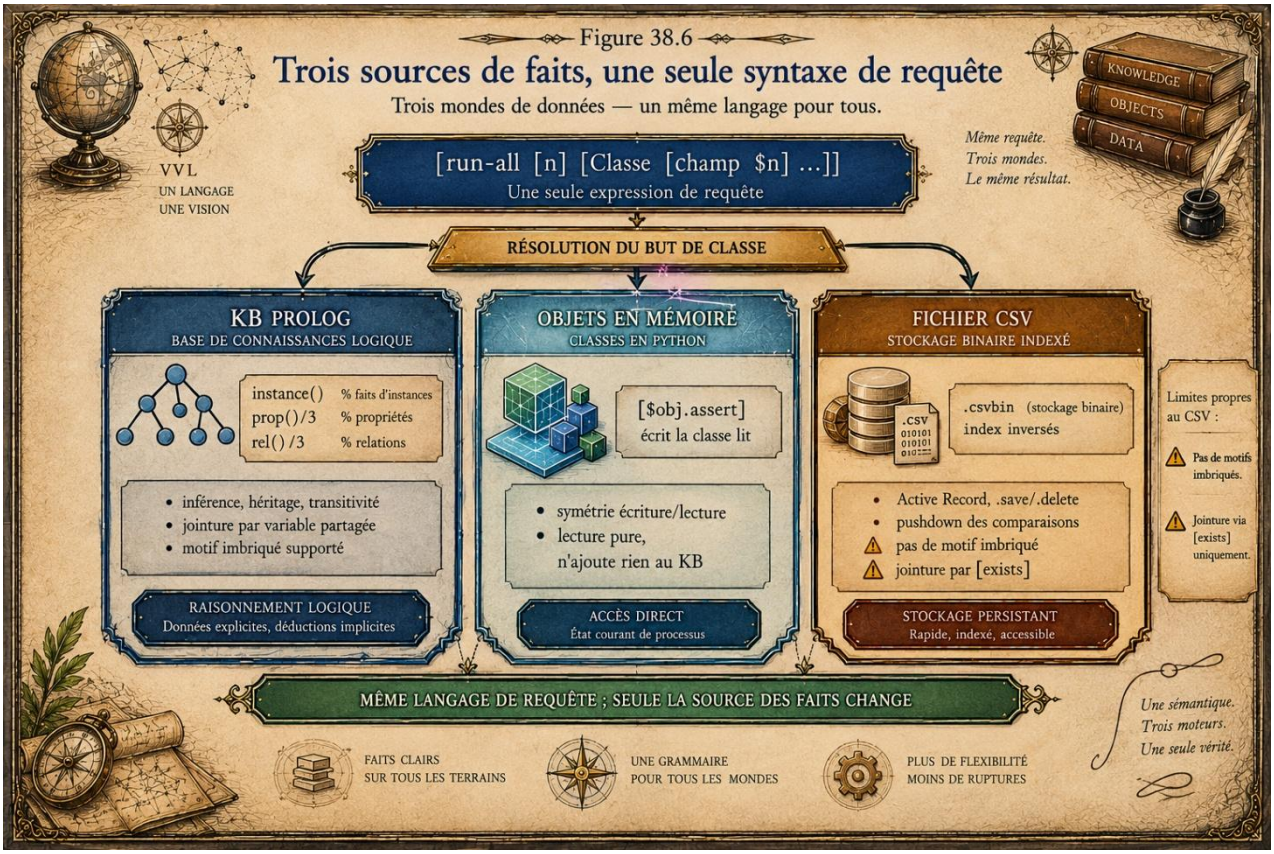


Figure 38.6 — Les trois ponts : le but de classe se résout sur le KB, les objets ou le CSV

La conception mérite d'être soulignée, car elle illustre une constante de VVL : zéro nouveau moteur. On réutilise le module CSV existant — stockage binaire, index inversés, opérations where, find, add, update, delete — sans le modifier, si bien que l'usage CSV antérieur reste intact. Le langage de requête est le même que celui du pont KB ; on ne fait que rediriger la source des faits. Et comme le scan ne parle qu'aux données et aux index du CSV, un futur stockage colonnaire ou mappé en mémoire, au-delà de la RAM, ne changerait ni le pont ni les requêtes.

38.12. eval relationnel : synthèse de programmes et quines

Les sections précédentes décrivaient des relations sur des données. On franchit ici un cap en écrivant un interpréteur relationnel — une relation eval qui relie un programme à sa valeur. Comme tout en

miniKanren, elle est réversible : tournée à l'envers, elle synthétise des programmes, et à la limite des quines. Les trois briques de cette section datent de la version langage 7.1.7.

Une question se pose d'emblée : pourquoi ne pas réutiliser l'évaluateur de VVL ? Le langage est homoiconique, après tout. Mais il n'expose pas de fonction eval de première classe, et son évaluateur de lambda est une fonction à sens unique, non réversible. Un eval relationnel doit donc réimplémenter l'évaluation comme une relation, sur une représentation décomposable par l'unification. On réutilise alors la syntaxe de VVL, pas son interprète.

D'où la première brique : le quasiquote comme constructeur de terme. Sans backtick, lambda, list et quote sont des builtins, et en position de donnée logique ils seraient évalués. Le backtick les fige en données. La virgule, elle, ouvre un trou : ,$x est une variable logique qui décompose ou capture un sous-terme.

```
[run-all [q] [unify $q `[a b c]]]
# => [[a b c]]                          un TERME ; les mots restent des mots

[run-all [q] [unify $q `[lambda [x] x]]]
# => [[lambda [x] x]]                   le builtin est gardé comme DONNÉE

[run-all [b] [unify `[lambda [a] ,$b] `[lambda [a] [list a a]]]]
# => [[list a a]]                       ,$b capture le sous-terme
```

C'est la séparation de l'objet et du méta. Dans un programme, une variable est un symbole — un mot. Dans un motif, ,$x est la variable logique qui filtre. On écrit donc ses motifs comme du vrai VVL, sans tags artificiels.

La seconde brique est [excludes $terme atome], l'absento de cKanren : l'atome n'apparaît nulle part dans le terme, ni maintenant ni après instanciation. C'est une contrainte différée, qui se suspend tant que le terme porte des variables libres et se re-vérifie à chaque liaison, avec affichage résiduel comme les contraintes de la section 38.6.

```
[run-all [x] [all ($x == `[b c]) [excludes $x a]]]   # => [[b c]]
[run-all [x] [all ($x == `[b a]) [excludes $x a]]]   # => []      a apparaît

# différée : posée AVANT la liaison fautive, rejetée tout autant
[run-all [x] [all [excludes $x a] ($x == `[b a])]]   # => []
[run-all [x] [all [excludes $x a] ($x == `[b c])]]   # => [[b c]]

[run-all [x] [excludes $x closure]]
# => [[_.0 [[excludes _.0 closure]]]]                résiduelle
```

Le petit langage interprété est de la syntaxe VVL quotée : un nombre s'auto-évalue, `[quote d] rend une donnée littérale, `[list e ...] construit, `[lambda [x] corps] définit une fonction, `[f a] applique, et un symbole est une variable cherchée dans l'environnement — une liste d'association. Trois relations suffisent.

```
[predicate lookup [x env val]
    [exists [n v rest]
        [unify $env [cons [cons $n $v] $rest]]
        [any [all [unify $n $x] [unify $v $val]]
             [all [distinct $n $x] [lookup $x $rest $val]]]]]

[predicate eval-list [exprs env vals]
    [any [all [unify $exprs []] [unify $vals []]]
         [exists [e es v vs]
            [unify $exprs [cons $e $es]] [unify $vals [cons $v $vs]]
            [eval $e $env $v] [eval-list $es $env $vs]]]]

[predicate eval [expr env val]
    [any
        # un nombre s'évalue en lui-même
        [all [a-number $expr] [unify $val $expr]]
        # (quote d) -> d ; la donnée citée ne doit jamais contenir 'closure
        [exists [d] [unify $expr `[quote ,$d]] [excludes $d closure] [unify $val $d]]
        # (list e ...) -> liste des valeurs
        [exists [args vals] [unify $expr [cons `list $args]]
            [eval-list $args $env $vals] [unify $val $vals]]
        # variable (un symbole) -> lookup
        [all [a-symbol $expr] [lookup $expr $env $val]]
        # (lambda (x) body) -> clôture
        [exists [x body] [unify $expr `[lambda [,$x] ,$body]]
            [unify $val `[closure ,$x ,$body ,$env]]]
        # (f a) -> application
        [exists [f a x body cenv arg] [unify $expr `[,$f ,$a]]
            [eval $f $env `[closure ,$x ,$body ,$cenv]]
            [eval $a $env $arg]
            [eval $body [cons [cons $x $arg] $cenv] $val]]]]
```

Le contrôle [excludes $d closure] dans la branche quote empêche une clôture — une valeur interne du moteur — de fuiter dans une donnée citée, ce qui nettoie la synthèse inverse. Dans le sens normal, l'évaluation se comporte comme attendu.

```
[run 1 [v] [eval `[quote 42] [] $v]]                    # => [42]
[run 1 [v] [eval `[list 1 2 3] [] $v]]                  # => [[1 2 3]]
[run 1 [v] [eval `[[lambda [x] x] 7] [] $v]]            # => [7]
[run 1 [v] [eval `[[lambda [x] [list x x]] 5] [] $v]]   # => [[5 5]]
```

Le sens inverse est celui qui frappe : la même relation, but inversé, répond à la question « quels programmes valent 5 ? » — et énumère le littéral, la forme quotée, puis des formes lambda.

```
[run 4 [e] [eval $e [] 5]]
# => [5  [quote 5]  ...]      littéral, quote, puis des formes lambda
```

Une quine est un programme qui s'évalue en lui-même, soit eval q [] q. La quine triviale se trouve instantanément, et sa réponse est remarquable : le moteur ne rend pas une valeur mais une contrainte résiduelle, c'est-à-dire l'énoncé général « n'importe quel nombre est une quine ».

```
[run 1 [q] [eval $q [] $q]]
# => [[_.0 [[a-number _.0]]]]
```

La quine auto-reproductrice classique, à base de lambda, existe aussi dans l'espace des solutions. On le prouve par aller-retour : évaluer Q rend une valeur identique à Q.

```
# Q = ((lambda (x) (list x (list (quote quote) x)))
#      (quote (lambda (x) (list x (list (quote quote) x)))))
[run 1 [v]
    [eval `[[lambda [x] [list x [list [quote quote] x]]]
           [quote [lambda [x] [list x [list [quote quote] x]]]]] [] $v]]
# => [[[lambda [x] [list x [list [quote quote] x]]]
#      [quote [lambda [x] [list x [list [quote quote] x]]]]]]
#    ... soit EXACTEMENT Q : auto-reproduction vérifiée.
```

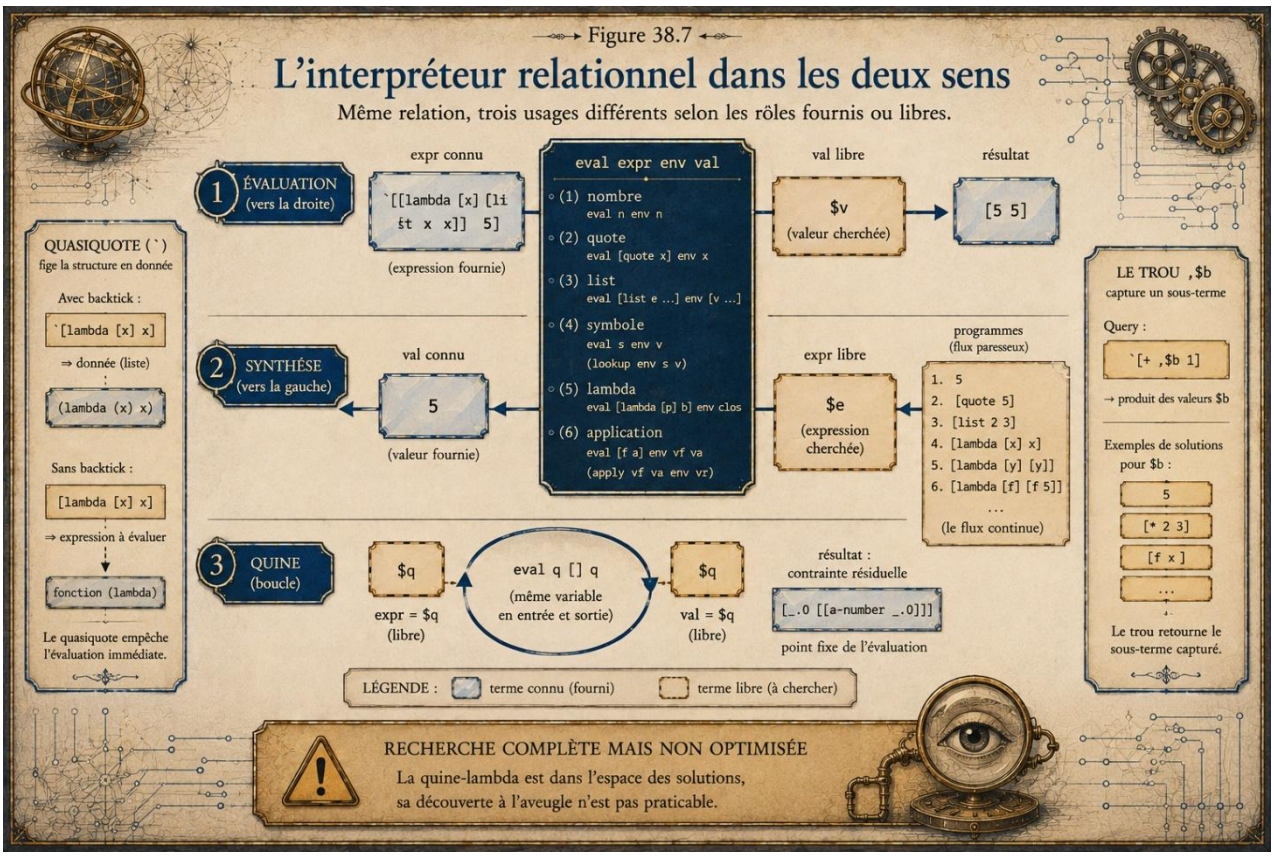


Figure 38.7 — La même relation évalue, synthétise, ou cherche son point fixe

Une limite doit être énoncée franchement. Trouver cette quine à l'aveugle — en lançant [run 1 [q] [eval $q [] $q]] tout en forçant $q à être une application — n'aboutit pas en temps pratique sur ce moteur. La recherche est complète, donc la solution serait atteinte en un temps fini, mais le moteur n'est pas optimisé comme un faster-miniKanren. La quine est bien dans l'espace des solutions, ainsi que l'aller-retour ci-dessus le prouve ; c'est la recherche aveugle qui est trop lente. Complétude et efficacité sont deux propriétés distinctes, et il serait malhonnête de laisser croire que la première entraîne la seconde.

38.13. Synthèse : ce que la logique relationnelle ajoute à VVL

Le moteur repose sur trois principes qu'il vaut la peine de reformuler. Premièrement, un seul langage : un but est une valeur VVL composable, pas un sous-langage. [any-weighted], [constrain], [fact] et une requête de classe se mélangent dans le même [run-all]. Deuxièmement, [run-all] est le résolveur : les buts sont inertes, et seuls [run], [run-all] et [run-stream] déclenchent la résolution et reifient les variables de requête. Troisièmement, la réversibilité : une relation se résout dans toutes les directions.

Le tableau des capacités effectivement livrées, de la version langage 7.0.0 à la 7.1.7, récapitule le périmètre : unification préfixe et infixe avec occurs-check ; recherche complète par entrelacement ; relations récursives et réversibles ; variables logiques isolées des variables VVL ; diségalité cKanren avec affichage des résiduelles ; comparaisons d'ordre à propagation différée ; contraintes de type et de domaine fini ; contraintes symboliques par SymPy ; contrainte d'absence absento ; code-comme-donnée par quasiquote et trou unquote ; interpréteur relationnel réversible ; négation par l'échec ; pont KB Prolog avec jointures et héritage ; pont OO ↔ KB ; pont OO ↔ CSV avec pushdown et index ; agrégation par composition ; jointures de graphe ; probabiliste façon ProbLog avec poids Beta. Le moteur est stack-safe — trampoline et reify itérative — et tous ses buts sont marqués EffectKind.PURE au sens du protocole d'atomes du chapitre 23, ce qui les rend sûrs pour du code non fiable.

Sur le plan de l'architecture de l'ouvrage, ce chapitre ajoute une pièce que les précédents laissaient manquante. Le chapitre 11 apportait l'inférence, le 12 l'ontologie, le 27 l'apprentissage de programmes logiques. Il manquait la capacité d'interroger dans n'importe quelle direction, sur une recherche dont la complétude est garantie, en composant des sources de faits hétérogènes — mémoire, base Prolog, objets, fichiers tabulaires — sous une syntaxe unique. C'est aussi, avec la section 38.12, le premier endroit du livre où le langage se retourne complètement sur lui-même : non plus générer du code par LLM et le vérifier, comme au chapitre 37, mais le synthétiser par résolution logique à partir de la seule spécification de sa valeur.

La comparaison avec le chapitre précédent est éclairante et elle clôt cette paire de chapitres. La preuve formelle vérifie ce qu'un oracle propose ; la logique relationnelle engendre les candidats sans oracle du tout. Les deux se rejoignent sur un point : ni l'une ni l'autre ne prétend passer à l'échelle sur des espaces arbitraires, et toutes deux préfèrent le dire. Dans un ouvrage consacré à des systèmes qui hallucinent, cette honnêteté sur les limites n'est pas un détail de documentation — c'est la propriété qui rend le reste utilisable.

39. La VM Rust : le calcul symbolique passe natif

Mots-clés : VM Rust, vvl-vm, [compile], couverture, parité, bignum, Prolog SLD natif, OntologyFixpoint, quote, match, TCO, frontière hôte, host bridge, codec wire

Les deux chapitres précédents ont ajouté des capacités au langage. Celui-ci n'en ajoute aucune : il raconte comment le langage existant s'est mis à s'exécuter dix à trente fois plus vite, sans que rien de sa sémantique ne change. C'est le plus gros chantier de la période — environ trois cents commits — et le plus instructif, parce que sa principale décision technique fut d'annuler une optimisation qui fonctionnait.

VAILS embarque une machine virtuelle écrite en Rust, nommée vvl-vm, distribuée sous forme de binaire dans le serveur — un par système d'exploitation. Elle exécute nativement un sous-ensemble de VVL, exposé à l'utilisateur par une forme unique : [compile]. Le principe directeur tient en trois mots — une seule langue. On écrit du VVL normal ; [compile] le compile vers le sous-ensemble VM, et le compilateur décide lui-même de ce qui s'exécute nativement, de ce qui est réécrit, de ce qui repart vers l'hôte Python, et de ce qui n'est pas encore supporté.

```
$p = [compile [+ [* 2 3] 4]]
[$p.run]                          # => 10        exécution sur la VM Rust
[$p.ir]                           # => l'IR JSON compilé
[$p.coverage]                     # => native / expand / host / unsupported
[$p.bytecode]                     # => le bytecode de la VM à pile

[[compile [/ 7 2]].run]           # => 3.5       en une ligne
```

Le chapitre suit l'ordre de construction : l'opérateur de compilation et son diagnostic de couverture, la campagne de parité qui a précédé tout le reste, l'arithmétique exacte, les deux moteurs d'inférence natifs — Prolog puis ontologie —, la métaprogrammation, l'objet et les données mutables, les fonctions récursives compilées, et enfin la frontière hôte, qui est moins une liste de lacunes qu'une déclaration de principe.

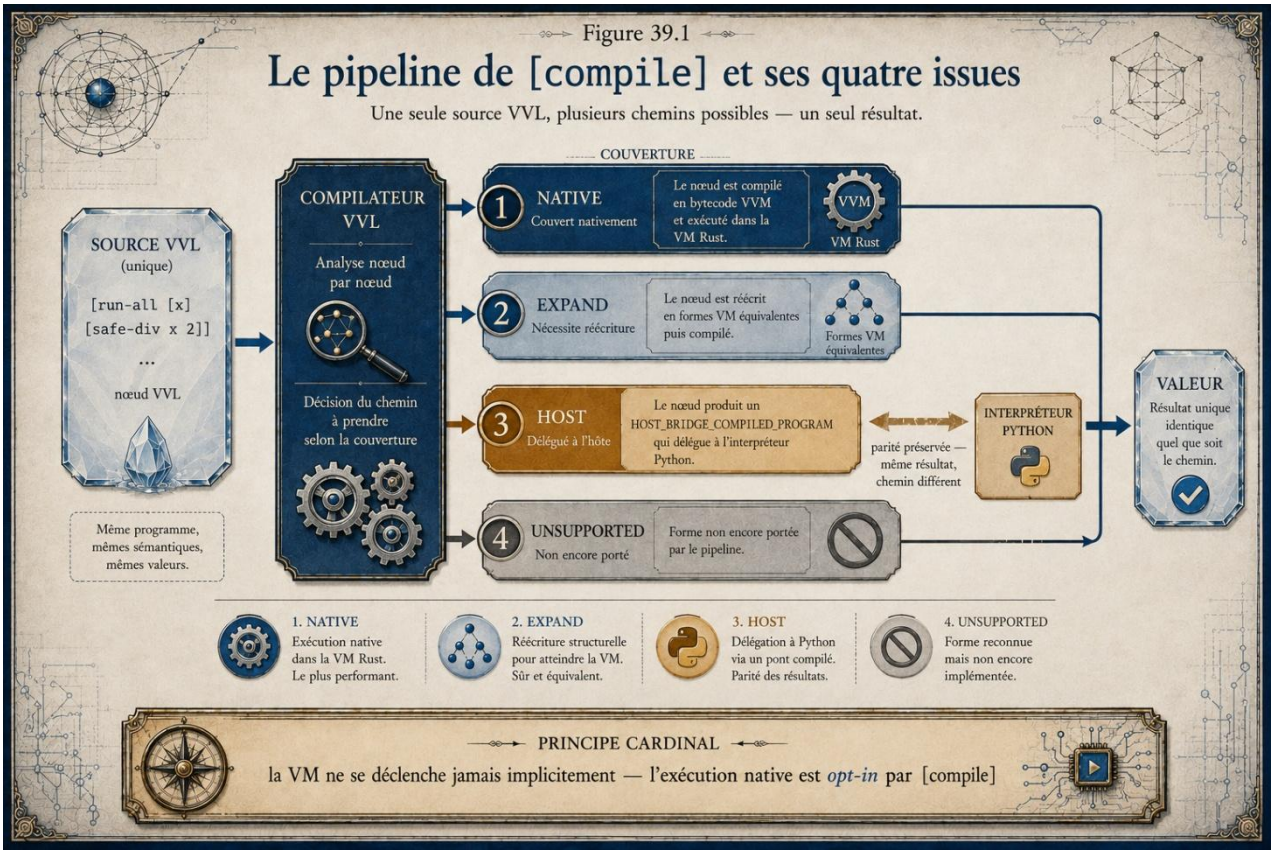


Figure 39.1 — Un seul langage en entrée, quatre chemins d'exécution, un seul résultat

39.1. Une seule langue, un opérateur explicite

La tentation, quand on adjoint une machine virtuelle rapide à un interpréteur, est de la faire intervenir automatiquement dès qu'elle le peut. VVL a choisi l'inverse, et cette décision structure tout le reste : la VM ne se déclenche jamais implicitement. L'exécution native est opt-in, uniquement via [compile]. La section 39.6 racontera ce que cette règle a coûté et pourquoi elle a été maintenue.

[compile] accepte une expression ou un chemin de fichier .vvl, et rend un objet programme réutilisable. Le bytecode est gardé chaud côté worker, si bien qu'un même programme peut être exécuté plusieurs fois sans recompilation. L'exécution des .run concurrents est parallélisée par un pool borné de workers collants par thread, réglé par la variable VVL_VM_WORKERS, ce qui apporte un facteur d'environ quatre sur les charges concurrentes.

39.2. Le diagnostic de couverture

La couverture est l'outil central de ce chapitre, et il mérite d'être compris avant les performances. [$p.coverage] classe chaque appel du programme en quatre catégories : native pour ce qui s'exécute dans le bytecode Rust, expand pour ce qui est réécrit en formes VM équivalentes, host pour ce qui est délégué à l'interpréteur Python, et unsupported pour ce qui n'est pas porté.

```
1  [[compile [+ 1 2]].coverage]
2  # => {"schema": "vvl-compile-coverage.v1", "target": "vm-rs",
3  #     "compilable": true, "native": true,
4  #     "summary": {"calls": 1, "native": 1, "expand": 0, "host": 0, "unsupported": 0}}
```

Le type du programme compilé indique d'emblée le régime d'exécution. Un calcul pur donne un COMPILED_PROGRAM ; une forme à effet donne un HOST_BRIDGE_COMPILED_PROGRAM. Dans les deux cas, [[compile prog].run] rend exactement le même résultat que prog exécuté normalement — seul le chemin diffère.

```
[getType [compile [+ 1 2]]]
# => "COMPILED_PROGRAM"                calcul pur -> natif

[getType [compile [ontology:fromText $S "Person" $txt]]]
# => "HOST_BRIDGE_COMPILED_PROGRAM"    appel LLM -> hôte
```

Cette transparence n'est pas un confort de développeur : c'est ce qui rend la frontière auditable. Un programme dont on croit qu'il tourne en natif mais qui repart silencieusement vers Python serait une régression de performance invisible. La couverture interdit ce genre de surprise, et la politique du projet en tire une règle : aucune divergence ne doit rester silencieuse.

39.3. La campagne de parité et le catalogue des divergences

Avant d'accélérer quoi que ce soit, il fallait prouver que les deux exécutions donnent le même résultat. La méthode retenue est simple et elle a été appliquée sans exception : pour chaque fonction du langage, un test en interpréteur et un test en [[compile f].run], exécutés en live contre le serveur, puis comparaison stricte. Quatre passes successives ont porté la matrice de 96 à 117 puis 133 fonctions, jusqu'à une parité de 38 sur 38 sur la matrice de référence et 16 sur 16 sur la consolidation.

L'intérêt du procédé est qu'il a produit un catalogue de divergences, chacune tranchée explicitement plutôt que contournée. Onze cas ont été relevés, et la façon dont ils ont été résolus en dit long sur la discipline du projet : dans six cas c'est la VM qui a été alignée sur l'interpréteur, mais dans quatre cas c'est l'interpréteur qui a été corrigé, parce que c'était lui qui avait tort.

Le cas D1 concerne l'arrondi : [round 0.5] rendait 0 en interpréteur, arrondi bancaire au pair le plus proche, et 1 dans la VM, arrondi au supérieur. La VM a été alignée. Les cas D2 et D3 portaient sur des signatures divergentes de [join] et [append], également alignées sur l'interpréteur.

Le cas D4 est plus intéressant. [first [list 10 20]] rendait la liste entière côté interpréteur et 10 côté VM. Ce n'est pas une différence d'implémentation mais de sémantique : [first] n'est pas un accesseur de tête de liste, c'est une forme de contrôle qui organise une course concurrente et rend la première valeur non nulle qui arrive, en élaguant les branches perdantes. La VM a été alignée sur ce sens, et l'accès à la tête d'une liste passe désormais par l'index. Une divergence de comportement a ainsi mis au jour une ambiguïté de documentation.

Les cas D5, D9 et D11 vont dans l'autre sens. [upper], [lower], [trim] et [substring] existaient dans la VM mais pas dans l'interpréteur : ils y ont été ajoutés. Et surtout, D9 : [new C [x 3]] appliquait les paires nommées côté VM mais les ignorait côté interpréteur, laissant le champ à sa valeur par défaut. C'était un bug de l'interpréteur, corrigé en version langage 7.3.322 — désormais, en l'absence de méthode init, les paires nommées sont affectées des deux côtés et un champ inconnu est rejeté.

Les cas D6, D7 et D8 relevaient du rendu : formatage JSON avec espaces à la manière de json.dumps, retour d'un tableau json-decode retagué pour rester un JSON côté hôte plutôt que de devenir une liste VVL, et [avg] rendant toujours un flottant. Le cas D10 alignait les alias miniKanren du chapitre précédent : disj et conde vers any, conj vers all, fresh vers exists — le compilateur de buts n'honorait pas ses propres alias.

Ce catalogue est une pièce importante du dossier de crédibilité de la VM. Un projet qui publierait « parité atteinte » sans exhiber ses divergences demanderait à être cru sur parole ; celui-ci fournit la liste, la décision prise pour chacune, et le numéro de version qui la résout.

39.4. Arithmétique exacte : bignum, promotion, démotion

Un langage qui manipule des connaissances ne peut pas se permettre un débordement silencieux à soixante-quatre bits. La VM implémente donc les entiers de précision arbitraire, arrivés en version 0.7.3.

Le mécanisme est à trois temps. Un littéral au-delà de 2^{63} est compilé en nœud BigInt, ce qui lui permet de survivre à la sérialisation. Les opérations +, − et × partent sur le chemin rapide en entier 64 bits et promeuvent automatiquement en grand entier exact lorsque le résultat déborde. Et dès qu'un résultat repasse dans les 64 bits, il redescend — cette démotion garantit une forme canonique unique et préserve le chemin rapide.

```
1  [[compile 9223372036854775808].run]                  # => 9223372036854775808
2  [[compile [+ 9223372036854775807 1]].run]             # => 9223372036854775808
3  [[compile [* 10000000000000000000 10000000000000000000]].run]
4  # => 100000000000000000000000000000000000000       10^40 exact
5  [[compile [- 9223372036854775808 1]].run]             # => 9223372036854775807   démotion
6  [[compile [> 9007199254740993 9007199254740992]].run] # => 1   comparaison exacte au-delà de
2^53
```

La limite est documentée sans détour, et il faut la connaître. Seuls +, − et × ainsi que les comparaisons sont exacts en grand entier. Les agrégats — [sum], [avg], [min], [max] — et les fonctions mathématiques — [abs], [pow], [sqrt] — repassent par un flottant double précision et perdent donc l'exactitude sur de très grands entiers.

```
1  [[compile [sum [list 9223372036854775807 9223372036854775807]]].run]
2  # => 18446744073709551616   attention : devrait finir par 614, sum passe par f64
3  [[compile [pow 2 100]].run]
4  # => 1.2676506002282294e+30   attention : flottant, pas 2^100 exact
```

39.5. Un moteur Prolog SLD natif — et deux bugs révélés dans l'hôte

Le bloc 0.8 apporte un moteur Prolog écrit en Rust. La décision de conception mérite d'être relevée : plutôt que de réemployer le moteur miniKanren du chapitre 38, l'équipe a écrit un moteur SLD dédié. Les deux sémantiques diffèrent — miniKanren fait de la recherche complète par entrelacement, Prolog de la résolution SLD dans l'ordre avec cut — et tenter de faire porter les deux par le même moteur aurait dégradé les deux.

Sous [compile], la base de connaissances est snapshotée à la compilation et la requête résolue nativement. Quatre formes de requête sont exposées : [prologExists] rend 1 ou 0 selon l'existence d'une solution, [prologSolutions] rend toutes les solutions, [prologValues] la liste des valeurs d'une variable, et [prologFirstValue] la première valeur ou un défaut.

```
[facts "fam" "parent(tom, bob)" "parent(tom, sue)" "parent(bob, ann)"]
[rules "fam"
    "ancestor(X, Y) :- parent(X, Y)"
    "ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y)"]

[[compile [prologValues "fam" "parent(tom, X)" "X"]].run]        # => ["bob" "sue"]
[[compile [prologSolutions "fam" "ancestor(tom, X)"]].run]
# => [[["X" "bob"]] [["X" "sue"]] [["X" "ann"]]]              clôture transitive

# le cut coupe les alternatives
[rules "fam" "maxof(A, B, A) :- A >= B, !"]
[facts "fam" "maxof(A, B, B)"]
[[compile [prologFirstValue "fam" "maxof(9, 2, M)" "M" "?"]].run]   # => "9"
[[compile [prologFirstValue "fam" "X is 2 + 3 * 4" "X" "?"]].run]   # => "14"
```

Le moteur couvre le vrai Prolog et pas seulement le fragment de Horn : cut, négation par l'échec, arithmétique par is, comparateurs, findall. Les requêtes employant des prédicats de bibliothèque non implémentés nativement — length, member — retombent proprement sur l'hôte. Sur récursion profonde, grâce à une substitution avec trail défaisable en temps constant, le moteur natif est environ 7,4 fois plus rapide que le moteur Python, à résultat identique.

Mais l'épisode le plus intéressant est ailleurs. En testant la parité entre les deux moteurs, l'équipe a découvert deux bugs — non pas dans le moteur Rust, qui était correct, mais dans le moteur Python de référence, en service depuis longtemps. Le cut était totalement cassé, et il fuyait d'une requête à l'autre. Et la négation préfixe écrite avec une espace se parsait en n'importe quoi. Autrement dit, la réimplémentation a servi d'oracle à l'implémentation d'origine.

Ce phénomène — le natif révèle l'hôte — mérite d'être noté par quiconque envisage de réécrire un composant. Le bénéfice attendu était la performance ; un bénéfice non anticipé fut la vérification. Écrire deux fois la même sémantique et exiger qu'elles coïncident est une forme de test que ni l'une ni l'autre des implémentations ne pouvait fournir seule. C'est, à l'échelle d'un moteur, le même principe que le noyau de vérification du chapitre 37.

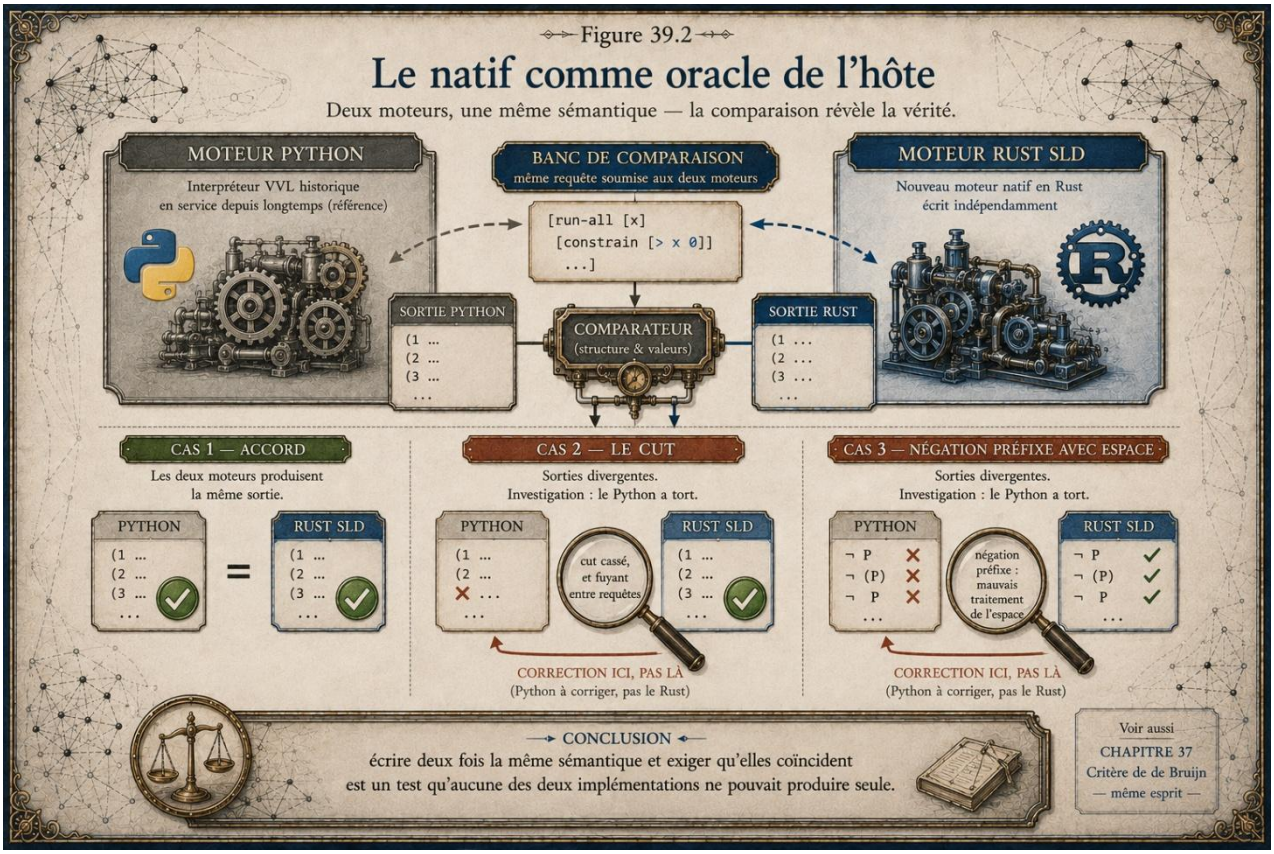


Figure 39.2 — La réimplémentation native a servi de vérificateur à l'implémentation d'origine

39.6. L'ontologie native, ou l'optimisation annulée volontairement

Voici l'épisode central de ce chapitre, et le seul qui relève de la doctrine d'ingénierie plutôt que de la technique. Il se raconte en trois temps.

Premier temps. Le moteur d'inférence ontologique est le chemin de calcul le plus lourd du langage : sur un banc de cent cinquante instances saturées de 0 à 12, soit mille huit cents modifications, l'interpréteur mettait 16,6 secondes. Un fast-path natif fut écrit, qui détectait un appel à [ontology:infer-fixpoint] et le routait vers la VM. Résultat : 1,2 seconde, soit un gain mesuré de 13,8 fois, avec une parité de base de connaissances prouvée. La VM fut déclarée 1.0.0 et le serveur passa en 7.14.0.

Deuxième temps. Ce fast-path se déclenchait implicitement, depuis un builtin, sans que l'utilisateur ait écrit [compile]. Il violait donc le principe posé à la section 39.1. La décision fut prise de le retirer : le dispatch implicite fut supprimé en version langage 7.3.340, et la VM rétrogradée de 1.0.0 à 0.9.1 — un serveur qui annule publiquement une montée de version majeure et abandonne un facteur 13,8 déjà acquis.

Troisième temps. L'ontologie fut réimplémentée entièrement, cette fois via [compile] uniquement. Ce n'était pas un simple déplacement d'appel : le verrou technique était que les règles portent du code. Une règle s'écrit [when condition] [then actions], où condition et actions sont des arbres syntaxiques VVL arbitraires. Deux voies s'offraient : extraire ces arbres à la compilation et les incorporer dans l'instruction OntologyFixpoint, ou introduire une valeur-clôture dans la VM pour stocker le code comme donnée. La seconde était plus générale mais exigeait une fondation absente ; la première fut choisie, avec sa limite assumée — les règles doivent être statiquement présentes dans le bloc compilé.

Une contrainte apparut alors, qui explique la forme finale : on ne peut pas compiler [ontology:infer-fixpoint] isolément à parité. L'interpréteur rend un compteur de modifications et mute la base du schéma ; l'état — schéma, base, instances — est enfilé de [ontology:define] à [new] puis [ontology:commit], [ontology:infer] et [ontology:entails]. La parité observable exige donc que tout le programme tourne dans la même exécution, sur un état ontologique natif partagé. Le livrable n'est pas une fonction accélérée mais un programme ontologique complet.

```
$prog = [compile [do
    # schéma : une classe avec attributs typés et contraintes
    [$S = [ontology:define "shop"
        [ontology:class widget
            [attributes
                [price Integer [required true] [min 0] [max 1000]]
                [tier  String]]]]]

    # une règle d'inférence
    [ontology:rule "bump" $S
        [target widget]
        [when  ($this.price < 100)]
        [then  ($this.price = ($this.price + 10))]]

    # des instances, puis inférence jusqu'au point fixe, puis requête
    [$w1 = [new widget [price 50] [tier "gold"]]]
    [$w2 = [new widget [price 95] [tier "silver"]]]
    [ontology:commit [$w1 $w2]]
    [ontology:infer-fixpoint $S 30]
    [ontology:select $S "widget"]]]

[$prog.run]
# => [{price: 100, tier: "gold", _id: <id>} {price: 105, tier: "silver", _id: <id>}]
```

La règle est bien appliquée : la première instance passe de 50 à 100 en cinq passes, la seconde de 95 à 105 en une, et le select qui suit, dans le même programme, voit les prix saturés. Sept formes sont natives — define, rule, new suivi de commit, infer, infer-fixpoint, entails, consistent et select — et la validation de cohérence reproduit les messages exacts de l'interpréteur, y compris le remplissage des valeurs par défaut de classe.

Le serveur bascula en 7.15.0 avec la mention « bascule assumée ». Le mot est bien choisi : la première bascule avait été prématurée, la seconde est revendiquée. Ce qu'il faut retenir n'est pas le facteur de gain, c'est qu'un principe architectural a prévalu sur une performance déjà mesurée et livrée. Dans un ouvrage qui défend l'auditabilité, l'exemple vaut d'être signalé : une VM qui s'active toute seule est une VM dont on ne peut pas prédire quand elle s'active.

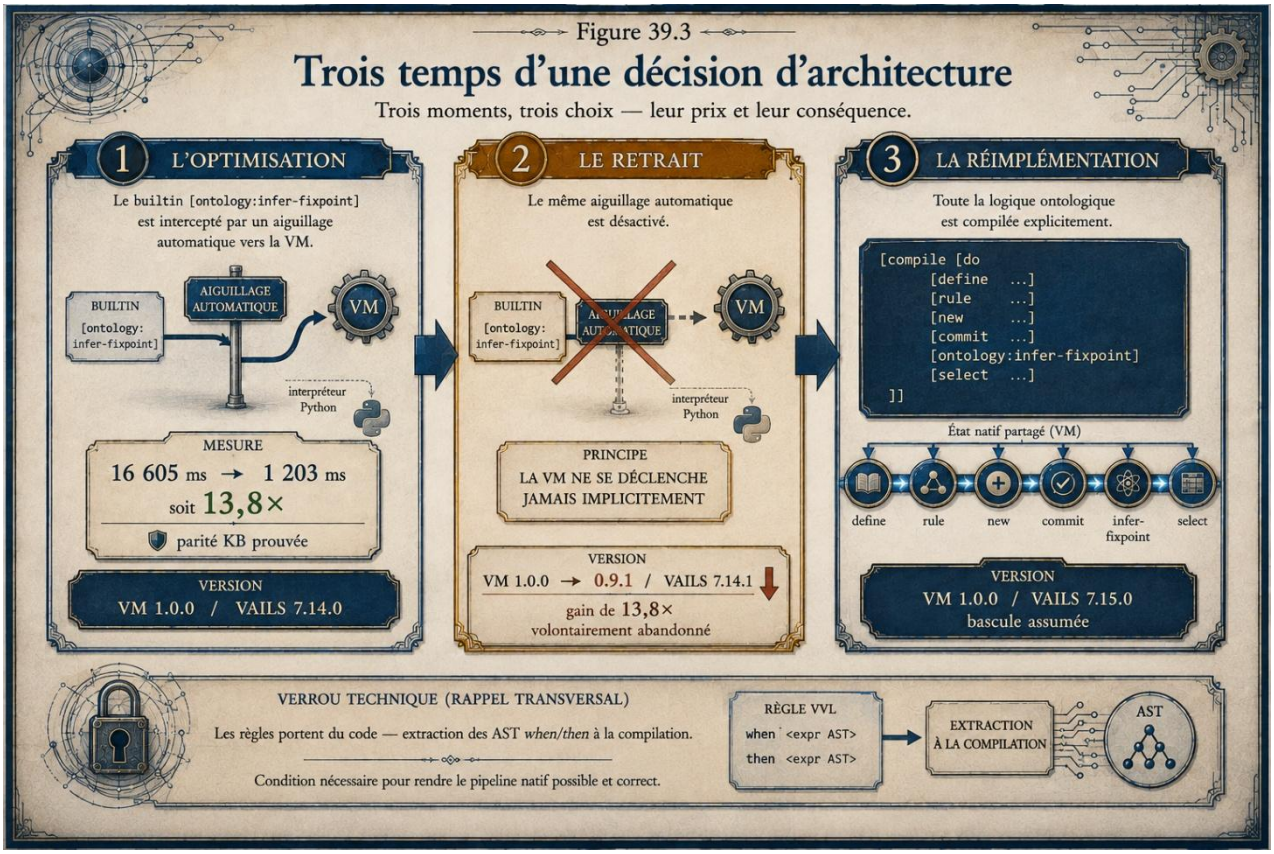


Figure 39.3 — Un facteur 13,8 abandonné pour préserver un principe, puis regagné autrement

39.7. Métaprogrammation native : quote et match

L'obstacle attendu pour porter la métaprogrammation était la représentation du code comme valeur — il aurait fallu étendre l'énumération des valeurs de la VM avec un variant AST, ce qui touche tout le moteur. Le bloc 1.0.0 résout les deux formes sans cette extension, et c'est la partie élégante du chantier.

[quote X] rend la forme brute non évaluée. Comme X est connu à la compilation, c'est une constante compile-time : le compilateur sérialise l'arbre en littéral que la VM rend tel quel, reconstruit à l'identique côté hôte grâce à des tags qui distinguent les mots des variables. Aucune extension du modèle de valeurs n'a été nécessaire.

```
1  [[compile [quote foo]].run]                    # => foo        un mot
2  [[compile [quote $x]].run]                     # => $x         une variable
3  [[compile [quote [a b c]]].run]                # => [a b c]    une liste
4  [[compile [quote [set $x [nested list]]]].run] # => [set $x [nested list]]
```

[match] devient une instruction VM dédiée, avec son propre type de motif et un matcher natif. Le langage de motifs comprend le joker, la capture qui lie une variable, le littéral comparé par affichage, la liste structurelle de même longueur, et la décomposition tête-reste.

```
[[compile [match 2  1 "one"  2 "two"]].run]              # => "two"
[[compile [match 5  "?x" [+ $__x 1]]].run]                # => 6      capture
[[compile [match 9  1 "a"  "_" "other"]].run]             # => "other"
[[compile [match 7  1 "a"]].run]                          # => ""     aucun match
[[compile [match [1 2]  [1 "?y"]  $__y]].run]             # => 2      liste structurelle
[[compile [match [1 2 3]  ("?h" :: "?t")  $__t]].run]     # => [2 3]  tête et reste
```

Les deux formes sont natives quand elles sont éligibles : quand la forme quotée fait correctement l'aller-retour, quand les motifs et les corps compilent. Sinon, elles retombent sur l'hôte, parité préservée. La parité mesurée est de 8 sur 8 pour quote et 10 sur 10 pour match.

39.8. Objet natif, données mutables et requêtes CSV

Le bloc le plus volumineux du chantier — environ cent cinquante commits — porte sur l'état mutable, et c'est aussi celui dont la documentation de synthèse manque encore côté serveur. Il recouvre le magasin d'objets natif, les canaux et sémaphores du scheduler, le protocole de réécriture CSV, les instantanés d'état des ressources et les plans de restauration.

Côté objet, la VM exécute nativement les classes avec champs par défaut et méthodes, l'instanciation par paires nommées ou par init positionnel, l'accès aux champs, l'appel de méthode avec $self, l'héritage et l'introspection.

```
[[compile [do [class Rect [field w 0] [field h 0]
     [method aire [] [* $self.w $self.h]]]
   [$r = [new Rect [w 5] [h 3]]] [$r.aire]]].run]          # => 15

[[compile [do [class Base [field w 0] [method aire [] $self.w]]
     [class Carre : Base [method init [c] [do [$self.w = $c] $self]]]
   [$s = [new Carre 4]] [$s.aire]]].run]                   # => 4
```

Quelques points de vigilance méritent d'être connus. Un champ se lit sans crochets et une méthode s'appelle avec : [$p.x] sur un champ échoue en réclamant une méthode. La forme [is-a $obj Classe] ne compile pas, le nom de classe nu étant un mot non supporté ; on emploie [class-of]. Chaque [compile] est indépendant, si bien qu'une classe parente doit être définie dans le même programme que son enfant. Et l'état de classe persistant entre les évaluations d'une session peut produire des collisions si l'on redéfinit une classe avec des champs différents.

Côté CSV, le compilateur prend un instantané des lignes et exécute les requêtes de lecture sans repasser par Python.

```
$csv = [CSV "vmdemo"]
[[compile [csv-count $csv]].run]                # => 3
[[compile [csv-where $csv "age" 30]].run]
# => [{age: 30, id: 1, nom: "Alice"} {age: 30, id: 3, nom: "Carol"}]
```

Les mutations et la persistance passent par un pont dédié : la VM possède la table, l'hôte assure l'aller-retour disque par un protocole versionné de réécriture, avec deltas, files ordonnées et lots. Le même

schéma s'applique au magasin d'objets persistés, dont la VM détient le graphe tandis que l'hôte fait le tour de la base SQLite.

39.9. Fonctions compilées récursives, frames et optimisation d'appel terminal

Le bloc 1.1.0 franchit une étape que les précédents laissaient ouverte. Jusque-là, une macro définie par [keyword] était réexpansée et réinterprétée à chaque appel. Enveloppée dans [compile], la même définition devient une vraie fonction : le corps est compilé une seule fois, et l'appel lie les arguments positionnels dans une frame puis exécute le code compilé.

```
[compile [keyword facto `[if [<= @1 1] 1 (@1 * [facto [- @1 1]])]]]
[facto 5]      # => 120
[facto 30]     # => 265252859812191058636308480000000     bigint exact
[facto 100]    # => 100! (158 chiffres), sans aucune recompilation
```

La définition et le site d'appel sont identiques à ceux d'une macro : mêmes paramètres positionnels, même syntaxe d'appel. Seule l'exécution change. Les appels se résolvent par nom dans un registre de fonctions scopé à la session. Un appel non terminal pousse une frame explicite sur le tas — et non sur la pile Rust — tandis qu'un appel en position terminale est compilé en instruction dédiée qui réutilise la frame courante, à profondeur constante.

```
# récursion terminale : un million d'appels, profondeur constante
[compile [keyword sumto `[if [<= @1 0] @2 [sumto [- @1 1] [+ @2 @1]]]]]
[sumto 1000000 0]     # => 500000500000

# récursion à deux branches, non terminale : frames sur le tas
[compile [keyword fibo `[if [< @1 2] @1 ([fibo [- @1 1]] + [fibo [- @1 2]])]]]
[fibo 25]             # => 75025

# récursion mutuelle, compilée quel que soit l'ordre de définition
[compile [keyword isodd  `[if [<= @1 0] 0 [iseven [- @1 1]]]]]
[compile [keyword iseven `[if [<= @1 0] 1 [isodd  [- @1 1]]]]]
[iseven 50000]        # => 1
```

La récursion mutuelle fonctionne dans les deux sens de définition parce qu'un appel vers un nom inconnu — ni builtin, ni forme native — est compilé en appel résolu par nom au moment de l'exécution, ce qui autorise la référence en avant.

Un piège de syntaxe mérite d'être signalé, car il est le plus fréquent. Dans un corps de keyword compilé, les crochets dénotent un appel et les parenthèses une expression infixe. Écrire [@1 * [facto …]] ne produit pas un produit mais une liste à un élément contenant l'expression, d'où une erreur de type à l'exécution. Il faut grouper les calculs avec des parenthèses : (@1 * [facto [- @1 1]]). Symétriquement, les parenthèses sont infixes uniquement — (+ a b) est une erreur de syntaxe, il faut écrire [+ a b].

Si le corps contient une forme non compilable, la définition ne casse pas : elle retombe sur la macro interprétée classique. La règle est constante dans tout ce chantier — cela accélère quand cela peut, cela ne casse jamais. Deux limites subsistent en première version : l'appel d'une fonction compilée depuis une tâche du scheduler ou en exécution pas à pas n'est pas supporté, et le traitement récursif de listes ainsi que la mémoïsation ne sont pas encore dans le sous-ensemble compilé.

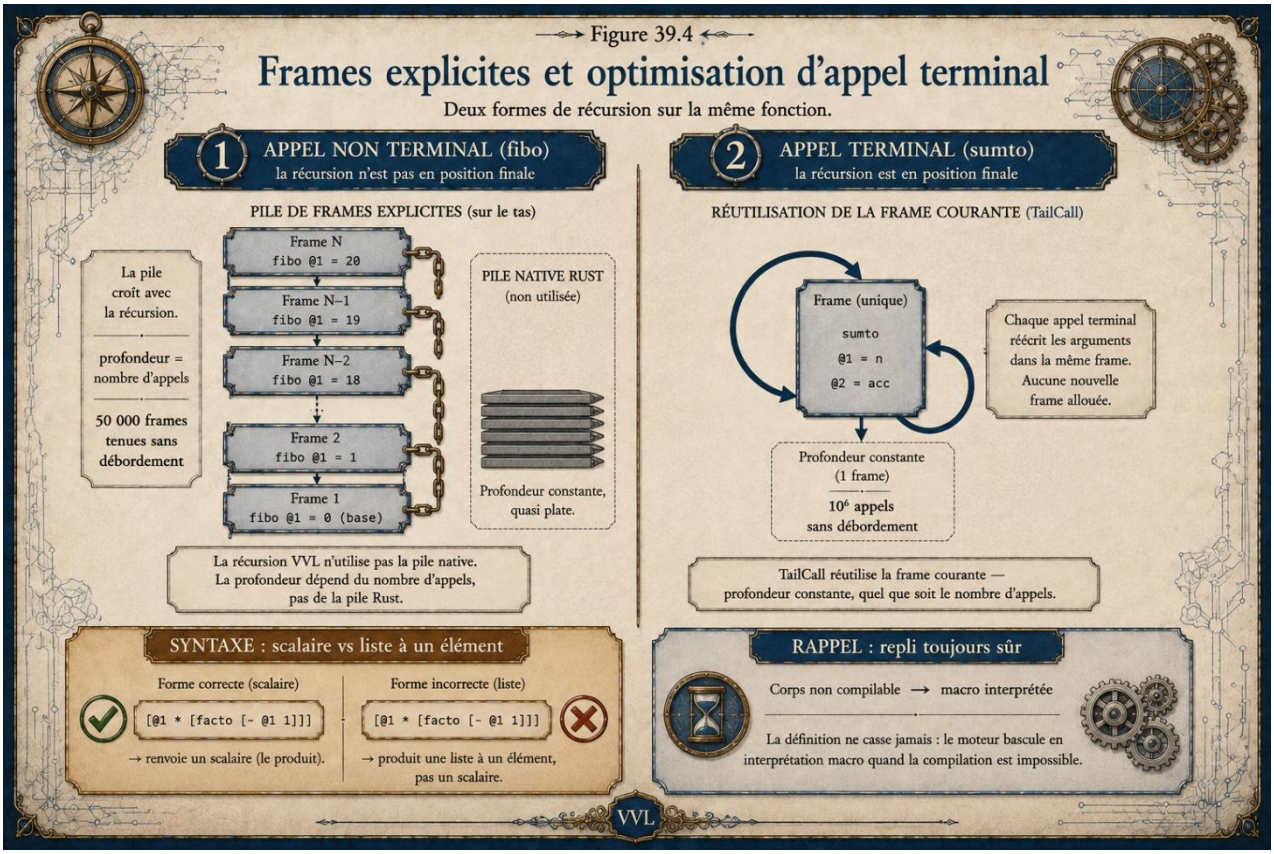


Figure 39.4 — Deux régimes d'appel, une seule syntaxe de définition

39.10. La frontière hôte assumée

La version 1.0.0 s'accompagne d'un document qui n'est pas une liste de lacunes mais une déclaration : ce qui n'est pas natif ne le sera pas, et pour une raison. La ligne de partage est celle du calcul symbolique déterministe d'un côté, de l'effet non répliquable de l'autre.

Sont hôtes par contrat : tout ce qui appelle un modèle de langage — extraction depuis texte ou image, satisfaction hybride, approximation, algorithmes génétiques, induction de règles, construction de systèmes flous ; les entrées-sorties fichier et base de données — persistance d'ontologie, chargement et écriture CSV, réécriture SQLite, indexation RAG ; la console et l'interface — [log], [input], les dialogues ; le chargement et l'affichage d'images ; les solveurs externes de planification et d'optimisation, qui appellent un moteur tiers ; la sémantique d'exception Python, puisque reproduire exactement le comportement de try et except de CPython n'aurait pas de sens ; et enfin l'ontologie avancée — mondes possibles, objets abstraits de Zalta, règles ajoutées dynamiquement — qui reste hôte pour complexité et rareté plutôt que pour effet.

La distinction est parfois fine et elle est tracée avec soin. Les opérations floues scalaires sont natives, car ce sont des calculs purs ; seule la construction d'un système flou est hôte. Les requêtes CSV sur une table possédée par la VM sont natives ; seul l'accès disque est hôte. Les transformations d'image pures sont classées calcul ; le chargement et l'affichage restent hôtes. Une règle d'ontologie écrite statiquement dans le programme compilé est native ; la même règle ajoutée au moment de l'exécution retombe sur l'hôte.

S'ajoute une catégorie intermédiaire, native quand éligible : le programme ontologique complet, quote, match, les requêtes Prolog, les requêtes CSV. Le compilateur choisit automatiquement, l'utilisateur écrit du VVL normal, et [$p.coverage] détaille le choix nœud par nœud.



Figure 39.5 — Natif ou hôte : la frontière suit l'effet, pas l'état d'avancement

39.11. Versionnement double et durcissement

Deux versions coexistent et il faut les distinguer pour lire les journaux du projet : celle du serveur VAILS, et celle du binaire VM, indépendante. Le numéro 1.0.0 de la VM marque la porte entre calcul natif et frontière hôte ; ensuite, le deuxième chiffre compte les blocs de surface fonctionnelle livrés — 1.1 pour les fonctions récursives — et le troisième les correctifs. La bannière de lancement du serveur affiche la version détectée, et la couverture reste la source de vérité pour savoir ce qui s'exécute où.

Les versions 1.2.x ne livrent aucune fonctionnalité : elles durcissent. La 1.2.0 puis la 1.2.1 introduisent un codec de communication explicite entre la VM et l'hôte. Le problème était réel : la sérialisation dérivée aplatissait les variants de valeur, si bien qu'un JSON imitant un marqueur pouvait changer d'identité en traversant la frontière. Le codec explicite préserve chaque variant — typé, signal, JSON, intervalle — à la frontière, dans les arguments d'appel de fonction et dans les continuations, et gère au passage les grands entiers que la sérialisation automatique ne passait pas.

La 1.2.2 traite la robustesse d'exécution, et ses cinq mesures se lisent comme un catalogue des façons dont un worker peut mal tourner. Un budget d'instructions borne les programmes non terminants — nécessaire précisément parce que l'optimisation d'appel terminal réutilise la frame, donc la garde de profondeur ne suffisait plus. Le solveur Prolog tourne sur un thread à grande pile avec garde de profondeur, si bien qu'une clause qui s'appelle elle-même ne fait plus déborder la pile native. Un délai côté client tue et remplace un worker calé. Des opérations de libération permettent à l'expiration d'une

session REST de rendre son cluster de fonctions compilées, évitant l'accumulation entre sessions. Deux fuites mémoire du scheduler furent enfin corrigées, la seconde exigeant de mémoriser chaque programme temporaire avec le worker qui l'avait créé — un identifiant étant propre au processus, le libérer sur un worker remplacé aurait écrasé par collision un lot fraîchement créé.

39.12. Synthèse : ce que prouve ce chantier

Trois mesures résument le gain. Une boucle de somme des carrés sur cinquante mille itérations passe de 4 502 à 141 millisecondes, soit environ 32 fois. Le moteur Prolog récursif gagne un facteur 7,4. L'inférence ontologique jusqu'au point fixe gagne 13,8. Ce sont les trois chemins de calcul les plus lourds du langage, et c'est délibéré : le critère de bascule retenu n'était pas « compile sans erreur » mais « tout le calcul perf-critique tourne nativement, avec parité prouvée en live ».

Sur le plan de l'architecture du livre, ce chapitre occupe une place particulière. Il ne parle ni de LLM ni de logique : il parle de la couche d'exécution qui porte tout le reste. Un lecteur pressé pourrait le tenir pour une annexe de performance. Ce serait manquer trois enseignements qui débordent largement la question de la vitesse.

Le premier est que la réimplémentation vérifie. Écrire deux fois la même sémantique et exiger la coïncidence a révélé, dans le moteur Prolog d'origine, deux bugs qu'aucune suite de tests n'avait attrapés. C'est le critère de de Bruijn du chapitre 37 transposé à l'échelle d'un moteur — sauf qu'ici, aucune des deux implémentations n'est le noyau de confiance : c'est leur accord qui l'est.

Le deuxième est qu'un principe peut valoir plus qu'un facteur 13,8. Le fast-path ontologique fonctionnait, il était mesuré, il était livré — et il a été retiré parce qu'il rendait imprévisible le moment où la VM s'active. Pour un système qui prétend à l'auditabilité, l'imprévisibilité du chemin d'exécution est un défaut du même ordre qu'un cache sans provenance (chapitre 3) ou qu'une trace de preuve sans noyau (chapitre 37).

Le troisième est la valeur d'une frontière énoncée. La documentation de la version 1.0.0 ne dit pas « ce qui n'est pas encore fait » mais « ce qui ne le sera pas, et pourquoi ». La différence est considérable pour qui construit dessus : une lacune invite à attendre, une frontière permet de concevoir. Là encore, c'est la même honnêteté sur les limites que celle des deux chapitres précédents — le prouveur qui répond « je ne sais pas » plutôt que de deviner, le moteur relationnel qui reconnaît que sa recherche est complète mais lente. Trois sous-systèmes très différents, une même discipline.

40. Le cluster de calcul : distribuer du VVL sur plusieurs machines

Mots-clés : cluster, master/worker, code adressé par contenu, SHA-256, shard, lease, réducteur, capability, SQLite/WAL, spool, dead-letter, ABI, gate économique

Le chapitre précédent a rendu le calcul symbolique natif sur une machine. Celui-ci le fait sortir de la machine. Le VVL Compute Cluster exécute une fonction VVL compilée et fermée sur plusieurs instances VAILS : un master conserve le code, les jobs et les baux dans une base SQLite en mode WAL ; les workers tirent des lots de travail, vérifient et mettent le code en cache, exécutent dans une VM isolée, puis mettent le résultat en file avant de l'envoyer.

Ce chapitre s'ouvre sur une question que la plupart des présentations de systèmes distribués posent en dernier, quand elles la posent : à partir de quand la distribution est-elle rentable ? Le projet y a répondu par un banc de mesure obligatoire avant d'écrire la première ligne du système, et la réponse mérite d'être connue avant les détails d'architecture.

40.1. Le gate économique : mesurer avant de construire

Distribuer un calcul a un coût : sérialiser les données, les transporter, ordonnancer, vérifier l'intégrité, rapatrier les résultats. Ce coût est fixe par unité de travail. Le gain n'existe donc que si le temps de calcul d'une unité dépasse largement ce coût de transport. Le projet a formalisé cette évidence en un seuil : un ratio calcul sur transport d'au moins cent pour déclencher un feu vert.

Les mesures, prises sur un banc dédié, sont instructives et le rapport ne les enjolive pas. Un appel natif isolé donne un ratio de 0,7 — autrement dit, distribuer coûte plus cher que calculer. Des lots de 32 à 128 appels indépendants restent sous 34. Un lot de 192 appels atteint 95,2 et devient économiquement plausible, mais reste sous le seuil. Et à 256 appels, le payload plus volumineux fait retomber le ratio à 64,1.

Ce dernier point est le plus utile : le seuil n'est ni monotone ni robuste. Augmenter la taille des lots améliore le ratio jusqu'à un certain point, après quoi le volume de données à transporter reprend le dessus. Il existe donc une granularité optimale, propre à chaque charge, et la trouver relève de la mesure, pas du raisonnement.

Le rapport énonce aussi ses propres limites, ce qui est plus rare. Le lien réel entre deux instances, TLS compris, n'a pas été mesuré — seule une boucle locale l'a été, et un réseau ne peut qu'ajouter du coût. La fonction métier réelle n'était pas disponible ; le code testé est une transposition compilée d'un cas de régression existant. La décision fut donc un feu vert conditionnel, avec une granularité cible d'au moins 192 unités pour cette charge, et une autorisation explicite du propriétaire du projet pour poursuivre malgré le seuil non atteint.

On peut voir dans cet épisode une simple prudence de gestion. Il vaut mieux y lire la même discipline que dans les trois chapitres précédents : le prouveur qui répond « je ne sais pas », le moteur relationnel qui distingue complétude et efficacité, la VM qui publie son catalogue de divergences. Ici, un système distribué qui documente le régime dans lequel il n'apporte rien.

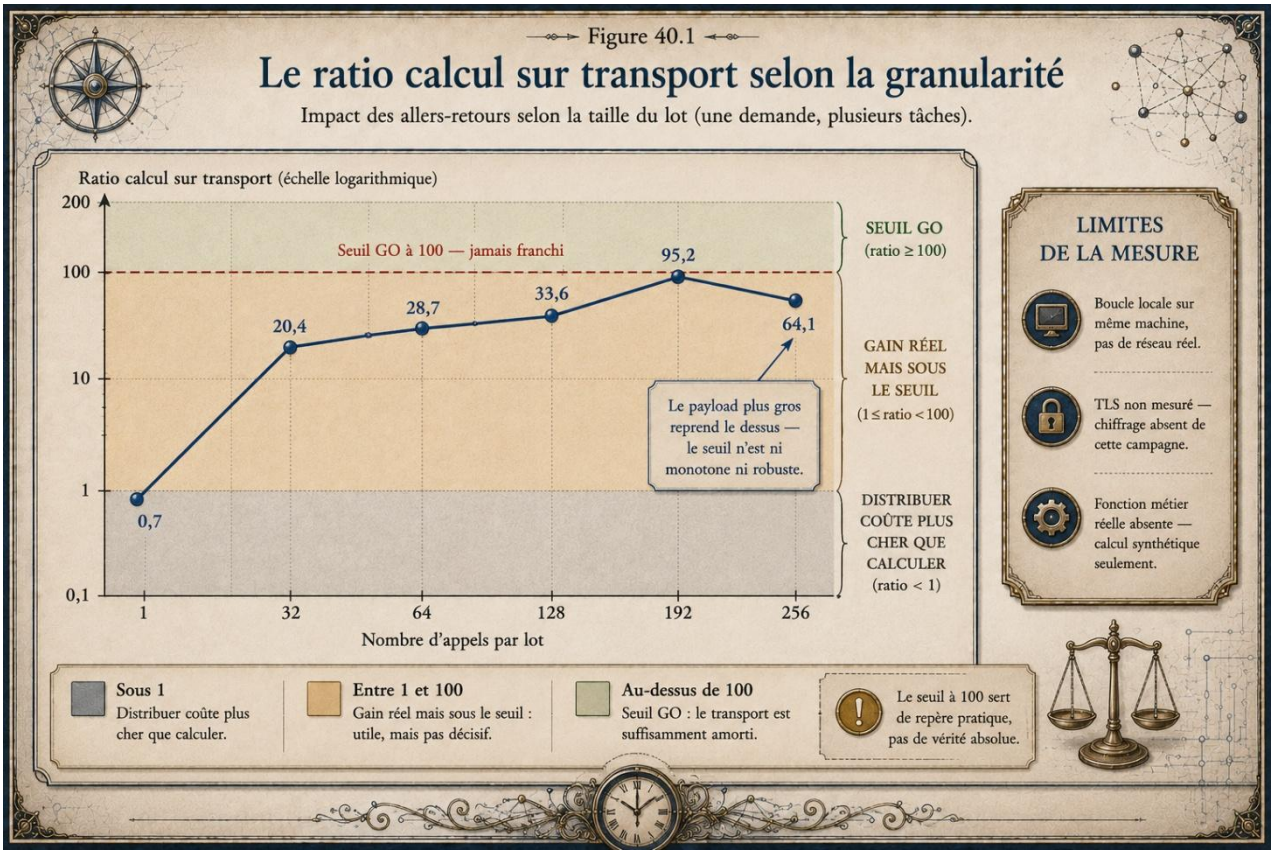


Figure 40.1 — Il existe une granularité optimale, et elle se mesure — elle ne se déduit pas

40.2. Architecture : master, workers et code adressé par contenu

Une instance VAILS prend l'un de trois rôles, fixé par sa configuration : standalone, master ou worker. Le master expose l'API cliente et ordonnance ; les workers exécutent et n'exposent pas cette API ; une instance standalone fait les deux avec ses processus locaux.

La pièce centrale est le magasin de code adressé par contenu. Une fonction enregistrée n'est pas désignée par un nom mais par le hachage SHA-256 de son code compilé. La référence rendue à l'enregistrement a la forme sha256: suivie de soixante-quatre caractères hexadécimaux, et elle est stable pour ce contenu. Deux conséquences en découlent. D'abord, un worker qui possède déjà ce hachage dans son cache n'a pas besoin qu'on lui renvoie le code. Ensuite, il ne peut pas exécuter autre chose que ce qui a été enregistré : le hachage est vérifié avant exécution.

Les dépendances suivent le même principe. Une fonction qui en appelle d'autres voit son graphe de dépendances résolu par hachage et matérialisé sur chaque worker avant l'appel racine — il n'y a donc jamais d'appel vers une définition absente ou divergente.

La persistance est assurée par SQLite en mode WAL, qui conserve les jobs, les workers, les baux et les événements. Un bail — une lease — est l'attribution temporaire d'un lot à un worker ; s'il expire sans résultat, le lot redevient disponible. Côté worker, un spool durable met le résultat en file avant envoi, ce qui permet de survivre à une indisponibilité momentanée du master.



Figure 40.2 — Le code circule par son empreinte, jamais par son nom

40.3. Enregistrer une fonction : la surface distribuable

[cluster:register] compile un keyword par le chemin de compilation canonique, audite le résultat, puis stocke l'IR de façon atomique et adressée par contenu. Le code n'est pas exécuté à l'enregistrement.

```
$fn = [cluster:register [keyword square `[* @1 @1]]]
# => "sha256:..."

# cluster:call exécute exactement le même objet compilé, mais localement
[cluster:call $fn 12]
# => 144
```

Quatre conditions sont exigées, et elles définissent ce qu'est une unité de travail distribuable. Le keyword doit être accepté par le compilateur ; il doit être fermé, sans capture de variable ni de contexte ; il ne doit contenir que des opérations pures et autorisées ; et son déterminisme comme sa compatibilité de plateforme doivent être vérifiables.

La surface autorisée recouvre l'essentiel du calcul général : les formes de contrôle do, if, when, switch, let et while ; l'itération par for, range, map, filter et take ; les listes, cons, unpack, mset et les objets purs ; les primitives déterministes de collections, de chaînes, le hachage SHA-256 et JSON ; l'arithmétique entière et les comparaisons. S'y ajoutent, depuis que le compilateur et le distributeur partagent un catalogue commun de capacités, [match], les instantanés Prolog, la logique miniKanren fermée du chapitre 38, les programmes objet fermés y compris leurs méthodes auditées, les constructeurs typés, le calcul flou, ainsi que les appels récursifs et les dépendances vers d'autres keywords compilés.

```
$sumDown = [cluster:register
  [keyword sum-down `[do
    $i = @1
    $total = 0
    [while [do
      $total = [+ $total $i]
      $i = [- $i 1]
      [> $i 0]]]
    $total]]]
```

Les refus sont tout aussi explicites, et ils dessinent en creux la définition de la pureté retenue. Sont rejetés : les entrées-sorties, l'horloge, l'aléatoire non seedé, les canaux et sémaphores, la persistance mutable, les captures de session, et les formes dont le résultat dépend d'une course — [first], [parallel], la liaison par course. Les fonctions flottantes non natives — sinus, cosinus, exponentielle, logarithme, tangente — restent exclues tant que la VM ne les implémente pas nativement. Les opérations flottantes admises sont liées à l'empreinte exacte de la plateforme ; racine carrée et puissance sont couvertes par des vecteurs de référence garantissant l'identité binaire des résultats d'un nœud à l'autre.

Ce dernier détail mérite qu'on s'y arrête. Distribuer un calcul sur des machines hétérogènes suppose que toutes rendent exactement le même résultat, au bit près. Un flottant calculé différemment selon le processeur suffirait à rendre un job non reproductible. D'où le descripteur de plateforme attaché à chaque worker et le refus, par le master, de confier une fonction à un worker dont l'ABI est incompatible.

40.4. Soumettre un calcul : shards, parallélisme et réducteurs

[cluster:map] crée un job persistant en appliquant la fonction à chaque élément d'une liste. Chaque appel reçoit l'élément courant en premier argument, suivi des valeurs de l'option args — l'arité enregistrée doit donc valoir un plus la longueur de args. Le retour immédiat est un handle JSON, le calcul se poursuivant en arrière-plan.

```
$job = [cluster:map $fn [1 2 3 4 5 6 7 8]
          [JSON "{\"parallelism\":4,\"shards\":8,\"reduce\":\"collect\"}"]]
# => {"job_id": "...", "status": "PENDING", "item_count": 8, "shard_count": 8}

[cluster:await $job 60]
[cluster:result $job]
# => [1 4 9 16 25 36 49 64]
```

Les options gouvernent le découpage et la robustesse. Le parallélisme demandé vaut par défaut le nombre de slots configurés. Le découpage s'exprime soit par shards, le nombre de lots, soit par shard_size, le nombre maximal d'items par lot — les deux stratégies s'excluent et ne doivent pas être fournies ensemble. S'y ajoutent la priorité, le nombre maximal de tentatives par tâche, fixé à trois par défaut, le délai d'une tâche à soixante secondes, celui du job à une heure, et des tags exigés sur le worker.

Sept réducteurs sont disponibles pour l'agrégation finale : collect rend la liste ordonnée de tous les résultats, sum leur somme, min et max les extrema, count le nombre de résultats, concat leur concaténation, et none n'en conserve aucun. Le choix n'est pas anodin sur de gros volumes : un réducteur scalaire évite de rapatrier l'ensemble des résultats.

Un détail d'implémentation explique un gain important obtenu en cours de route. Sur chaque worker, le manifeste immuable de la fonction n'est vérifié qu'une fois puis conservé dans un cache borné, et les appels appartenant au même shard sont composés en une fonction batch native envoyée à la VM en un seul échange. La version antérieure relisait le manifeste et faisait un aller-retour par item — la section 40.9 chiffre ce que ce changement a rapporté.

Le suivi se fait par [cluster:status], qui rend la progression persistante — identifiant, état, nombre d'items et de shards, shards terminés, progression entre zéro et un, compteurs de tâches, dates et erreur éventuelle. Un job passe par sept états : en attente, en cours, en réduction, en annulation, terminé, échoué ou annulé. [cluster:await] bloque jusqu'à un état terminal ; dépasser son délai interrompt l'attente mais n'annule pas le job. [cluster:result] lit le résultat, avec pagination possible si c'est une liste.

40.5. Quatorze builtins, quatre capabilities

L'API cliente compte quatorze builtins, chacun gardé par une capability du modèle de privilèges. Quatre capabilities suffisent à couvrir l'ensemble, et leur répartition dit qui peut faire quoi.

La capability cluster:code n'autorise qu'une opération, [cluster:register] — publier du code distribuable. La capability cluster:submit couvre [cluster:map] et [cluster:cancel] — lancer et arrêter un calcul. La capability cluster:read couvre la consultation : [cluster:status], [cluster:await], [cluster:result] et [cluster:jobs]. Enfin cluster:admin couvre l'exploitation : [cluster:workers], [cluster:drain], [cluster:metrics], [cluster:audit], [cluster:backup] et [cluster:gc]. Seul [cluster:call], qui exécute localement une fonction enregistrée à des fins de test, n'exige aucune capability particulière.

Cette granularité a une conséquence pratique : on peut donner à un utilisateur le droit de soumettre des calculs sans celui d'en publier le code, ou le droit de lire sans celui de soumettre. Les règles de sécurité complètent le dispositif — les commandes de jobs ne traversent jamais les espaces de travail, une capability manquante est refusée avant l'exécution du builtin, le rôle worker refuse les builtins clients, et les options inconnues sont rejetées plutôt qu'ignorées silencieusement.

Ce dernier point est une bonne pratique trop rare pour ne pas être relevée. Une option mal orthographiée qui serait ignorée produirait un job s'exécutant avec des paramètres autres que ceux voulus, sans le moindre signal. Le refus est ici la bonne réponse.

Une garantie de fond mérite d'être énoncée : les délais et les reprises ne rendent pas une fonction impure distribuable. La pureté et le déterminisme sont contrôlés à l'enregistrement, une fois pour toutes, et non compensés à l'exécution.

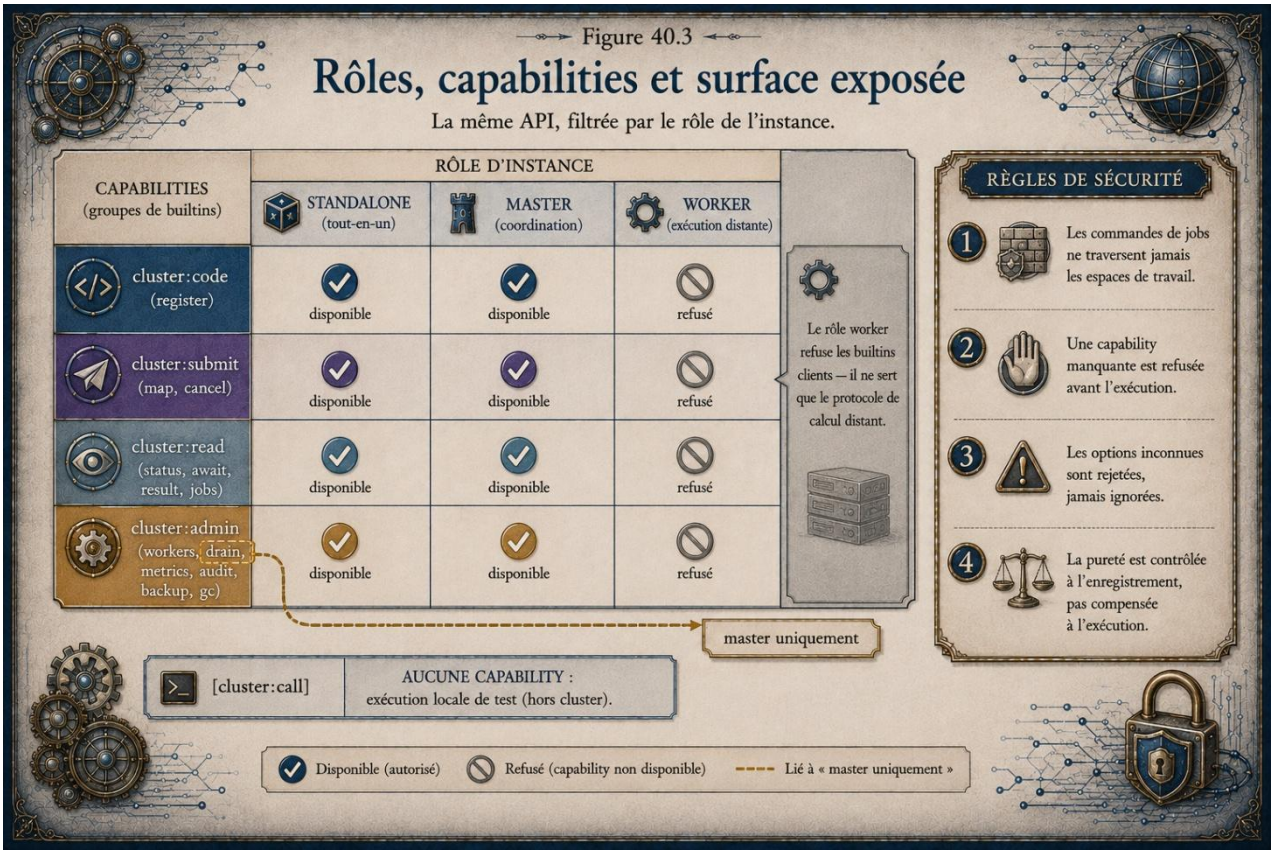


Figure 40.3 — Quatorze builtins, quatre capabilities, trois rôles d'instance

40.6. Déploiement, TLS et rotation de secrets

Chaque instance possède sa propre configuration, son répertoire de données ou de cache, et un secret propre à son identifiant de worker. Le master lie cryptographiquement chaque secret à un seul worker — il n'y a pas de secret partagé par défaut.

```
<CLUSTER>
  <ENABLED>true</ENABLED>
  <ROLE>master</ROLE>
  <DATA>data/cluster-master</DATA>
  <SLOTS>1</SLOTS>
  <VM_WORKERS>8</VM_WORKERS>
  <MASTER_SLOTS>0</MASTER_SLOTS>
  <WORKER_TOKENS>
    <WORKER id="compute-02">secret-compute-02</WORKER>
  </WORKER_TOKENS>
  <HEARTBEAT_SECONDS>5</HEARTBEAT_SECONDS>
  <OFFLINE_SECONDS>30</OFFLINE_SECONDS>
  <LEASE_SECONDS>90</LEASE_SECONDS>
</CLUSTER>
```

Deux réglages se confondent facilement et il faut les distinguer. SLOTS fixe le nombre de shards qu'une instance peut traiter simultanément ; VM_WORKERS fixe le nombre de processus VM Rust disponibles dans cette instance. Pour une charge à dominante calcul, il faut au moins autant de processus VM que de slots locaux, sans quoi les shards se disputent les mêmes processus. Mettre le master à zéro slot le réserve à l'ordonnancement.

Le master peut terminer TLS lui-même et exiger un certificat client, chaque worker vérifiant le master par l'autorité de certification et présentant son propre couple certificat-clé. La règle est classique mais rappelée : ne jamais partager une clé privée entre nœuds.

La rotation d'un secret se fait sans interruption, en trois temps. On déclare sur le master le nouveau secret en conservant l'ancien ; on met à jour le worker concerné et on le redémarre ; puis, après validation, on retire l'ancien. Pendant la fenêtre, les deux secrets sont comparés en temps constant — une précaution contre les attaques temporelles. Un mode à secret partagé subsiste pour compatibilité, mais il n'offre aucune isolation d'identité et le documentation le réserve aux clusters entièrement dignes de confiance.

40.7. Exploitation : métriques, audit, drain et sauvegarde

Cinq commandes couvrent l'exploitation courante, avec leurs équivalents REST pour l'outillage.

```
[cluster:workers]                                    # identité, plateforme, tags, état
[cluster:metrics]                                    # instantané opérationnel borné
[cluster:audit [JSON "{\"after_event_id\":0,\"limit\":100}"]]
[cluster:drain "compute-02" 1]                       # ne plus attribuer de bail
[cluster:backup]                                     # snapshot cohérent à chaud
[cluster:gc 3600]                                    # purge des objets CAS orphelins
```

[cluster:workers] rend les workers enregistrés avec leur descripteur de plateforme, leurs tags, leurs slots, leur dernier battement de cœur et leur état — en ligne, suspect, en drainage, hors ligne. Une instance standalone peut légitimement rendre une liste vide : ses slots locaux ne sont pas des workers distants enregistrés.

[cluster:metrics] rend un instantané borné et sans payload : comptes de jobs, tâches et workers par état, slots totaux et libres, nombre de tentatives et somme de leur temps de calcul, nombre et taille des objets stockés. [cluster:audit] parcourt le journal persistant des événements dans l'ordre, sans exposer les payloads de calcul, et sa reprise par identifiant d'événement permet un dépouillement incrémental.

Le drainage est l'outil des mises à jour sans interruption : un worker drainé ne reçoit plus de nouveaux baux mais termine ses tâches en cours. La procédure canari consiste à drainer un worker, le mettre à jour, le remettre en ligne, contrôler métriques et audit, puis poursuivre — le master étant mis à jour en dernier si le protocole reste compatible.

La sauvegarde crée à chaud un instantané atomique de la base cohérente, du code compilé immuable et des seuls blobs référencés, validé avant d'être annoncé complet. La restauration est volontairement hors ligne et refuse un répertoire non vide — un choix conservateur qui évite d'écraser des données actives par inadvertance. La documentation rappelle qu'un répertoire de sauvegarde local ne protège pas de la perte du disque.

40.8. Les pannes attendues

Un système distribué se juge à la façon dont il nomme ses modes de défaillance. Celui-ci en documente sept, chacun avec son traitement.

La perte d'un worker fait expirer son bail ; le shard redevient disponible et sera réattribué jusqu'au nombre maximal de tentatives. La perte du master n'entraîne pas la perte de l'état : SQLite en mode

WAL conserve jobs, workers, baux et événements, et un résultat mis en file chez un worker sera renvoyé au retour du master. Un résultat définitivement invalide part immédiatement en lettre morte, tandis qu'une panne transitoire est retentée avec un délai croissant avant d'y être déplacée à son tour.

Un token, un identifiant de démarrage ou un bail périmés donnent lieu à un refus explicite et le résultat est ignoré — une protection contre le worker zombie qui reviendrait livrer un calcul obsolète. Une incompatibilité d'ABI laisse le worker visible mais il ne reçoit pas la fonction, ce qui vaut mieux qu'un résultat calculé différemment. Une VM bloquée est tuée par le délai de tâche et le pool la recrée. Enfin, un quota de stockage atteint fait échouer les nouvelles écritures sans corrompre les objets existants.

La logique commune à ces sept traitements est le refus de la dégradation silencieuse. Dans chaque cas, le système préfère un échec visible et rattrapable à un résultat douteux — ce qui est, à l'échelle de l'infrastructure, la même position que celle défendue au chapitre 37 pour la preuve.

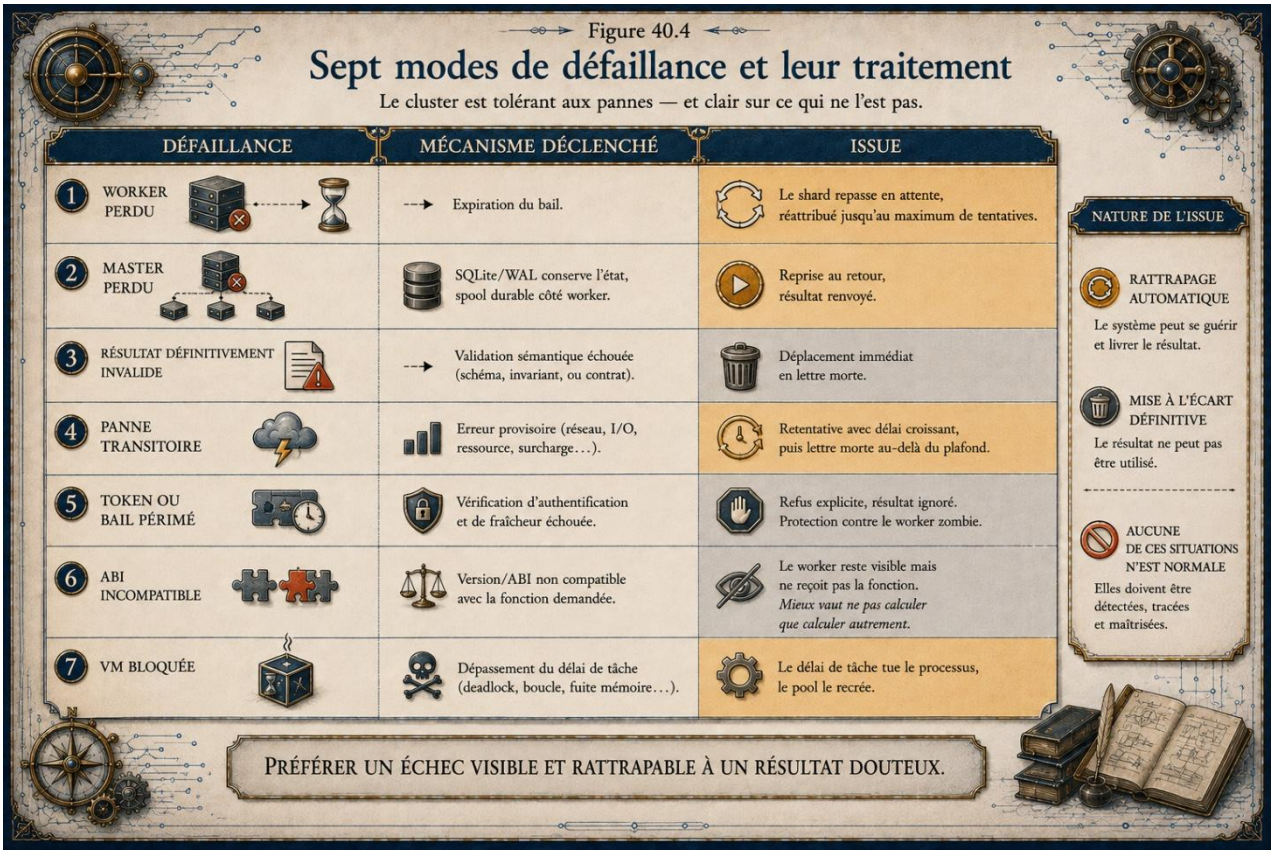


Figure 40.4 — Aucune défaillance ne dégrade silencieusement le résultat

40.9. Un cas réaliste : la tarification ontologique distribuée

L'exemple de référence traite un lot de devis. Pour chaque devis, une fonction VVL compilée instancie une ontologie, applique une chaîne de trente-neuf règles de tarification interdépendantes par calcul de point fixe — le moteur du chapitre 39 — puis ajoute la majoration obtenue au montant de base. Les devis étant indépendants, la charge se prête au découpage.

```
256 devis
   |
   +-- shard 0 -- worker 0 -- 32 devis x 39 règles
   +-- shard 1 -- worker 1 -- 32 devis x 39 règles
   +-- ...
   +-- shard 7 -- worker 7 -- 32 devis x 39 règles
                    |
                    +-- réducteur collect -- 256 prix dans l'ordre initial
```

Le protocole de mesure mérite d'être décrit, car il évite les pièges habituels des démonstrations de parallélisme. La fonction est enregistrée une seule fois par contenu ; un devis est validé par [cluster:call] avant toute soumission ; les processus sont chauffés ; puis exactement le même lot est soumis deux fois, avec un slot puis avec huit. Les cinq cent douze résultats des deux passages sont comparés aux montants attendus, un par un. Le speedup est donc mesuré sur un calcul validé et non sur un temps de réponse.

La première mesure, en juillet 2026 sur un ordinateur portable à douze processeurs logiques, donne un séquentiel de 15 776 millisecondes et un parallèle de 7 960, soit un facteur 1,98 pour huit slots — une efficacité de 24,8 %. Le rapport ne se contente pas du chiffre : il explique. La base SQLite confirme huit tentatives simultanées, donc le scheduler n'a pas sérialisé le travail. Mais un shard prend environ 1,9 seconde seul et 6,3 secondes quand huit shards sollicitent la machine ensemble. Le gain est réel et plafonné par la contention processeur et mémoire d'une seule machine.

Une optimisation intervint ensuite — cache de manifeste, appel natif par shard, lecture de tube sans thread par item — et le même lot fut rejoué. Les résultats sont sans ambiguïté : le séquentiel passe de 15 776 à 4 444 millisecondes, soit 3,55 fois ; le parallèle sur huit slots de 7 960 à 1 426, soit 5,58 fois ; le speedup interne monte de 1,98 à 3,12 ; le débit passe de 32,2 à 179,6 devis par seconde. Un second passage sur cinq cent douze devis donne 3,77 de speedup pour 180,2 devis par seconde, avec une occupation processeur de la VM à 32,3 % en moyenne et 53,3 % au pic.

La leçon est celle que tout praticien du calcul distribué finit par apprendre : avant de distribuer davantage, vérifier ce que coûte le transport à l'intérieur d'un nœud. Ici, le principal gain n'est pas venu d'un worker supplémentaire mais de la suppression d'un aller-retour par item au profit d'un appel groupé par shard.

L'exemple est représentatif pour six raisons que la documentation énumère, et qui constituent une bonne grille de lecture pour tout banc de mesure : le code distribué est du vrai VVL compilé et non une fonction Python injectée ; la charge emploie le moteur ontologique et son calcul de point fixe ; chaque item porte une donnée métier différente ; les résultats dépendent réellement des entrées et sont contrôlés un par un ; le passage à plusieurs slots est comparé au même ordonnanceur limité à un slot ; et la persistance, les shards, les réducteurs et les codecs sont inclus dans le temps observé.

Le modèle s'étend directement aux simulations de portefeuille, aux analyses documentaires précalculées et aux évaluations de règles métier — à la condition, posée dès la section 40.1, que chaque unité de travail reste pure, fermée et assez lourde pour amortir son transport.

40.10. Synthèse : distribuer sans renoncer au déterminisme

Ce chapitre clôt l'extension consacrée aux évolutions du runtime, et il en constitue l'aboutissement naturel. Le chapitre 37 a donné au langage de quoi vérifier ce qu'un modèle propose ; le 38, de quoi

engendrer des solutions par résolution logique ; le 39, de quoi exécuter tout cela nativement ; celui-ci, de quoi le faire à l'échelle de plusieurs machines.

La cohérence de l'ensemble tient à une exigence unique, déclinée à chaque niveau : le résultat ne doit pas dépendre du chemin qui l'a produit. Au chapitre 39, cela s'appelait la parité entre interpréteur et VM. Ici, cela s'appelle le déterminisme distribué, et le prix en est explicite — une fonction fermée et pure, un code désigné par son empreinte, une compatibilité d'ABI vérifiée avant toute attribution, des vecteurs de référence pour les flottants, et le refus catégorique de tout ce qui dépend d'une horloge, d'un aléa ou d'une course.

Ce prix est élevé et il exclut beaucoup de code utile. Mais il achète quelque chose de rare : un calcul distribué dont on peut rejouer un shard sur une autre machine et obtenir exactement le même résultat — ce qui, pour un système qui prétend à l'auditabilité de ses décisions, n'est pas une commodité mais une condition.

Reste à mentionner ce que ce chapitre ne couvre pas, puisque l'ouvrage s'est fait une règle de le dire. Le lien réseau réel entre instances, TLS inclus, n'a pas encore été mesuré ; toutes les mesures citées proviennent d'une machine unique en boucle locale. Le gate économique reste conditionnel. Et la contention observée à huit slots sur un seul ordinateur portable ne préjuge pas du comportement sur plusieurs serveurs physiques, où elle devrait en grande partie disparaître. Le système est fonctionnel et éprouvé sur un nœud ; sa validation multi-hôtes reste devant lui.

Conclusion générale

Quarante chapitres plus tard, il vaut la peine de revenir à la question posée en introduction : d'où viennent les règles formelles qui dirigent un système, et comment s'assurer qu'elles sont justes ?

Sur la première moitié de la question, l'ouvrage a exposé quatre réponses complètes. Un humain écrit les règles avec [ontology:rule], et l'on obtient l'intelligibilité maximale au prix de l'effort. Un LLM les synthétise depuis une description ou des exemples, et l'on gagne en vitesse ce que l'on perd en garantie. Une recherche logique les infère depuis des exemples positifs et négatifs, et le résultat est correct par construction. Un modèle bayésien apprend leurs poids plutôt que leur forme, et l'incertitude devient mesurable. Ces quatre voies ne se concurrencent pas : le chapitre 34 montre comment un même système les combine, et les chapitres 19 à 21 le montrent sur des cas réels.

Sur la seconde moitié — comment s'assurer qu'elles sont justes — la réponse s'est construite plus lentement, et c'est peut-être le fil le plus intéressant du livre. Le protocole d'atomes du chapitre 23 rend chaque valeur du runtime introspectable et filtrable. Les tests intégrés du chapitre 32 et les plugins signés du chapitre 33 verrouillent les frontières. La preuve formelle du chapitre 37 fournit enfin le juge qui manquait : un petit noyau auditable qui vérifie ce que le modèle propose, et qui préfère répondre « je ne sais pas » plutôt que de deviner.

Ce refus de deviner est, rétrospectivement, la discipline qui traverse tout l'ouvrage. Le cache sémantique du chapitre 3 enregistre le prompt et le modèle qui ont produit chaque valeur, faute de quoi la trace mentirait. Le moteur relationnel du chapitre 38 distingue explicitement complétude et efficacité. La machine virtuelle du chapitre 39 publie son catalogue de divergences et abandonne un gain mesuré de 13,8 fois pour ne pas rendre son déclenchement imprévisible. Le cluster du chapitre 40 documente le régime dans lequel il n'apporte rien. Aucun de ces sous-systèmes ne prétend à plus qu'il ne tient — et c'est précisément ce qui les rend utilisables ensemble.

Un livre sur un langage n'a pas vocation à conclure. Le langage continue, et ce qu'il deviendra dépend moins de ses concepteurs que de ceux qui l'emploieront sur des problèmes que personne n'a anticipés. Restent donc trois invitations, déjà formulées au chapitre 34 et qui gardent ici leur pleine valeur : prendre un problème réel et le reformuler en VVL ; choisir un paradigme d'écriture des règles sur lequel on n'a aucune expérience et mesurer ce qu'il coûte et ce qu'il rapporte ; et faire remonter ce que l'usage aura révélé, y compris et surtout ce qui n'a pas marché.

Si ce livre a donné, ne serait-ce qu'une fois, l'envie d'essayer un fragment de VVL sur un problème qui vous est propre, alors il a rempli sa fonction.

L'auteur vous souhaite beaucoup de plaisir.

Annexe A — Référence des builtins VVL

Mots-clés : référence, builtins, syntaxe, conventions, opérateurs, modules, ontologie, RAG, Prolog, agents, persistance, concurrence

Cette annexe constitue la **référence exhaustive** des fonctions intégrées (*builtins*) du langage VVL et du serveur VAILS. Elle est destinée aux développeurs qui ont assimilé les concepts présentés dans le corps du livre et qui cherchent un accès direct aux signatures et aux comportements de chaque commande.

L'écosystème compte environ **434 builtins** répartis en 51 modules thématiques. La structure de cette annexe suit cette organisation modulaire, avec une première partie consacrée aux **conventions du langage et aux exemples de syntaxe testés sur le serveur**, puis un inventaire par catégorie. Chaque entrée présente le nom du builtin, sa signature canonique, et une description concise du comportement attendu.

Trois conventions de notation sont utilisées tout au long de cette annexe :

- **`[name args...]`** : forme d'appel préfixe standard de VVL ; c'est la syntaxe canonique.

- **`a op b`** : forme infix acceptée par le parser pour les opérateurs (arithmétiques, comparaisons, opérateurs flous).

- ``[...] : littéral de liste *quasiquoté* qui empêche l'évaluation de son contenu ; `,expr` (unquote) à l'intérieur d'un quasiquote permet de réinjecter une valeur évaluée.

- **Marqueur `RAW`** : les arguments de l'expression sont passés *sans évaluation préalable* au builtin. Ce mode est utilisé par les primitives de contrôle de flux (`if`, `lambda`, `class`, `when`, etc.) qui doivent contrôler elles-mêmes l'évaluation de leurs sous-expressions.

A.1. Conventions fondamentales du langage

Avant de plonger dans l'inventaire des builtins, cette section rassemble les conventions syntaxiques de base que tout code VVL respecte. Ces règles sont stables et constituent la grammaire d'usage du langage.

Affectation et variables

L'affectation utilise la forme `$var = expr`. Elle retourne la valeur affectée. Une forme builtin équivalente est `[dim $var expr]`. Les variables non définies retournent une chaîne vide (`""`) — elles ne lèvent jamais d'exception. Cette tolérance simplifie les scripts mais demande de l'attention lors du débogage.

Booléens et égalité

VVL n'a pas de littéral `true` ni `false` : les booléens sont des entiers — `1` représente vrai, `0` représente faux. L'opérateur d'égalité est `==` ; le caractère `=` est exclusivement réservé à l'affectation. Cette distinction évite l'ambiguïté courante des langages C-like.

Quasiquote et unquote

Le littéral de liste de travail est le quasiquote `` : `[1 2 3] produit la liste [1 2 3] sans évaluer ses éléments comme des appels. À l'intérieur d'un quasiquote, l'unquote ,expr permet d'évaluer une expression et d'injecter son résultat. Exemple : $n = 5 puis `[1 ,$n ,[+ 1 2]] produit [1 5 3]`.

Interpolation de chaînes

L'interpolation de chaînes supporte les variables et les expressions entre `${...}`. Exemple : `$n = 3` puis `"n*n=${[* $n $n]}"` produit `"n*n=9"`. Cette fonctionnalité réduit considérablement le besoin de concaténation explicite.

Accesseurs pointés sur valeurs de base

Plusieurs accesseurs pointés sont disponibles sur les valeurs primitives : `.length`, `.upper`, `.lower`, `.trim`, `.int`, `.float`. Ils peuvent être chaînés. Exemple : `$s = " 42 "` puis `$s.trim.int` produit `42`.

A.2. Opérateurs arithmétiques, logiques et comparaisons

Les opérateurs arithmétiques et de comparaison acceptent à la fois la forme préfixe `[+ a b]` et la forme infix `a + b`. Le parser détecte automatiquement la forme utilisée.

```
[+ 1 2 3 4]        # => 10 (n-aire)
1 + 2              # => 3
[- 10 3]           # => 7
[* 2 3 4]          # => 24
[/ 10 2]           # => 5.0
[% 10 3]           # => 1 (modulo)
[pow 2 8]          # => 256.0
[sqrt 16]          # => 4.0
[min `[3 1 4 1 5]] # => 1
[max `[3 1 4 1 5]] # => 5
[== 1 1]           # => 1
1 == 2             # => 0
[!= 1 2]           # => 1
[< 1 2]            # => 1
[<= 5 5]           # => 1
[> 5 3]            # => 1
[>= 5 5]           # => 1
[&& 1 1]           # => 1
[&& 1 0]           # => 0
[|| 0 1]           # => 1
[not 0]            # => 1
[not 1]            # => 0
```

Précisions importantes issues du test serveur :

- Les formes textuelles `and`, `or`, `xor`, `abs` ne sont **pas** des builtins — il faut utiliser `&&`, `||`, etc.

- L'opérateur `**` n'est pas l'élévation à la puissance — c'est la **multiplication approximative** avec fallback scalaire. Pour la puissance, utiliser `[pow base exp]`.

- L'opérateur `~~` retourne `1.0` ou `0.0` (flottants), pas des booléens entiers.

- L'opérateur `++` concatène les chaînes en fallback scalaire : `"abc" ++ "def"` retourne `"abcdef"`.

- L'opérateur `//` effectue la division entière en fallback scalaire : `7 // 2` retourne `3`.

Opérateurs flous infix

La logique floue (chapitre 24) introduit quatre opérateurs supplémentaires : `~&` (T-norme AND), `~|` (T-conorme OR), `~!` (négation préfixe), `=>` (implication). Leur sémantique exacte dépend de la famille active (`zadeh`, `probabilistic`, `lukasiewicz`).

A.3. Contrôle de flux

Cette section regroupe les douze primitives de contrôle de flux. La plupart sont marquées `RAW` car elles doivent contrôler elles-mêmes l'évaluation de leurs branches.

Builtin	Mode	Signature	Description
delay		[delay ms]	Pause coopérative (vérifie le signal stop toutes les 100 ms).
do	RAW	[do expr1 expr2 ...]	Évaluation séquentielle ; retourne la dernière valeur.
exit		[exit val?]	Lève le signal stop dans le contexte courant.
first	RAW	[first expr1 expr2 ...]	Course parallèle ; retourne le résultat du premier thread terminé.
if	RAW	[if cond then else?]	Conditionnel ; retourne la branche then si vrai, sinon else.
silent		[silent expr1 expr2 ...]	Exécute mais retourne toujours une chaîne vide.
switch	RAW	[switch val case1 body1 ... _ default?]	Match valeur contre cas ; `_` matche n'importe quoi.
timeout	RAW	[timeout ms expr]	Limite de temps ; lève StopBranchException, capturable par `~>`.
try	RAW	[try try-body catch-body]	Sur exception : bind `$__error` puis exécute catch-body.
when	RAW	[when cond body]	Exécute body uniquement si cond est vrai.
while	RAW	[while body]	Boucle tant que body est vrai. Plafond strict de 100 000 itérations.
not		[not val]	Retourne 1 si val est faux/vide, sinon 0.

***Figure A.1 —** Builtins de contrôle de flux*

```
[if [> 5 3] "yes"]                          # => "yes"
[if [< 5 3] "yes" "no"]                     # => "no"
[when [> 5 3] "go"]                         # => "go"
[do 1 2 3]                                  # => 3
1 >> 2 >> 3                                 # => 3 (chaînage seq.)
[try [/ 1 0] "fallback"]                    # => "fallback"
[switch 2 1 "one" 2 "two" 3 "three"]        # => "two"
[switch 99 1 "one" 2 "two" "default"]       # => "default"
[match 5 5 "five" "other"]                  # => "five"
[match 42 ?z $__z]                          # => 42
[match `[10 20] `[?a ?b] [list $__a $__b]]  # => [10 20]
```

`[match ...]` capture les patterns `?name` dans `$__name`. L'interpolation de chaînes peut lire ces captures par `${__name}`. `[first expr1 expr2 ...]` est une **course parallèle**, pas un court-circuit booléen — avec des branches qui se terminent instantanément, le gagnant peut varier.

A.4. Fonctions d'ordre supérieur

Builtin	Mode	Signature	Description
filter		[filter lambda list]	Filtre la liste selon un prédicat. Supporte VVL_List et VVL_Json.
flat		[flat list depth?]	Aplatit une liste imbriquée. Le paramètre depth limite la profondeur.
for	RAW	[for $var list body]	Boucle sur liste/array JSON. Retourne la liste des résultats.
map		[map lambda list]	Applique la fonction à chaque élément.
pmap		[pmap lambda list]	Applique la fonction en parallèle (concurrence native).

Figure A.2 — *Builtins d'ordre supérieur*

```
[[lambda [] 42]]                                  # => 42
[[lambda [$x] [* $x 2]] 5]                        # => 10
[[lambda [$a $b] [+ $a $b]] 3 4]                  # => 7
$dbl = [lambda [$x] [* $x 2]]
[$dbl 7]                                          # => 14
[map [lambda [$x] [* $x 2]] `[1 2 3]]             # => [2 4 6]
[filter [lambda [$x] [> $x 2]] `[1 2 3 4]]        # => [3 4]
[for $i `[1 2 3] [* $i 10]]                       # => [10 20 30]
[pmap [lambda [$x] [* $x $x]] `[1 2 3 4]]         # => [1 4 9 16]
```

Les paramètres canoniques de lambda incluent le `$` dans la liste de paramètres : `[lambda [$x] ... $x ...]`. La forme alternative `[lambda [x] ... $__x ...]` fonctionne aussi, mais mélanger `[x]` avec `$x` échoue silencieusement car `$x` est alors une variable distincte du paramètre formel `x`.

A.5. Listes

Les indices de liste sont **basés sur 1** pour `getValue`, `insert`, `remove`, `put` et `update`. La fonction `[exist value list]` est strictement typée.

Builtin	Mode	Signature	Description
append		[append head|tail val list]	Ajoute une valeur en tête ou en queue.
concat		[concat l1 l2 l3...]	Aplatit d'un niveau — concatène les listes données.
dim	RAW	[dim $var value]	Affecte une valeur à une variable (alternative à `$var = value`).
exist		[exist value list]	Retourne 1 si la valeur est dans la liste, sinon 0.
getType	RAW	[getType element]	Tag de type VVL : "NUM", "STRING", "LIST", "JSON", ...
getValue		[getValue idx list]	Élément à l'index 1-basé.
insert		[insert idx value list]	Insère une valeur à l'index 1-basé.
length		[length x]	Longueur d'une liste ou d'une chaîne.
list		[list items...]	Constructeur variadique de VVL_List.

Builtin	Mode	Signature	Description
listMatch		[listMatch list default lambda_cons]	Si vide : default ; sinon `lambda_cons(head, tail)`.
match	RAW	[match target pat1 body1 ... default?]	Pattern matching avec exécution différée du corps.
mset	RAW	[mset [$vars...] $values]	Affectation de destructuration multi-variables.
pop		[pop head|tail list]	Retourne la tête ou la queue de la liste.
put		[put type idx value list]	Insère/remplace à l'index avec typage (NUM/STRING/LIST/JSON).
remove		[remove idx list]	Supprime l'élément à l'index 1-basé.
slice		[slice start stop step? list]	Tranche 1-basée, stop inclus. Step négatif inverse.
unpack		[unpack $list $headVar $tailVar]	Destructure liste en tête/queue. 1 si succès, 0 si vide.
update		[update type idx value list]	Met à jour la valeur à l'index avec typage.
with	RAW	[with $source $alias expr1 ...]	Évaluation parallèle, chaque expression avec `$alias = source`.

Figure A.3 — *Builtins de manipulation de listes*

```
[list 1 2 3]                        # => [1 2 3]
[length `[1 2 3 4]]                 # => 4
[getValue 1 `[10 20 30]]            # => 10
[exist 2 `[1 2 3]]                  # => 1
[concat `[1 2] `[3 4]]              # => [1 2 3 4]
[append "head" 0 `[1 2 3]]          # => [0 1 2 3]
[append "tail" 99 `[1 2 3]]         # => [1 2 3 99]
[insert 1 99 `[1 2 3]]              # => [99 1 2 3]
[remove 2 `[10 20 30 40]]           # => [10 30 40]
[pop "head" `[1 2 3]]               # => 1
[pop "tail" `[1 2 3]]               # => 3
[put "NUM" 2 99 `[1 2 3]]           # => [1 99 3]
[sort `[3 1 4 1 5]]                 # => [1 1 3 4 5]
[reverse `[1 2 3]]                  # => [3 2 1]
[range 0 5 1]                        # => [0 1 2 3 4]
[getType `[1 2 3]]                  # => "LIST"
```

A.6. Chaînes de caractères

Les positions des fonctions `mid`, `instr`, `instrrev`, `instrall` sont **1-basées** ; le retour `0` signifie « non trouvé ». La fonction `replace` utilise la sémantique des regex Python.

Builtin	Signature	Description
contains	[contains str sub]	Retourne 1 si str contient sub, sinon 0.
endsWith	[endsWith str suffix]	Retourne 1 si str se termine par suffix.
instr	[instr needle haystack]	Position 1-basée de needle dans haystack ; 0 si absent.
instrall	[instrall needle haystack]	Liste des positions 1-basées de toutes les occurrences.
instrrev	[instrrev needle haystack]	Position 1-basée de la dernière occurrence.
join	[join list sep]	Joint les éléments avec le séparateur.

Builtin	Signature	Description
mid	[mid start length str]	Sous-chaîne ; longueur 0 = jusqu'à la fin.
replace	[replace pattern repl str]	Remplacement par regex (sémantique Python).
split	[split sep str]	Découpe en liste selon le séparateur.
startsWith	[startsWith str prefix]	Retourne 1 si str commence par prefix.
strToLowerCase	[strToLowerCase str]	Convertit en minuscules.
strToUpperCase	[strToUpperCase str]	Convertit en majuscules.

Figure A.4 — *Builtins de manipulation des chaînes*

```
[length "hello"]                        # => 5
[contains "hello world" "world"]        # => 1
[startsWith "hello" "he"]               # => 1
[endsWith "hello" "lo"]                 # => 1
[split "," "a,b,c"]                     # => ["a" "b" "c"]
[join `["a" "b" "c"] "-"]              # => "a-b-c"
[replace "l" "L" "hello"]               # => "heLLo"
[replace "[aeiou]" "*" "hello"]         # => "h*ll*"
[mid 1 3 "hello"]                       # => "hel"
[mid 1 0 "hello"]                       # => "hello"
[instr "world" "hello world"]           # => 7
[instrrev "b" "abcabc"]                 # => 5
[instrall "a" "ababab"]                 # => [1 3 5]
```

A.7. Conversions de types et fonctions mathématiques

Builtin	Signature	Description
BOOL	[BOOL x]	Conversion booléenne (chaîne vide / 0 = faux ; sinon vrai).
FLOAT	[FLOAT x]	Conversion en flottant.
INT	[INT x]	Conversion en entier (troncature pour les flottants).
STRING	[STRING x]	Conversion en chaîne (sérialise listes et JSON).
chr	[chr code]	Caractère depuis code ASCII.
convert	[convert type val]	Conversion générique : "int", "float", "string", "bool".
guid	[guid]	UUID v4 sous forme de chaîne.
max	[max list]	Maximum d'une liste.
min	[min list]	Minimum d'une liste.
ord	[ord char]	Code ASCII du caractère.
pow	[pow base exp]	Élévation à la puissance.
sqrt	[sqrt x]	Racine carrée.
timestamp	[timestamp]	Millisecondes depuis l'epoch (pas secondes).

Figure A.5 — *Builtins de conversion et de math*

La fonction `BOOL` suit la véracité du runtime : chaîne vide, liste vide et zéro numérique sont faux ; toutes les autres valeurs non vides sont vraies. La fonction `timestamp` retourne des **millisecondes**, pas des secondes.

A.8. JSON et XML

Builtin	Signature	Description
JSON	[JSON str]	Parse une chaîne JSON en nœud VVL_Json.
XML	[XML str]	Parse une chaîne XML en nœud VVL_XML.
getXMLValue	[getXMLValue $x tag]	Valeur textuelle d'une balise.
json-decode	[json-decode str]	Décode une chaîne JSON littérale.
json-encode	[json-encode val]	Encode une valeur en chaîne JSON.
json-get-opt	[json-get-opt key json default]	Get avec valeur par défaut si absent.
json-keys	[json-keys $j]	Liste des clés du JSON.
json-path-get	[json-path-get json "x.y"]	Get avec chemin pointé ; "null" si absent.
json-path-set	[json-path-set json "x.y" v]	Set avec chemin pointé ; retourne JSON modifié.
json-set	[json-set json key val]	Modifie ou ajoute une clé.
jsonGetValue	[jsonGetValue key $j]	Get sur clé directe ; alias historique.
jsonToList	[jsonToList $j]	Convertit un array JSON en VVL_List.
setXMLValue	[setXMLValue val $x tag]	Modifie la valeur d'une balise.

Figure A.6 — *Builtins JSON et XML*

A.9. Système de fichiers (jail-protégé)

Toutes les opérations de système de fichiers sont **protégées par jail** : les chemins comme `"../../etc/passwd"` lèvent `PermissionError`. Les opérations sont confinées au workspace actif. Après suppression d'un fichier, `[file-read path]` retourne `""` plutôt que de lever une exception.

Builtin	Signature	Description
file-delete	[file-delete path]	Supprime un fichier ; 1 si succès, 0 si absent.
file-read	[file-read path]	Lit un fichier texte ; chaîne vide si absent.
file-read-b64	[file-read-b64 path]	Lit un fichier binaire en base64.
file-rename	[file-rename old new]	Renomme un fichier ; 1 si succès.
file-write	[file-write path content]	Écrit du texte dans un fichier ; 1 si succès.
file-write-b64	[file-write-b64 path b64]	Écrit un fichier binaire depuis base64.
ls	[ls path?]	Liste JSON du contenu d'un répertoire.

Figure A.7 — *Builtins du système de fichiers*

A.10. Utilitaires PDF et images

Builtin	Signature	Description
PDF	[PDF path]	Charge un objet PDF depuis un fichier.
pdfText	[pdfText pdf|path]	Extrait tout le texte du PDF.
pdfPages	[pdfPages pdf]	Nombre de pages du PDF.
pdfPage	[pdfPage pdf n]	Texte d'une page spécifique.
pdfToImages	[pdfToImages pdf zoom?]	Convertit chaque page en image. Zoom 2.0 recommandé.
imageInfo	[imageInfo path]	Métadonnées de l'image (dimensions, format).
imageResize	[imageResize path WxH out]	Redimensionne et écrit l'image.

Figure A.8 — *Builtins PDF et images*

A.11. Client et serveur HTTP

Builtin	Signature	Description
fetch	[fetch url options?]	Requête HTTP générique (GET/POST/PUT/DELETE).
scrape	[scrape url]	Récupère le texte d'une page web (HTTP simple).
scrapeLinks	[scrapeLinks url]	Extrait tous les liens d'une page.
serveHttp	[serveHttp port handler]	Démarre un serveur HTTP avec gestionnaire VVL.
stopHttp	[stopHttp port]	Arrête le serveur HTTP sur le port.

Figure A.9 — *Builtins HTTP*

A.12. Programmation orientée objet

Le système orienté objet de VVL (chapitre 6) repose sur huit primitives qui permettent la définition de classes, mixins et protocoles, ainsi que la création et la manipulation d'instances.

Builtin	Mode	Signature	Description
class	RAW	[class Name [field f v] [method m [...] body] ...]	Définit une classe avec champs et méthodes.
mixin	RAW	[mixin Name [field ...] [method ...]]	Définit un mixin réutilisable.
protocol	RAW	[protocol Name [requires methods...]]	Définit un protocole (interface).
new		[new ClassName args?]	Crée une instance.
extends	RAW	[extends Parent ...]	Marque l'héritage dans une définition de classe.
implements	RAW	[implements Protocol ...]	Marque la conformité à un protocole.
mixes-in	RAW	[mixes-in MixinName]	Inclut un mixin dans une classe.
call		[call obj method args...]	Appel méthode dynamique sur un objet.

Figure A.10 — *Builtins du système objet*

```
[class Point [field x 0] [field y 0]]
$p = [new Point]
$p.x                                # => 0
$p.x = 10
$p.x                                # => 10

[class Counter
    [field n 0]
    [method incr [] [do $self.n = ($self.n + 1) $self.n]]
]
$c = [new Counter]
[$c.incr]                           # => 1
[$c.incr]                           # => 2
```

A.13. Fibers (coroutines coopératives)

Builtin	Signature	Description
fiber	[fiber [lambda [] body]]	Crée une fiber prête à être démarrée.
fiber-resume	[fiber-resume $f val?]	Reprend la fiber suspendue ; passe optionnellement une valeur.
fiber-status	[fiber-status $f]	Statut : "suspended", "running", "done".
fiber-yield	[fiber-yield val?]	Suspend la fiber courante en retournant val.
fiber-current	[fiber-current]	Référence à la fiber en cours d'exécution.
fiber-done	[fiber-done $f]	Retourne 1 si la fiber est terminée.

Figure A.11 — *Builtins des fibers (coroutines coopératives)*

A.14. Asynchronisme, promesses, channels et sémaphores

La concurrence native de VVL combine plusieurs primitives complémentaires : promesses pour la composition asynchrone, channels pour la communication producteur-consommateur, sémaphores pour le contrôle de flux, mutex pour l'exclusion mutuelle. Toutes sont coopératives — elles s'intègrent dans la boucle d'événements interne sans création de threads OS.

Builtin	Signature	Description
promise	[promise [lambda [] body]]	Crée une promesse qui exécute le body en arrière-plan.
promise-await	[promise-await $p timeout?]	Attend la résolution ; lève si timeout.
promise-all	[promise-all promises]	Attend toutes les promesses ; retourne la liste des résultats.
promise-race	[promise-race promises]	Retourne le résultat de la première résolue.
promise-status	[promise-status $p]	"pending", "fulfilled", "rejected".
channel	[channel buffer?]	Crée un channel ; buffer optionnel.
channel-send	[channel-send $ch val]	Envoie sur le channel (bloque si plein).
channel-recv	[channel-recv $ch]	Reçoit du channel (bloque si vide).

Builtin	Signature	Description
channel-close	[channel-close $ch]	Ferme le channel.
channel-tryrecv	[channel-tryrecv $ch]	Reçoit sans bloquer ; "" si vide.
semaphore	[semaphore permits]	Crée un sémaphore avec N permits.
semaphore-acquire	[semaphore-acquire $s]	Acquiert un permit (bloque si épuisé).
semaphore-release	[semaphore-release $s]	Libère un permit.
mutex	[mutex]	Crée un mutex (sémaphore à 1 permit).
mutex-lock	[mutex-lock $m]	Verrouille le mutex.
mutex-unlock	[mutex-unlock $m]	Déverrouille le mutex.
async	[async [lambda [] body]]	Variante de promise ; alias historique.

Figure A.12 — *Builtins de concurrence*

A.15. Administration : utilisateurs, workspaces, sessions

Les commandes d'administration sont jail-protégées : elles requièrent une authentification administrateur active (`[proadmin login pass]`). Les workspaces isolent les fichiers, ontologies et caches d'un projet.

Builtin	Signature	Description
proadmin	[proadmin login pass]	Authentification administrateur.
proadmin-logout	[proadmin-logout]	Termine la session admin.
users	[users action ...]	Gestion des utilisateurs (list, create, delete).
workspaces	[workspaces]	Liste les workspaces disponibles.
workspace	[workspace go|create|delete name]	Active, crée ou supprime un workspace.
workspace-current	[workspace-current]	Nom du workspace actif.
session	[session id?]	Information de session courante.
sessions	[sessions]	Liste les sessions actives.

Figure A.13 — *Builtins d'administration*

A.16. Logs

Builtin	Signature	Description
log	[log msg1 msg2 ...]	Concatène et trace ; retourne le message concaténé.
log	[log val]	Imprime sans retour ; retourne chaîne vide.
logTable	[logTable list]	Affiche une liste sous forme de tableau ; 1 si succès.
log-tail	[log-tail file lines?]	Dernières lignes d'un fichier de log ; admin-only.

Figure A.14 — *Builtins de logs*

A.17. UI : prompts, dialogues, alertes

Ces builtins produisent des éléments d'interface dans le client REPL ou la console interactive. En mode automatisé (script), ils retournent typiquement la valeur sans afficher de dialogue.

Builtin	Signature	Description
md	[md "markdown"]	Affiche un bloc markdown rendu.
html	[html "<...>"]	Affiche du HTML brut dans le client.
dialog	[dialog title body]	Dialogue modal avec titre et corps.
alert	[alert msg]	Alerte simple.
confirm	[confirm msg]	Confirmation OK/Cancel ; retourne 1 ou 0.
prompt	[prompt msg default?]	Saisie de texte ; retourne la chaîne.
choice	[choice msg options]	Choix dans une liste.
progress	[progress current total]	Barre de progression.
table	[table rows]	Tableau formaté.
code	[code lang src]	Bloc de code coloré.
link	[link url label?]	Lien cliquable.
image	[image url alt?]	Image dans le client.
notify	[notify msg level?]	Notification système.
speak	[speak text]	Synthèse vocale (si configurée).

Figure A.15 — *Builtins UI*

A.18. Méta : aide, introspection, évaluation

Builtin	Signature	Description
help	[help "name"]	Docstring du builtin enregistré.
inspectWord	[inspectWord "name"]	"builtin", "keyword" ou "word".
evaluate	[evaluate src_str]	Parse et évalue une chaîne source.
execute	[execute val]	Réévalue récursivement les variables/listes ; chaînes inchangées.
keyword	[keyword name body]	Définit un keyword utilisateur.
undefkeyword	[undefkeyword name]	Supprime un keyword.
keywords	[keywords]	Liste des keywords définis.
vvl-version	[vvl-version]	Version du runtime VVL.
quote	[quote x]	Empêche l'évaluation ; renvoie l'expression telle quelle.
deparse	[deparse x]	Représentation textuelle d'une expression.
symbols	[symbols]	Liste des symboles dans le contexte courant.
env	[env]	Snapshot du contexte courant (variables locales).

Figure A.16 — *Builtins méta et introspection*

Distinction importante : `[evaluate src]` parse une **chaîne** source et l'évalue ; `[execute val]` parcourt récursivement les **variables et listes** existantes pour les réévaluer — les chaînes restent inchangées par `execute`.

A.19. Utilitaires divers

Builtin	Signature	Description
random	[random min max]	Entier aléatoire dans [min, max].
randomf	[randomf]	Flottant aléatoire dans [0, 1).
hash	[hash str algo?]	Hash MD5/SHA1/SHA256 d'une chaîne.
sleep	[sleep ms]	Pause synchrone (alias de delay).

Figure A.17 — *Utilitaires divers*

A.20. Logique floue et bootstrap de système expert

Ce module est l'objet du chapitre 24. Il combine les opérateurs flous fondamentaux, les hedges linguistiques, et les commandes haut niveau qui compilent une description NL en système expert opérationnel.

Opérateurs flous et familles

Builtin	Signature	Description
fuzzy:and	[fuzzy:and a b ... :family n?]	T-norme AND n-aire ; family override.
fuzzy:or	[fuzzy:or a b ... :family n?]	T-conorme OR n-aire ; family override.
fuzzy:not	[fuzzy:not x]	Négation : 1 − x.
fuzzy:imp	[fuzzy:imp a b]	Implication floue selon famille active (Gödel par défaut).
fuzzy:family	[fuzzy:family name?]	Lit/définit la famille T-norme : "zadeh", "probabilistic", "lukasiewicz".

Figure A.18 — *Opérateurs flous et familles T-normes*

Hedges linguistiques et agrégateurs

Builtin	Signature	Description
fuzzy:very	[fuzzy:very x]	x^2 (concentration).
fuzzy:extremely	[fuzzy:extremely x]	x^4 (super-concentration).
fuzzy:somewhat	[fuzzy:somewhat x]	$\sqrt{x}$ (dilation).
fuzzy:slightly	[fuzzy:slightly x]	x^(1/3) (super-dilation).
fuzzy:wmean	[fuzzy:wmean values weights]	Moyenne arithmétique pondérée.
fuzzy:most	[fuzzy:most values]	Moyenne du top-half (« la plupart »).
fuzzy:few	[fuzzy:few values]	Moyenne du bottom-half (« peu »).

Figure A.19 — *Hedges et agrégateurs flous*

Système expert : ontologie et règles depuis NL

Builtin	Signature	Description
fuzzy:ontology	[fuzzy:ontology "description NL"]	NL → ontology:define complet (classes, abstracts, relations).
fuzzy:rule	[fuzzy:rule "règle NL" $schema?]	NL → ontology:rule formelle avec opérateurs flous infix.
expert	[expert "domaine + règles" "Name"]	Bootstrap one-shot : ontology + règles + keyword <Name>.
expert:save	[expert:save "Name"]	Persiste l'expert dans le workspace.
expert:load	[expert:load "Name"]	Restaure schéma, abstracts, règles et keyword.
expert:list	[expert:list]	Liste JSON des experts sauvegardés.

Figure A.20 — *Bootstrap de système expert*

A.21. Commandes IA : sémantique, agent, architect

Ces six commandes constituent le cœur de la délégation au LLM. Elles sont présentées en détail aux chapitres 7 (sémantique), 8 (opérateur `@`), et 22 (agents).

Builtin	Signature	Description
semantic	[§ x] ou [semantic x]	VVL_Semantic pour texte ; vision LLM si image ; DALL-E 3 si format IMG.
agent	[agent "goal" options?]	Agent autonome ReAct multi-étapes.
architect	[architect "goal"]	Orchestrateur single-shot multi-agents.
semanticCache	[semanticCache action ...]	Gestion du cache : reset, delete, purge, list.
semanticMode	[semanticMode mode]	Mode de génération : "normal" ou "advanced".
semanticModel	[semanticModel role? name?]	Inspecte/définit les modèles vision et image.

Figure A.21 — *Commandes IA principales*

Les options de `[agent]` sont passées par dictionnaire JSON : `budget` (max d'itérations, défaut 8), `tools` (liste des builtins autorisés, sandbox), `model` (override du modèle). Les images générées par `[§ desc IMG "WxH"]` sont sauvegardées sous le workspace utilisateur actif.

A.22. RAG (Retrieval-Augmented Generation)

Le module RAG (chapitre 14) implémente la recherche sémantique et la génération augmentée. Quatre modes de recherche cohabitent : `hybrid` (défaut, équilibre RAG vectoriel + graphe), `naive` (RAG vectoriel pur), `local` (focus document), `global` (synthèse corpus).

Indexation et insertion

Builtin	Signature	Description
ragInsert	[ragInsert kb text]	Insère du texte ; retourne "OK".
ragInsertId	[ragInsertId kb text]	Insère et retourne l'ID (recommandé).

Builtin	Signature	Description
ragInsertBatch	[ragInsertBatch kb texts]	Insère plusieurs textes en lot (plus rapide).
ragUpdate	[ragUpdate kb text]	Met à jour (delete + insert).
ragUpdateById	[ragUpdateById kb docId text]	Met à jour par ID.
ragRemove	[ragRemove kb text]	Supprime par contenu (texte exact ; préférer RemoveById).
ragRemoveById	[ragRemoveById kb docId]	Supprime par ID (recommandé).
ragRemoveEntity	[ragRemoveEntity kb entity]	Supprime une entité du graphe.
ragRemoveRelation	[ragRemoveRelation kb src tgt]	Supprime une relation entre deux entités.

Figure A.22 — *Indexation et CRUD du RAG*

Requêtage

Builtin	Signature	Description
ragAsk	[ragAsk kb question mode?]	Recherche sémantique (modes hybrid/naive/local/global).
ragAskSimple	[ragAskSimple kb question top_k?]	Recherche simple sans LLM (top_k=5).
ragAskAdvanced	[ragAskAdvanced kb q top_k? mode?]	Recherche avancée avec contrôle fin.
ragSearchContext	[ragSearchContext kb query mode?]	Récupère uniquement le contexte (sans génération LLM).
ragSearchDocs	[ragSearchDocs kb query]	Recherche par mots-clés dans les documents (sans LLM).
ragExtractJson	[ragExtractJson kb question schema?]	Extraction structurée en JSON.

Figure A.23 — *Requêtage RAG*

Gestion des KB et inspection

Builtin	Signature	Description
ragList	[ragList]	Liste les KB RAG du contexte actuel.
ragInfo	[ragInfo kb]	Métadonnées sur la KB (taille, fichiers, cache).
ragDocs	[ragDocs kb]	Liste tous les documents (id, statut, taille).
ragDocContent	[ragDocContent kb docId]	Contenu d'un document.
ragClear	[ragClear kb]	Vide la KB sans la détruire.
ragClose	[ragClose kb]	Ferme la KB (libère le cache mémoire).
ragDelete	[ragDelete kb]	Supprime la KB (irréversible).
ragCacheClear	[ragCacheClear]	Vide le cache RAG en mémoire.
ragCacheInfo	[ragCacheInfo]	Info cache (instances chargées, mémoire).

Figure A.24 — *Gestion des KB RAG*

A.23. Base de connaissances Prolog

La KB Prolog (chapitre 11) est citoyenne de première classe. Les huit primitives ci-dessous permettent de la manipuler directement, en complément du système ontologique qui s'appuie dessus.

Builtin	Signature	Description
kb	[kb set|reset name]	Initialise ou efface une base nommée.
facts	[facts name "fait"...]	Ajoute des faits à la KB.
rules	[rules name "règle"...]	Ajoute des règles à la KB.
query	[query JSON|VVL name query...]	Exécute une requête ; retour JSON ou liste VVL.
prologSolutions	[prologSolutions name query]	Liste des solutions (chacune = liste de paires [var, valeur]).
prologValues	[prologValues name query var]	Liste des valeurs liées à var dans toutes les solutions.
prologFirstValue	[prologFirstValue name query var def]	Première valeur, ou défaut.
prologExists	[prologExists name query]	1 si au moins une solution, sinon 0.

Figure A.25 — *Builtins Prolog*

```
[kb set famille]
[facts famille
    "parent(alice, bob)"
    "parent(bob, charlie)"]
[rules famille
    "ancetre(X, Y) :- parent(X, Y)."
    "ancetre(X, Y) :- parent(X, Z), ancetre(Z, Y)."]

[prologExists famille "ancetre(alice, charlie)"]   # => 1
[prologValues famille "ancetre(alice, X)" "X"]     # => ["bob" "charlie"]
```

A.24. Système ontologique

Le système ontologique (chapitre 12) compose 34 builtins qui forment l'épine dorsale du raisonnement neuro-symbolique. Les commandes sont organisées en groupes fonctionnels.

Définition de schémas et classes

Builtin	Mode	Signature	Description
ontology:define	RAW	[ontology:define name définitions...]	Définit classes, attributs, abstracts, relations.
ontology:concept-add		[ontology:concept-add name attrs?]	Ajoute un concept à l'ontologie active.
ontology:abstract	RAW	[ontology:abstract Name : Class encode]	Marque concept abstrait + encodages Zalta.
ontology:reset		[ontology:reset]	Efface tous les faits.
ontology:context		[ontology:context name?]	Bascule ou retourne le contexte ontologique courant.

Figure A.26 — *Définition de schémas*

Extraction et instances

Builtin	Signature	Description
ontology:fromText	[ontology:fromText onto class text debug?]	Extraction LLM enum-aware d'une instance depuis du texte.
ontology:fromImage	[ontology:fromImage image]	Extraction par vision LLM depuis une image.
ontology:comprehend	[ontology:comprehend text]	Extraction libre concepts/relations.
ontology:identify	[ontology:identify entity]	Identifie le concept le plus proche d'une description.
ontology:commit	[ontology:commit instance]	Inscrit l'instance dans la KB Prolog (instance/2 + prop/3 + rel/3).
ontology:set	[ontology:set instance attr value]	Modifie un attribut d'une instance.
ontology:clearInstances	[ontology:clearInstances concept?]	Supprime toutes les instances d'un concept (ou toutes).

Figure A.27 — *Extraction et manipulation d'instances*

Inférence et règles

Builtin	Mode	Signature	Description
ontology:rule	RAW	[ontology:rule name body]	Définit une règle d'inférence (target/priority/when/then).
ontology:infer		[ontology:infer query]	Une seule passe d'inférence.
ontology:infer-fixpoint		[ontology:infer-fixpoint max?]	Boucle jusqu'à point fixe ou max itérations.
ontology:satisfies		[ontology:satisfies inst concept]	Score de satisfaction (alias `~~`).
ontology:learn		[ontology:learn examples target?]	Induit des règles depuis des exemples.

Figure A.28 — *Inférence et apprentissage de règles*

Mondes possibles (Zalta)

Builtin	Mode	Signature	Description
ontology:world		[ontology:world name]	Bascule vers un monde possible.
ontology:in-world	RAW	[ontology:in-world name body]	Évalue body dans un monde puis restaure.

Figure A.29 — *Mondes possibles*

Requêtes et exports

Builtin	Mode	Signature	Description
ontology:select	RAW	[ontology:select query]	Sélection d'instances par requête structurelle.
ontology:ask		[ontology:ask query]	Requête Prolog brute.
ontology:askMagic		[ontology:askMagic question]	Question NL avec résolution assistée LLM.
ontology:json		[ontology:json]	Sérialise l'ontologie en JSON.
ontology:toVVL		[ontology:toVVL]	Sérialise l'ontologie en code source VVL.

Builtin	Mode	Signature	Description
ontology:export		[ontology:export path format?]	Export disque : JSON / TTL / Gherkin.
ontology:loadGherkin		[ontology:loadGherkin path]	Charge depuis syntaxe Gherkin (.feature).
ontology:logKB		[ontology:logKB]	Dump des faits Prolog en console.
ontology:debugLog		[ontology:debugLog on|off]	Active le log verbeux.

Figure A.30 — *Requêtes et exports*

Persistance et stores nommés

Builtin	Signature	Description
ontology:save	[ontology:save path?]	Sauvegarde sur disque (path par défaut = store courant).
ontology:saveAs	[ontology:saveAs name path]	Sauvegarde un store nommé à un chemin.
ontology:load	[ontology:load path]	Charge depuis disque (remplace le store actif).
ontology:loadNamed	[ontology:loadNamed name path]	Charge sous un slot nommé (multi-store).
ontology:listStores	[ontology:listStores]	Liste les stores chargés.
ontology:storePath	[ontology:storePath name?]	Lit/définit le chemin disque d'un store.
ontology:merge	[ontology:merge source target]	Fusionne deux stores.

Figure A.31 — *Persistance ontologique*

A.25. Réification de concepts (`@`)

Le mécanisme de réification (chapitre 9) transforme un nom de concept en objet manipulable, doté de propriétés découvertes paresseusement par LLM. Sept builtins le supportent.

Builtin	Signature	Description
@	[@ "Name" mode?]	Réifie un concept ; modes "lazy" (sans bootstrap) ou ClassName.
concept:method	[concept:method $c "name" lambda]	Définit une méthode utilisateur sur un concept.
concept:reset	[concept:reset name?]	Vide le cache des concepts (tous ou spécifique).
concepts	[concepts "/path" as "prefix"?]	Charge les concepts persistés sous kb/neuro/.
concepts:available	[concepts:available]	Liste les index de concepts persistés.
concepts:info	[concepts:info path?]	Informations de session ou d'un store.
concepts:list	[concepts:list]	Liste les chemins chargés dans la session.

Figure A.32 — *Réification de concepts*

```
$soleil = [@ "Soleil"]
$soleil.diametre_km                # => découverte paresseuse via LLM
```

```
$soleil.luminosite                # => idem

$terre = [@ "Terre"]
$commun = ($soleil * $terre)      # intersection conceptuelle
($soleil ~~ "Etoile")             # satisfies hybride => ~0.95
($soleil / "type")                # projection => "G-type main-sequence"

# Mode lazy : pas de bootstrap, juste le nom
$arch = [@ "Chantier etancheite terrasse" "lazy"]
```

A.26. Résolution NL→VVL (`§§`, `§§?`)

Ces deux builtins compilent une question en langage naturel en code VVL, puis l'exécutent. La forme verbose `§§?` montre le code généré, ce qui est utile pour le debug et l'apprentissage.

Builtin	Signature	Description
resolve	[resolve "question"] ou [§§ "question"]	NL → VVL → exécution → résultat.
resolveV	[resolveV "question"] ou [§§? "question"]	Idem mais affiche le code VVL généré.

Figure A.33 — *Résolution symbolique NL→VVL*

A.27. Catalogue des génériques (keywords paramétrés)

Le mécanisme `[generic]` généralise une résolution NL en keyword catalogue persistant. Douze builtins gèrent ce catalogue : création, exécution, introspection, validation.

Builtin	Signature	Description
generic	[generic "question" "name"?]	Crée un keyword générique depuis une question.
generic:invoke	[generic:invoke "name" ...]	Exécute un keyword générique avec préservation des approximations.
generic:list	[generic:list status?]	Liste les fonctions du catalogue workspace.
generic:info	[generic:info name]	Fiche complète d'une fonction.
generic:code	[generic:code name]	Code VVL source du keyword mémorisé.
generic:contract	[generic:contract name]	Paramètres et contrat de sortie.
generic:pseudocode	[generic:pseudocode name]	Pseudo-code markdown depuis le catalogue.
generic:tests	[generic:tests name]	Tests / cas de validation mémorisés.
generic:validate	[generic:validate name]	Exécute les tests et met à jour le score.
generic:evolve	[generic:evolve name]	Régénère depuis la question source en conservant le nom.
generic:load	[generic:load name]	Charge le keyword dans le contexte courant.
generic:forget	[generic:forget name]	Supprime du catalogue avec ses artefacts.

Figure A.34 — *Catalogue de keywords génériques*

A.28. Persistance objet (SQLite + Prolog)

Le système de persistance combine SQLite pour le stockage durable et la KB Prolog pour le requêtage symbolique. Huit builtins forment le pont entre la couche objet (chapitre 6) et ces deux stores.

Builtin	Signature	Description
obj-save	[obj-save $instance]	Sauvegarde en SQLite ; UUID auto, indexation des scalaires.
obj-load	[obj-load ClassName id]	Charge une instance depuis SQLite par UUID.
obj-delete	[obj-delete $instance]	Supprime de SQLite par UUID.
obj-find	[obj-find Class field value op?]	Recherche par valeur de champ ; op : =, >=, <=, >, <, !=, LIKE.
obj-query	[obj-query Class "SQL WHERE clause"]	SQL brut sur vvl_objects (à utiliser avec précaution).
obj-join	[obj-join FromClass relF ToClass fF fV]	Jointure relationnelle inter-classes via vvl_relations.
obj-assert	[obj-assert $instance]	Injecte instance/2 + prop/3 + rel/3 dans la KB Prolog.
obj-retract	[obj-retract $instance]	Supprime tous les faits Prolog d'une instance.

Figure A.35 — *Persistance objet*

A.29. Calcul stack (Uiua-style)

Builtin	Signature	Description
compute	[compute stack ops]	Moteur stack-based inspiré d'Uiua. Ex : `[compute [] [5 "iota" 2 "*"]]`.

Figure A.36 — *Calcul stack-based*

A.30. CSV : core et extensions

Le module CSV combine un noyau de deux constructeurs (`CSV` et `BATCH`) avec dix-neuf extensions pour le requêtage, l'indexation, le filtrage et la persistance.

Constructeurs

Builtin	Signature	Description
CSV	[CSV "filename.csv"]	Crée ou charge un fichier CSV.
BATCH	[BATCH "name" "template"]	Crée un batch RAG-formaté avec placeholders {champ}.

Figure A.37 — *Constructeurs CSV et BATCH*

Manipulation et requêtage CSV

Builtin	Signature	Description
csv-add	[csv-add $csv record]	Ajoute un enregistrement.
csv-all	[csv-all $csv]	Tous les enregistrements en liste.
csv-count	[csv-count $csv]	Nombre d'enregistrements.

Builtin	Signature	Description
csv-find	[csv-find $csv key]	Recherche par clé primaire.
csv-update	[csv-update $csv key updates]	Met à jour par clé primaire.
csv-delete	[csv-delete $csv key]	Supprime par clé primaire.
csv-filter	[csv-filter $csv lambda]	Filtre par prédicat.
csv-where	[csv-where $csv field op value]	Filtre rapide ; ops : =, !=, >, <, >=, <=, contains, startswith, endswith.
csv-where-indexed	[csv-where-indexed $csv field value]	Recherche O(1) sur champ indexé.
csv-create-index	[csv-create-index $csv field]	Crée un index sur un champ.
csv-orderby	[csv-orderby $csv fieldname]	Définit l'ordre de tri.
csv-key	[csv-key $csv fieldname]	Définit la clé primaire.
csv-setfield	[csv-setfield $csv name type]	Ajoute/modifie un champ du schéma.
csv-erase	[csv-erase $csv]	Vide données et schéma.
csv-save	[csv-save $csv]	Force la sauvegarde sur disque.
csv-reload	[csv-reload $csv]	Recharge depuis disque.

Figure A.38 — *Manipulation et requêtage CSV*

Manipulation des batchs

Builtin	Signature	Description
batch-add	[batch-add $batch record]	Ajoute un enregistrement formaté.
batch-data	[batch-data $batch]	Liste des chaînes formatées.
batch-erase	[batch-erase $batch]	Vide le batch.

Figure A.39 — *Batchs RAG-formatés*

A.31. Office COM (Word, Excel, Outlook, PowerPoint)

Le module Office COM expose 35 builtins pour piloter les applications Microsoft Office via leur interface COM (Windows uniquement). Cette section présente les principales sous-commandes ; le détail complet est accessible via `[help "office"]`.

Connexion et gestion de l'application

Builtin	Signature	Description
office	[office app action ...]	Dispatch principal pour Word/Excel/Outlook/PowerPoint.
call	[call obj method args...]	Appelle une méthode COM dynamique.
get	[get obj property]	Lit une propriété COM.
set	[set obj property value]	Définit une propriété COM.
released	[released obj]	Libère un objet COM.

Figure A.40 — *Pilotage COM générique*

Word, Excel, Outlook, PowerPoint

Les sous-commandes spécifiques de chaque application Office sont accessibles via `[office word ...]`, `[office excel ...]`, `[office outlook ...]`, `[office pwp ...]`. Elles couvrent : création/ouverture/sauvegarde de documents, manipulation de paragraphes, tableaux, cellules, feuilles, mails, slides, formes, et exports en PDF. Voir `[help "office"]` ou la documentation interactive du REPL pour la liste exhaustive.

A.32. Selenium WebDriver et scraping

Le module Selenium offre un wrapper VVL au-dessus du WebDriver standard pour automatiser un navigateur (Chrome, Firefox, Edge). Il complète `scrape` (HTTP simple) pour les pages dynamiques chargées par JavaScript.

Builtin	Signature	Description
webdriver	[webdriver chrome|firefox options?]	Démarre une session WebDriver.
go	[go url]	Navigue vers une URL.
title	[title]	Titre de la page courante.
url	[url]	URL courante.
text	[text selector]	Texte d'un élément (CSS/XPath).
click	[click selector]	Clique sur un élément.
type	[type selector value]	Saisit du texte dans un champ.
wait	[wait selector timeout?]	Attend qu'un élément apparaisse.
screenshot	[screenshot path]	Capture d'écran de la page.
executeScript	[executeScript js args?]	Exécute du JavaScript dans la page.
scrollTo	[scrollTo selector|y]	Défile jusqu'à un élément ou une position.
cookies	[cookies]	Lit les cookies de la session.
setCookie	[setCookie name val opts?]	Définit un cookie.
webQuit	[webQuit]	Termine la session WebDriver.

Figure A.41 — *Selenium WebDriver — principales commandes*

Le pattern de **fallback** sur Selenium pour les pages dynamiques utilise typiquement l'opérateur de tolérance `~>` : `[scrape url] ~> [do $wd = [webdriver chrome] [go url] ...]`. Le premier essai HTTP simple est rapide ; en cas d'échec (page vide ou JS non exécuté), Selenium prend le relais.

A.33. Image : vision et génération (DALL-E)

Le module image expose les capacités de vision (analyse d'image par LLM) et de génération (DALL-E). La syntaxe `[§ "description" IMG "WxH"]` est un raccourci pour la génération.

Builtin	Signature	Description
imageGen	[imageGen "desc" size?]	Génère une image via DALL-E.
imageDescribe	[imageDescribe path]	Décrit une image via vision LLM.
imageOcr	[imageOcr path]	OCR via vision LLM (sémantique, pas Tesseract).
imageCrop	[imageCrop path x y w h out]	Recadre une image.

Builtin	Signature	Description
imageRotate	[imageRotate path angle out]	Rotation.
imageFilter	[imageFilter path filter out]	Filtre PIL : blur, sharpen, contour, etc.
imageThumbnail	[imageThumbnail path size out]	Miniature.
imageColors	[imageColors path n?]	N couleurs dominantes.
imageCompare	[imageCompare path1 path2]	Similarité visuelle.
imageMontage	[imageMontage paths grid out]	Montage de plusieurs images.

Figure A.42 — *Vision et génération d'images*

A.34. Email (SMTP/IMAP/POP3)

Le builtin `[email]` est un dispatcher unifié pour les trois protocoles email standards. Les opérations couvrent connexion, envoi, lecture, recherche, gestion de drapeaux, déplacement et suppression.

Sous-commande	Signature	Description
connect	[email connect name proto host port user pass]	Connexion SMTP/IMAP/POP3.
disconnect	[email disconnect name]	Termine la connexion.
list	[email list]	Liste les connexions actives.
send	[email send name to subject body]	Envoi (SMTP).
folders	[email folders name]	Liste les dossiers IMAP.
inbox	[email inbox name folder? limit?]	Lecture de la boîte.
search	[email search name criteria folder? limit?]	Recherche IMAP.
read	[email read name uid folder?]	Lit un mail par UID.
flag/unflag	[email flag|unflag name uid flag folder?]	Gestion des drapeaux.
move	[email move name uid dest source?]	Déplace un mail.
delete	[email delete name uid folder?]	Supprime un mail.
save	[email save name uid dir folder?]	Sauvegarde un mail sur disque.
pop-list	[email pop-list name]	Liste POP3.
pop-read	[email pop-read name index]	Lit par index POP3.
pop-delete	[email pop-delete name index]	Supprime par index POP3.

Figure A.43 — *Sous-commandes du builtin email*

A.35. Google APIs (Gmail, Calendar)

Builtin	Signature	Description
gmailList	[gmailList query max]	Liste les emails non lus (JSON).
gmailSend	[gmailSend to subject body]	Envoie un email.
calendarList	[calendarList max]	Liste les événements à venir.
calendarCreate	[calendarCreate summary startISO endISO desc?]	Crée un événement.

Figure A.44 — *APIs Google*

A.36. Hypervisor (Portainer / Docker)

Le builtin `[hypervisor]` est un dispatcher pour la gestion d'infrastructure Docker via l'API Portainer. Il couvre conteneurs, stacks, images, volumes, réseaux, templates et le déploiement ProSuite.

Principales catégories de sous-commandes :

- **Connection** : connect, connect-saved, save, disconnect, list, use

- **System** : status, info, resources

- **Containers** : containers, container, start, stop, restart, logs, exec

- **Stacks** : stacks, stack, stack-file, stack-env, stack-deploy, stack-update, stack-start, stack-stop, stack-remove

- **Images** : images, pull, image-remove

- **Volumes / réseaux** : volumes, volume, networks, network

- **Templates** : templates, template, template-create, template-update, template-remove

- **ProSuite** : ps-deploy, ps-update, ps-health, ps-instances, ps-scale

A.37. MCP (Model Context Protocol)

Le module MCP expose les commandes pour communiquer avec les serveurs MCP standards via stdio. Il permet à VVL de consommer les outils définis dans n'importe quel serveur MCP tiers.

Builtin	Signature	Description
mcp-connect	[mcp-connect name command env?]	Connecte un serveur MCP via stdio.
mcp-list	[mcp-list $client]	Liste les outils disponibles (JSON).
mcp-call	[mcp-call $client tool args?]	Appelle un outil MCP ; retour JSON.
mcp-resource	[mcp-resource $client uri]	Lit une ressource MCP.
mcp-servers	[mcp-servers]	Liste les connexions MCP actives.
mcp-close	[mcp-close $client]	Ferme la connexion.

Figure A.45 — *Builtins MCP*

A.38. Bases de données (SQLite, PostgreSQL)

Builtin	Signature	Description
db	[db action name args...]	SQLite : open, query, exec, tables.
pg	[pg conn|action ...]	PostgreSQL : connexion et requêtes.

Figure A.46 — *Bases de données relationnelles*

A.39. Écosystème ProSuite — clients métier

L'écosystème ProSuite regroupe les clients VVL pour les services métier déployés par VAILS-AI. Chaque client expose un dispatcher unique qui retourne un objet callable avec ses sous-commandes propres. La connexion se fait par triplet `url / login / password`.

Client	Signature	Domaine
pis	[pis url user pass]	Partner Information System (partenaires, clients, parts, ordres, systèmes).
procontact	[procontact url user pass]	Gestion des contacts.
promail	[promail url user pass]	Mailboxes, mails, send, fetch, archives.
proplanning	[proplanning url user pass]	Planification d'interventions.
prosearch	[prosearch url user pass]	Moteur de recherche métier.
proshare	[proshare url user pass]	GED (files, storage, workflow, plugins, admin).
tekitisy	[tekitisy url login pass]	Gestion de sites web (pages, routing, macros, KV, deploy).
vdomAdmin	[vdomAdmin url login pass]	Administration VDOM (apps, types, plugins, timers, KV).
revolut	[revolut connect|order|...]	API Merchant Revolut (paiements, ordres, captures, refunds).

Figure A.47 — *Clients ProSuite*

Chaque client est documenté en détail par sa propre commande `[help "<client>"]`. À titre d'exemple, le client `proshare` est invoqué ainsi : `$ps = [proshare url user pass]` puis `[$ps "list" "/path"]`, `[$ps "upload" file]`, `[$ps "submit"]`, etc.

A.40. Voice (OpenAI Realtime) et Whole (FFI VDOM)

Voice

Le builtin `[voice]` expose la voix temps réel d'OpenAI. Sous-commandes : `support`, `config`, `start`, `stop`, `decode`, `list`, `get`.

Whole — FFI VDOM

Builtin	Signature	Description
vdomConnect	[vdomConnect url login pass]	Connexion SOAP à un serveur VDOM.
vdomGetApp	[vdomGetApp connection appName]	Récupère une application VDOM.
vdomCall	[vdomCall app funcName params?]	Appelle une fonction API VDOM.

Figure A.48 — *Whole — FFI VDOM*

A.41. Façades RAG spécialisées

Deux façades RAG sont pré-configurées pour interroger respectivement la documentation VVL et la documentation ProAI Builder. Elles requièrent une indexation préalable via les commandes REPL `:genius-learn` et `:proai-learn`.

Builtin	Signature	Description
genius	[genius "question" mode?]	Interroge la documentation VVL (modes hybrid/naive).
proai	[proai "question" mode?]	Interroge la documentation ProAI Builder.

Figure A.49 — *Façades RAG spécialisées*

A.42. Index alphabétique des principaux builtins

Pour terminer cette annexe, voici un index alphabétique synthétique des builtins les plus fréquemment utilisés. Pour la liste complète et exhaustive avec leur docstring d'origine, utiliser la commande REPL `[help "name"]` ou `[symbols]`.

Lettre	Builtins représentatifs
A	append, agent, architect, async, @ (réification)
B	BATCH, BOOL, batch-add, batch-data
C	class, channel, chr, compute, concat, contains, convert, csv-*
D	db, delay, dim, do
E	evaluate, execute, exit, expert, expert:save, expert:load
F	facts, fetch, fiber, file-read, file-write, filter, first, FLOAT, for, fuzzy:*
G	genius, generic, generic:*, getValue, getXMLValue, gmailList, gmailSend, go, guid
H	hash, help, hypervisor, html
I	if, imageGen, imageDescribe, imageOcr, INT, inspectWord, instr
J	JSON, json-keys, json-set, json-path-get, jsonGetValue, jsonToList, join
K	kb, keyword
L	lambda, length, list, listMatch, log, ls
M	map, match, mcp-call, mcp-connect, md, mid, min, mixin, mset, mutex
N	new, not
O	obj-save, obj-load, obj-find, obj-assert, ontology:*, ord
P	PDF, pdfText, pdfToImages, pis, pmap, pop, pow, log, procontact, prolog*, promail, promise, proshare
Q	query, quote
R	ragInsert, ragAsk, ragExtractJson, range, reduce, replace, resolve, reverse, revolut, rules
S	scrape, scrapeLinks, semaphore, semantic, set, slice, sort, split, sqrt, STRING, switch
T	tekitisy, timeout, timestamp, try
U	undefkeyword, unpack, update, users
V	vdomAdmin, vdomCall, voice, vvl-version
W	wait, when, while, with, workspace, workspaces
X	XML, ~~ (satisfies hybride)

Figure A.50 — *Index alphabétique synthétique*

A.43. Pour aller plus loin

Cette annexe a présenté l'ensemble des builtins VVL et VAILS structurés par domaine fonctionnel. Trois compléments sont accessibles depuis tout REPL VVL pour aller plus loin que cette référence statique :

`[help "name"]` retourne le docstring exact enregistré pour un builtin donné. C'est la source de vérité ultime — toujours synchronisée avec le runtime en cours d'exécution.

`[symbols]` liste tous les symboles disponibles dans le contexte courant, builtins et keywords confondus. Utile pour explorer rapidement ce qui est disponible.

`[genius "question"]` interroge la documentation VVL via RAG avec une question en langage naturel. Combine la puissance du LLM avec l'autorité des docstrings — particulièrement utile pour identifier le bon builtin face à un besoin métier exprimé en français.

Le langage VVL évolue régulièrement ; les builtins listés ici reflètent l'état du runtime au moment de la rédaction de cet ouvrage. La commande `[vvl-version]` retourne la version exacte du runtime en cours, à comparer avec celle citée en pied de page de cette annexe pour évaluer l'écart éventuel entre votre installation et la référence.

Annexe B — Système de plugins

Mots-clés : plugins, Ed25519, signature, bundle, .vvlplugin, sandbox, subprocess, inproc, anti-TOCTOU, magasin de clés, double garde, VScript, ws, dispatch, per-session

Le **système de plugins** de VVL fournit un mécanisme d'**extensibilité par keywords** : sans recompiler le serveur, un administrateur peut introduire dans un workspace de nouvelles commandes — préfixes d'évaluation qui se comportent en tout point comme des builtins, mais dont l'implémentation est portée par un script externe. Deux langages d'écriture sont disponibles : **Python** (mode par défaut) et **VScript** (langage embarqué, sandboxé ; voir l'Annexe C pour la spécification du langage proprement dit).

L'enjeu central d'un tel sous-système est moins fonctionnel que sécuritaire. Permettre l'exécution d'un code arbitraire dans le contexte d'un serveur multi-tenant expose à des risques de confidentialité, d'intégrité et de disponibilité. La réponse apportée par VVL repose sur une **chaîne de confiance cryptographique** : un plugin n'est exécutable qu'à la condition d'avoir été **signé hors-ligne** par un auteur dont la **clé publique** a été préalablement déclarée de confiance par l'administrateur du serveur. Cette discipline, complétée par une protection anti-TOCTOU qui sera détaillée plus bas, est invariante quel que soit le langage du plugin.

Cette annexe couvre l'intégralité du sous-système : modèle cryptographique, format du bundle distribuable, magasin de clés de confiance, cycle de vie d'un plugin, modes d'exécution et leurs gardes, dispatch dans le moteur, référence des commandes administrateur et utilisateur, SDK auteur en Python, limites connues. Le langage VScript est abordé ici uniquement sous l'angle « plugin » — sa grammaire, sa sémantique, son objet hôte `ws` et ses modernisations Tier 1 sont traités dans l'Annexe C.

> **Note —** Le sous-système repose sur la bibliothèque Python `cryptography` (primitives Ed25519). Si cette dépendance est absente d'une distribution figée, l'ensemble des commandes plugin se désactive proprement et lève une erreur explicite — aucune **dégradation silencieuse** vers un mode non vérifié n'est possible.

B.1. Principes directeurs

Quatre principes fondent l'architecture du sous-système. Aucun n'est négociable ; ils sont implémentés de telle sorte qu'un plugin ne peut pas s'en exonérer.

Confiance ancrée cryptographiquement

Le serveur ne fait pas confiance aux fichiers qu'on lui présente : il fait confiance à un **petit ensemble de clés publiques**, déclarées une fois pour toutes par l'administrateur via les commandes `plugin-trust-*`. Tout plugin doit prouver, par sa signature Ed25519, qu'il provient de l'une de ces clés. Aucune liste blanche de chemins, de noms ou d'origines réseau n'intervient : la confiance est une propriété cryptographique, vérifiable indépendamment du contexte d'exécution.

Vérification à chaque étape — anti-TOCTOU

La vulnérabilité time-of-check-to-time-of-use (TOCTOU) consiste à vérifier un fichier au moment de l'installation, puis à l'utiliser plus tard sans revérifier — fenêtre pendant laquelle un acteur malveillant peut substituer le contenu. VVL clôt cette fenêtre : la signature est vérifiée **deux fois**, une fois à

`[plugin-install]` et une fois à chaque `[plugin-load]`, contre une empreinte enregistrée dans le record d'installation. La confiance est ancrée dans ce record, pas dans la présence du fichier sur disque.

Code non signé : jamais d'exécution

Aucun mécanisme n'autorise l'exécution d'un fichier nu déposé dans le répertoire `plugins/` d'un workspace. Il n'existe pas de mode « test », pas de drapeau de développement, pas d'option permettant de contourner la vérification de signature. Pour développer un plugin, l'auteur dispose d'une **chaîne de signature locale** (`vvl-plugin-sign keygen` puis `sign`) qui produit un bundle vérifiable avec sa propre clé — qu'il aura déclarée de confiance sur son serveur de développement.

Privilèges : opt-in, doublement gardés

Les modes d'exécution privilégiés — `inproc` (Python) et exposition de l'objet `ws` (VScript) — ne s'activent jamais par défaut. Ils exigent **deux conditions conjointes** : un signataire de haut niveau (tier `high`) et un workspace de confiance (`plugins_trusted = 1`). Cette **double garde** garantit que l'octroi des privilèges relève d'une décision conjointe du signataire (par la qualité de sa clé) et de l'administrateur du workspace (par sa déclaration explicite).

B.2. Modèle cryptographique : Ed25519 et signature hors-ligne

La primitive cryptographique retenue est **Ed25519** (EdDSA sur la courbe Edwards 25519). Trois propriétés en motivent le choix : signatures déterministes (donc reproductibles, précieuses pour les builds), vérification rapide en présence d'un grand nombre de plugins, et clés courtes (32 octets) qui se transmettent aisément hors-bande — par comparaison d'empreinte SHA-256, par exemple. La signature elle-même fait 64 octets.

La propriété d'**asymétrie** structure le modèle de menace : l'auteur conserve la clé privée sur sa machine de développement (jamais transmise au serveur, jamais commitée dans un dépôt) ; seule la clé publique circule. Le serveur ne peut donc pas fabriquer un faux plugin, même s'il était compromis — il ne sait que vérifier. Cette asymétrie est ce qui rend la signature **hors-ligne** sûre : la machine de signature n'a jamais besoin d'être connectée au serveur cible.



Figure B.1 — *Chaîne de confiance de bout en bout — l'auteur signe avec sa clé privée, le serveur ne reçoit que la clé publique, et la signature est revérifiée à chaque chargement contre le digest enregistré dans le record d'installation.*

La **triple vérification** illustrée ci-dessus mérite une lecture attentive. Au moment de la signature (étape (i)), l'outil `vvl-plugin-sign` calcule un digest canonique du bundle — c'est-à-dire un hash SHA-256 d'une représentation **normalisée** du manifeste et des fichiers, indépendante de l'ordre des entrées ZIP, des horodatages ou des permissions. Ce digest est signé, et la paire `{digest, signature}` est insérée dans le bundle. À l'installation (étape (ii)), le serveur recalcule le même digest canonique, le compare à celui qui figure dans la signature, et vérifie la signature Ed25519 contre la clé publique référencée par `key_id`. À chaque chargement (étape (iii)), le digest du fichier `.py` (ou `.vscript`) sur disque est recalculé et comparé au digest enregistré dans `.registry.json` : toute divergence ferme la porte.

B.3. Le bundle .vvlplugin et son manifeste

Un bundle `.vvlplugin` est une **archive ZIP** contenant au minimum trois entrées : le **manifeste** (`manifest.json`), le **fichier d'entrée** (`<entry>.py` ou `<entry>.vscript`), et le **bloc signature** (`signature.json`). Des fichiers auxiliaires peuvent compléter le bundle (modules Python additionnels, ressources statiques) ; tous entrent dans le digest canonique.

Le **manifeste** est un objet JSON dont les champs structurants sont les suivants. Le tableau ci-dessous donne une description normative de chacun ; les valeurs entre crochets indiquent les défauts retenus en l'absence du champ.

Champ	Type	Description
`name`	String	Identifiant interne du plugin. Doit respecter **`[a-z][a-z0-9_-]*`** et ne pas entrer en collision avec un builtin du cœur (vérifié à l'installation).
`version`	String	Version sémantique de l'auteur (ex. **`1.2.0`**). Présente dans **`[plugin-info]`** ; ne participe pas à la sélection.

Champ	Type	Description
`entry`	String	Chemin relatif vers le fichier d'entrée dans l'archive (`geocode.py`, `validate.vscript`, …).
`lang`	String	Langage du plugin : `python` (défaut) ou `vscript`. Détermine le moteur d'exécution et la disponibilité des modes.
`mode`	String	Mode d'exécution par défaut au chargement : `subprocess` (défaut, Python) ou `inproc` (Python privilégié, sous double garde). Sans effet pour `lang: "vscript"`.
`env`	Boolean	VScript uniquement. Demande l'injection de l'objet hôte `ws` au chargement (sous double garde). Vaut `false` si omis.
`arity-min`	Integer	Nombre minimum d'arguments acceptés (vérifié à l'invocation, génère une `ArityError` si violé).
`arity-max`	Integer	Nombre maximum d'arguments acceptés. Une valeur égale à `arity-min` impose une arité fixe.
`description`	String	Texte libre exposé par `[plugin-info]` et `[help name]`. Optionnel mais recommandé.

Figure B.2 — *Champs structurants du manifeste — un manifeste minimal valide ne contient que name, version, entry et lang.*

Le **bloc signature** (`signature.json`) regroupe `digest` (SHA-256 du contenu canonique en hexadécimal), `sig` (signature Ed25519 du digest, encodée base64), `key_id` (identifiant déclaré au signer via `--key-id`, doit correspondre à une entrée du magasin `<PLUGIN_TRUST>`), et `signed_at` (timestamp ISO 8601, horodatage informatif, non vérifié contractuellement).

B.4. Le magasin de clés <PLUGIN_TRUST>

Le **magasin de clés de confiance** est persisté dans le fichier `vvl.config` du serveur, sous la forme d'un bloc XML global `<PLUGIN_TRUST>` placé à côté de `<USERS>` et `<SECURITY>` sous la racine `<CONFIG>`. Chaque clé publique connue est matérialisée par un sous-élément `<KEY>` portant les attributs suivants :

Attribut	Description
`id`	Identifiant logique de la clé (chaîne libre, par exemple `prosuite-2026`). Référencé par `key_id` dans la signature du bundle. Unicité imposée à l'ajout.
`tier`	Niveau de confiance accordé à la clé : `standard` ou `high`. Seul `high` autorise l'éligibilité au mode `inproc` (Python) et à l'objet `ws` (VScript), et toujours sous la condition supplémentaire `plugins_trusted = 1` au workspace.
`alg`	Algorithme de signature. Actuellement figé à `ed25519` ; la présence du champ permet d'envisager d'autres primitives sans rupture du schéma.
`fp`	Empreinte SHA-256 (DER) de la clé publique, sous forme hexadécimale. Sert à la **vérification hors-bande** : l'administrateur peut comparer l'empreinte annoncée par l'auteur à celle qu'il voit dans la configuration, sans avoir à manipuler la PEM elle-même.
`added`	Horodatage ISO 8601 de l'ajout de la clé au magasin.
`added_by`	Identifiant de l'administrateur ayant ajouté la clé (`proadmin` connecté au moment de l'appel `[plugin-trust-add]`).

Attribut	Description
contenu	La représentation **PEM** complète de la clé publique, en text content de l'élément `<KEY>`.

Figure B.3 — *Schéma d'une entrée du magasin de clés — la PEM occupe le contenu textuel, les métadonnées sont des attributs XML.*

La gestion du magasin est entièrement médiée par les huit commandes `plugin-trust-*`, toutes **gardées admin** : un appelant non authentifié comme administrateur (via `[proadmin email pass]`) reçoit une `PermissionError`. Cette restriction garantit que l'administrateur du serveur — et lui seul — décide quels auteurs sont habilités à livrer des plugins. La référence détaillée des huit commandes figure en section C.10.

B.5. Cycle de vie : keygen → sign → install → load

Le cycle de vie d'un plugin se compose de cinq étapes, dont deux sont à la charge de l'auteur (hors serveur) et trois à la charge de l'administrateur ou de l'utilisateur du serveur. La séquence ci-dessous illustre la mise en route complète d'un plugin `geocode` qui transforme une adresse postale en coordonnées géographiques.

5.1. Génération de la clé (auteur, hors-ligne)

L'auteur produit une paire de clés Ed25519 sur sa machine de développement. La clé privée (`author.pem`) reste **strictement locale** ; la clé publique (`author.pub`) est destinée à être communiquée à l'administrateur du serveur cible.

```
python scripts/vvl-plugin-sign.py keygen --out author.pem --pub author.pub
```

5.2. Signature du bundle (auteur, hors-ligne)

L'auteur signe son bundle en fournissant : la clé privée (`--key`), un identifiant logique de clé (`--key-id`, qui devra correspondre à une entrée du magasin côté serveur), les champs du manifeste à matérialiser dans l'archive (`--name`, `--version`, `--entry`, `--mode`, `--arity-min`, `--arity-max`) et le chemin de sortie du bundle. Pour un plugin VScript, on précisera `--lang vscript` et, le cas échéant, `--env` pour demander l'injection de l'objet `ws`.

```
python scripts/vvl-plugin-sign.py sign --key author.pem --key-id prosuite-2026 \
    --name geocode --version 1.2.0 --entry geocode.py \
    --mode subprocess --arity-min 1 --arity-max 1 --out geocode.vvlplugin
```

5.3. Import de la clé publique (admin, sur le serveur)

L'administrateur dépose le fichier `author.pub` reçu de l'auteur dans le répertoire `plugins/_inbox/` du serveur, puis enregistre la clé dans le magasin. Le `tier` détermine l'éligibilité ultérieure aux modes privilégiés ; la valeur `high` ne devrait être attribuée qu'aux auteurs dont l'identité a été vérifiée hors-bande par comparaison de l'empreinte SHA-256.

```
[proadmin "admin@example.org" "******"]
[plugin-trust-import "author.pub" prosuite-2026 high]
```

5.4. Installation du plugin (workspace)

L'utilisateur du workspace dépose le bundle `geocode.vvlplugin` dans `plugins/_inbox/` et lance la commande d'installation. Le serveur exécute alors la séquence complète de vérification — digest canonique, signature Ed25519, appartenance de `key_id` au magasin, politique (pas de collision avec un builtin, mode demandé compatible avec le tier de la clé) — puis matérialise atomiquement le fichier `geocode.py` dans `plugins/` et inscrit le record dans `.registry.json`.

```
[plugin-install "geocode.vvlplugin"]
```

5.5. Chargement et invocation (per-session)

Le chargement `[plugin-load]` re-vérifie le digest du fichier sur disque contre le record (défense anti-TOCTOU) puis lie le nom de session `geocode` au plugin dans `ctx.plugins`. Le keyword est alors disponible comme s'il s'agissait d'un builtin, avec la priorité de dispatch décrite en section C.9. La liaison étant **per-session**, elle disparaît à la fermeture du contexte ; on la rejoue en tête d'`init.vvl`.

```
[plugin-load "geocode" as geocode]
[geocode "10 rue de la Paix"]        # => {lat: 48.8566, lon: 2.3522}
```

> **Note —** Les liaisons posées par `[plugin-load]` ne sont **pas** persistées entre sessions. Le pattern recommandé consiste à regrouper toutes les liaisons de chargement dans un fichier `init.vvl` du workspace, lui-même chargé via `[import "init.vvl"]` au démarrage. Cette discipline rend explicite, par lecture du seul `init.vvl`, l'ensemble des extensions actives.

B.6. Garantie anti-TOCTOU : le record .registry.json

L'attaque time-of-check-to-time-of-use est l'écueil classique des systèmes qui vérifient un fichier au moment de l'admission, puis l'utilisent plus tard sans revérifier. Entre les deux instants, un acteur disposant d'un accès au système de fichiers peut substituer le contenu — par exemple, remplacer un script vérifié par une variante malveillante. La conséquence est sévère : le système exécute du code qu'il croit avoir vérifié mais qui ne l'a pas été.

La parade adoptée par VVL consiste à **ancrer la confiance dans un record persistant**, distinct du fichier exécutable. À l'installation, le serveur écrit dans `{ws}/plugins/.registry.json` une entrée qui décrit le plugin (`name`, `version`, `signer`, `mode`, `arity-min`, `arity-max`, `installed_at`) et contient surtout le **digest signé** du fichier d'entrée. À chaque `[plugin-load]`, le digest du fichier sur disque est recalculé et comparé à celui du record. Deux cas sont rejetés :

Situation	Détection	Réaction
Fichier déposé manuellement dans **`plugins/`**	Aucun record correspondant dans **`.registry.json`**.	Refus du chargement avec erreur **`PluginNotRegisteredError`**.

Situation	Détection	Réaction
Fichier installé puis altéré	Digest(fichier sur disque) ≠ digest enregistré dans le record.	Refus du chargement avec erreur `PluginDigestMismatchError`.
Record présent mais fichier absent	Tentative d'ouverture du `.py` ou `.vscript` échoue.	Refus avec `PluginFileMissingError`. `[plugin-info]` reste informatif.

Figure B.4 — *Trois rejets caractéristiques de la défense anti-TOCTOU — le record et le fichier doivent toujours s'accorder pour qu'un chargement aboutisse.*

Une conséquence opératoire utile : la commande `[plugin-remove]` est rendue **idempotente** par la présence du record. Elle débind le keyword s'il était chargé, supprime le fichier `.py` ou `.vscript` si présent, retire l'entrée du record. Si l'une de ces opérations était déjà accomplie (fichier supprimé à la main, par exemple), l'absence est silencieusement tolérée et le nettoyage des autres éléments se poursuit. Aucune **divergence durable** entre fichier et record ne peut ainsi persister.

B.7. Modes d'exécution Python : subprocess et inproc

Pour un plugin écrit en **Python** (`lang: "python"`, défaut), deux modes d'exécution sont disponibles. Ils diffèrent par leur mécanique d'invocation et, plus fondamentalement, par le niveau de privilège qu'ils accordent au code du plugin. Le tableau ci-dessous synthétise les différences avant de les détailler.

Propriété	subprocess (défaut)	inproc
Mécanique	Le serveur lance `python {entry}.py`, transmet la requête sur `stdin` au format JSON, lit la réponse sur `stdout`.	Le serveur importe le module via `importlib` et appelle directement la fonction `handle(args, ctx)`.
Contexte d'exécution	Processus séparé, isolé de la mémoire du serveur.	Même processus, même mémoire que le serveur.
Accès à `VVLContext`	Aucun. Le plugin reçoit ses arguments et renvoie sa valeur ; il ne peut pas appeler `[evaluate]`, lire le workspace, etc.	**Plein accès** au `VVLContext` réel : peut invoquer n'importe quel builtin, lire la configuration, accéder aux verrous DB, etc.
Privilèges effectifs	Aucun privilège serveur ; tout est limité à ce qu'autorise le jail du processus.	**Privilèges serveur** : équivalent à du code natif du cœur.
Bornage	Timeout dur, signal de kill propre, `cwd = {ws}/files`, variables d'environnement nettoyées.	Bornage logique uniquement (`ctx.set_timeout`, `ctx.should_stop`) ; pas d'isolation mémoire.
Condition d'activation	Aucune : disponible par défaut.	**Double garde** : clé signataire de `tier = high` et workspace `plugins_trusted = 1`.

Figure B.5 — *Comparaison des deux modes Python — subprocess est le mode par défaut, sûr en toute circonstance ; inproc est un mode à privilèges, doublement gardé.*

Mode subprocess

Le mode `subprocess` constitue le défaut sûr. Pour chaque invocation, le serveur exécute `python {entry}.py` dans un **processus enfant**, lui transmet la requête sérialisée en JSON sur `stdin`, attend la

réponse sur `stdout`, puis interprète le résultat. La requête contient les arguments du keyword (déjà évalués côté VVL), le nom du keyword tel qu'invoqué, et le nom du plugin :

```
# Format de la requête envoyée sur stdin
{
  "keyword": "geocode",
  "plugin":  "geocode",
  "args":    ["10 rue de la Paix"]
}

# Format de la réponse attendue sur stdout
{ "ok": true,  "value": { "lat": 48.8566, "lon": 2.3522 } }
# ou
{ "ok": false, "error": "adresse vide" }
```

Plusieurs **bornages** complètent cette isolation. Le `cwd` du sous-processus est positionné à `{ws}/files` (le plugin ne voit donc, par défaut, que le répertoire fichiers du workspace, et non l'arbre serveur). Les variables d'environnement sont filtrées : seules celles strictement nécessaires à Python sont conservées (`PATH`, `PYTHONPATH`, `LANG`, …). Un timeout dur, calé sur `ctx.set_timeout`, déclenche un signal `SIGTERM` puis `SIGKILL` si le processus persiste. Enfin, le signal de **stop utilisateur** (`ctx.should_stop`) interrompt le sous-processus de manière coopérative.

Mode inproc

Le mode `inproc` est, à l'inverse, un mode **à privilèges**. Le serveur importe le module Python via `importlib`, recherche la fonction `handle(args, ctx)`, et l'appelle avec les arguments du keyword et le `VVLContext` réel — celui-là même qu'utilisent les builtins du cœur. Le plugin peut donc, en pratique, tout faire ce que peut faire le serveur : ouvrir une base, écrire dans la configuration, déclencher d'autres plugins, etc.

L'octroi de ce mode est conditionné à la **conjonction** de deux propriétés. D'abord, la clé qui a signé le plugin doit avoir été enregistrée avec `tier = high` — ce qui suppose que l'administrateur du serveur a vérifié l'identité du signataire hors-bande (par exemple, par comparaison d'empreinte SHA-256). Ensuite, le workspace lui-même doit avoir été déclaré de confiance via `[plugin-workspace-trust name 1]` — décision qui revient à l'administrateur du serveur ou du workspace. La rupture d'**une seule** des deux conditions ramène le plugin au mode `subprocess`, quelle que soit la valeur de `mode` dans le manifeste.

> **Note —** La sémantique « **toujours du moins-privilégié vers le plus-privilégié** » est centrale : si un plugin déclare `mode: "inproc"` mais que la double garde n'est pas satisfaite, le serveur ne lève pas d'erreur — il dégrade silencieusement vers `subprocess`. C'est l'inverse qui serait dangereux : il ne faut **jamais** que la perte d'une condition de garde fasse passer un plugin d'un mode plus restreint à un mode moins restreint.

B.8. Plugins en VScript : sandbox natif et garde env

Un plugin peut, alternativement, être écrit en **VScript** — le langage de script embarqué dans VAILS (voir Annexe C). Cette possibilité ouvre deux avantages : d'abord, le langage est dérivé de VBScript et adopte une syntaxe accessible aux administrateurs déjà familiers de l'environnement VDOM historique ; ensuite, et surtout, VScript est **sandboxé par construction**.

Concrètement, le moteur VScript embarqué dans VAILS n'expose **aucune primitive d'I/O** : pas de lecture/écriture de fichier, pas de socket réseau, pas d'accès au système d'exploitation, pas d'invocation de processus. Cette restriction est de nature langage (les builtins correspondants ne sont pas présents dans le runtime) et non de nature processus — elle ne dépend donc pas de garde-fous OS qui pourraient être contournés. Conséquence directe : un plugin VScript est **sandboxé par défaut**, ce qui rend inutile la double garde du mode `inproc` Python.

L'objet hôte ws — accès opt-in à l'environnement

Pour les cas où un plugin doit malgré tout toucher à l'environnement du workspace (lire un fichier, interroger une base, écrire un journal), la façade VAILS injecte un objet hôte **`ws`** dont les méthodes sont des enveloppes fines au-dessus des builtins existants : elles héritent donc des **mêmes verrous** (DB, fichiers) et de la **même résolution de chemin** (`ctx.resolve_path`), c'est-à-dire du jail applicatif. Le tableau ci-dessous récapitule la surface exposée.

Objet	Méthodes	Jail / cible
`ws.name`	lecture du nom du workspace	—
`ws.files`	`.read` / `.write` / `.exists` / `.delete`	`{ws}/files`
`ws.csv(name)`	`.all` / `.find` / `.where` / `.add` / `.update` / `.delete` / `.save` / `.clear` / `.count`	`{ws}` (fichiers CSV nommés)
`ws.db(name)`	`.exec` / `.query` / `.tables` (lignes en `dictionary`)	`{ws}/db/<name>.sqlite`
`ws.mem(name)`	`.value` / `.put` (KV ; les noms `get`/`set` sont réservés)	`{ws}/db`
`ws.log msg`	journal du workspace	—
`ws.vvl(code)`	rappel VVL — évalue une chaîne VVL dans le contexte serveur	—

Figure B.6 — *Surface de l'objet hôte ws — chaque sous-objet est une enveloppe au-dessus des builtins existants, donc soumis aux mêmes verrous que le code natif.*

L'injection de `ws` n'est **pas** automatique : elle constitue un dépassement du sandbox naturel du langage, et obéit donc à la **même double garde** que le mode `inproc` Python. Trois conditions sont conjointement requises : (i) le manifeste déclare `env: true` (option `--env` du signer, donc décision de l'auteur matérialisée dans le bundle signé) ; (ii) la clé signataire est de `tier = high` (décision du serveur sur la qualité de la signature) ; (iii) le workspace cible est `plugins_trusted = 1` (décision du workspace sur l'acceptation de plugins privilégiés). Le défaut d'une seule des trois conditions fait que toute référence à `ws` dans le plugin lève une `EnvironmentAccessError`.

> **Note —** L'utilisation `[vscript "…"]` inline (voir Annexe C) constitue un cas particulier : `ws` y est **toujours disponible**, car le code inline est, par construction, du code serveur — comparable à `[evaluate]`. Seuls les plugins VScript (`lang: "vscript"`) sont soumis à la garde `env: true`.

B.9. Dispatch et priorité dans le moteur

Lorsque le moteur reçoit un symbole en position de tête d'expression — par exemple `[geocode "…"]` — il résout l'implémentation en parcourant **quatre niveaux dans un ordre strict** : (1) `BUILTINS` (cœur du langage), (2) `ctx.keywords` (keywords utilisateur définis par `[keyword]`), (3) `ctx.plugins` (liaisons de plugin de la session courante), (4) fallback (auto-évaluation, ou erreur `UnknownKeyword` si la position est applicative). La conséquence cardinale est qu'un plugin **ne peut jamais usurper un builtin** : la collision est rejetée à l'installation, et le dispatch protège ce qui aurait pu y échapper.

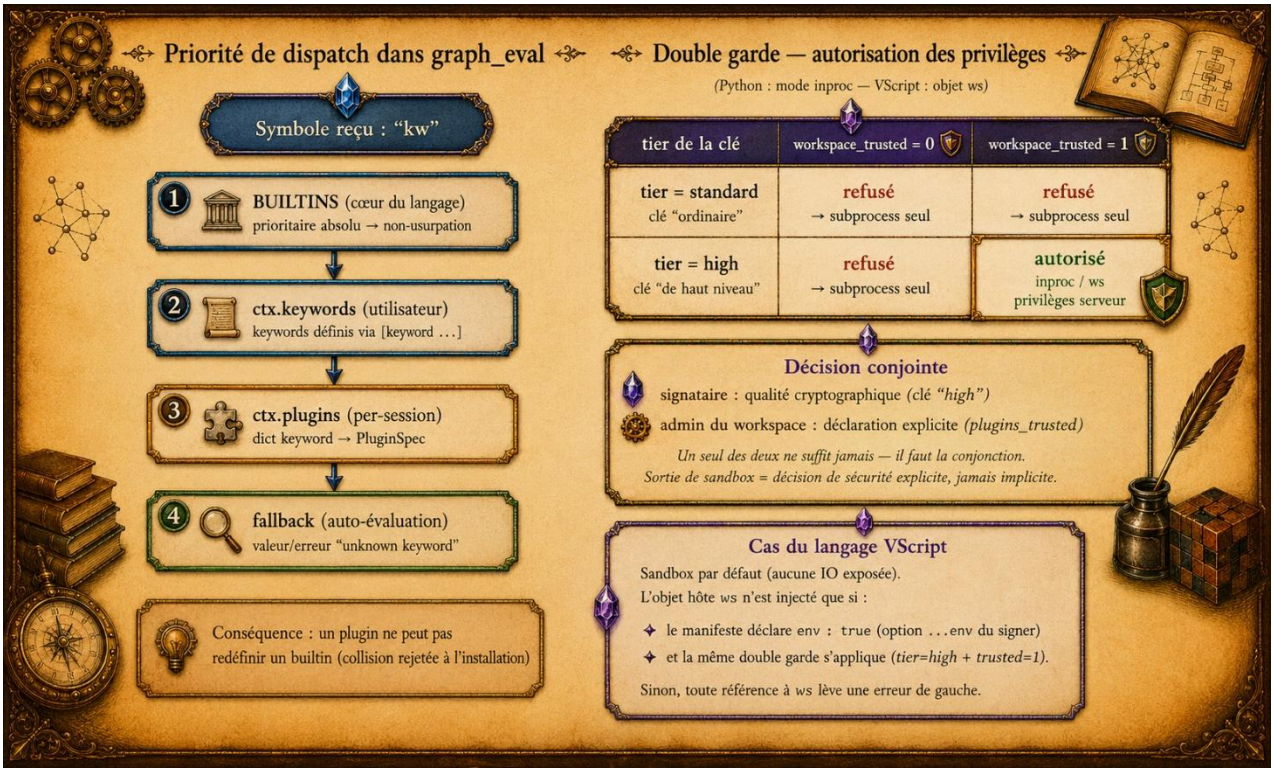


Figure B.7 — *Priorité de dispatch (à gauche) et matrice de double garde (à droite) — les deux mécaniques sont indépendantes mais cohérentes : la priorité protège contre l'usurpation, la matrice protège contre l'octroi non concerté de privilèges.*

La structure `ctx.plugins` est un dictionnaire dont les clés sont les noms de session liés par `[plugin-load … as kw]` et les valeurs sont des objets `PluginSpec` qui décrivent le manifeste, le digest, le mode, et la clé signataire. Cette structure est partagée par référence dans `ctx.clone()` (utilisé par les builtins de portée comme `[lambda]`) et dans l'expansion de macro, ce qui assure la propagation transparente des liaisons dans les contextes dérivés.

En revanche, `ctx.plugins` n'est pas persisté : la liaison disparaît à la fermeture du contexte racine. Le pattern recommandé pour rejouer automatiquement les liaisons à l'ouverture du workspace consiste à regrouper les `[plugin-load]` dans un fichier `init.vvl` du workspace :

```
# Fichier init.vvl — chargé en tête de session via [import "init.vvl"]
[plugin-load "geocode"   as geocode]
[plugin-load "validate"  as v.address]
[plugin-load "translate" as tr]
```

B.10. Référence des commandes `plugin-trust-*` (admin)

Les huit commandes ci-dessous gèrent le magasin `<PLUGIN_TRUST>`. Toutes exigent une connexion admin (`[proadmin email pass]`) au préalable ; à défaut, elles lèvent une `PermissionError`. Les commandes qui modifient le magasin (`add`, `import`, `update`, `remove`) écrivent atomiquement la nouvelle configuration sur disque.

Commande	Effet
`[plugin-trust-add id "<PEM>" tier]`	Ajoute une clé publique Ed25519 au magasin. **`tier`** ∈ {**`standard`**, **`high`**}. Refuse un **`id`** déjà présent (utiliser **`update`** pour modifier).
`[plugin-trust-import "f.pub" id tier]`	Lit un fichier **`.pub`** depuis **`plugins/_inbox/`** et procède à un **`add`** — évite de coller une PEM multi-ligne dans la commande.
`[plugin-trust-list]`	Renvoie un JSON de toutes les clés enregistrées (**`id`**, **`tier`**, **`alg`**, **`fp`**, **`added`**, **`added_by`**) **sans** la PEM brute, pour visualisation rapide.
`[plugin-trust-get id]`	Renvoie le détail complet d'une clé, avec la PEM, à des fins d'audit ou de réexport.
`[plugin-trust-update id tier]`	Modifie le tier d'une clé existante. Cas d'usage : promotion **`standard`** → **`high`** après vérification d'identité.
`[plugin-trust-remove id]`	Révoque/supprime la clé du magasin. Les plugins signés par cette clé deviennent **inchargeables** dès la prochaine vérification ; les chargements en cours ne sont pas interrompus.
`[plugin-trust-export id "out.pub"]`	Réécrit la PEM dans **`plugins/_inbox/<out.pub>`**, par exemple pour vérifier l'empreinte ou la communiquer à un tiers.
`[plugin-workspace-trust [name] 0|1]`	(Dé)active **`plugins_trusted`** sur un workspace. Un argument unique = workspace courant. Constitue la deuxième porte de la double garde des modes privilégiés.

Figure B.8 — *Référence des huit commandes plugin-trust-* — toutes gardées admin, toutes opérant sur <PLUGIN_TRUST> ou plugins_trusted.*

B.11. Référence des commandes de cycle de vie

Les six commandes ci-dessous gèrent les plugins au niveau du workspace courant. `[plugin-install]` et `[plugin-remove]` modifient la configuration persistante du workspace ; `[plugin-load]`, `[plugin-unload]`, `[plugins-list]` et `[plugin-info]` opèrent dans la session courante.

Commande	Effet
`[plugin-install "f.vvlplugin"]`	Lit le bundle dans **`plugins/_inbox/`**, vérifie signature et politique, écrit le fichier d'entrée et crée le record dans **`.registry.json`**. Atomique : en cas d'échec, aucun reliquat partiel.
`[plugin-remove "name"]`	Débind le keyword s'il est lié, supprime le fichier d'entrée et l'entrée du record. **Idempotent** — toute opération déjà accomplie est silencieusement tolérée.
`[plugin-load "name" as kw]`	Recompute le digest du fichier sur disque, le compare au record, et lie **`ctx.plugins[kw]`**. Variante : **`[plugin-load "name" as kw mode "inproc"]`** force le mode (sous condition de double garde). **RAW** — le nom de keyword n'est pas évalué.
`[plugin-unload kw]`	Retire la liaison **`ctx.plugins[kw]`** de la session courante. N'affecte ni le record ni le fichier. **RAW**.
`[plugins-list]`	Renvoie l'état des plugins du workspace : installés (présence du record) avec leur statut de chargement (lié à un keyword de session, ou non).
`[plugin-info name]`	Renvoie pour un plugin nommé : son manifeste, le signataire (**`id`**, **`tier`**, **`fp`**), le digest enregistré, et son statut courant.

Figure B.9 — *Référence des six commandes de cycle de vie — install/remove modifient la configuration persistante ; load/unload sont per-session.*

B.12. Écrire un plugin Python : le SDK vvl_plugin_sdk

Le SDK auteur expose trois primitives — `serve`, `ok`, `fail` — et impose une seule convention : le plugin doit exposer une fonction `handle(args, ctx=None)` qui prend les arguments du keyword sous forme de liste et retourne une valeur. Le même `handle` fonctionne dans les deux modes : en `subprocess`, `serve(handle)` orchestre la boucle stdin/stdout ; en `inproc`, le serveur appelle directement `handle(args, ctx)` avec le `VVLContext` réel.

```python
# geocode.py — squelette canonique d'un plugin Python
from vvl_plugin_sdk import serve, ok, fail

def handle(args, ctx=None):
    addr = args[0] if args else ""
    if not addr:
        return fail("adresse vide")
    # ... logique métier ...
    return ok({"lat": 48.8566, "lon": 2.3522})

if __name__ == "__main__":
    serve(handle)        # mode subprocess ; ignoré en inproc
```

Les retours du `handle` peuvent prendre trois formes équivalentes au niveau du protocole : `ok(valeur)` produit `{"ok": true, "value": valeur}` ; `fail("message")` produit `{"ok": false, "error": "message"}` ; une valeur nue retournée par `handle` est automatiquement enveloppée en `ok`. La distinction `ok` / `fail` est exposée côté VVL : un `fail` lève une exception capturable par `[try]`, alors qu'un `ok` rend simplement sa valeur.

Le **protocole de communication** entre le serveur et le sous-processus est le suivant. Le serveur écrit sur `stdin` un objet JSON unique :

```
{ "keyword": "geocode", "plugin": "geocode", "args": ["10 rue de la Paix"] }
```

Le sous-processus est censé répondre par un objet JSON unique sur `stdout`, soit en cas de succès :

```
{ "ok": true,  "value": { "lat": 48.8566, "lon": 2.3522 } }
```

soit en cas d'erreur métier :

```
{ "ok": false, "error": "adresse vide" }
```

En cas d'erreur système (timeout, signal, code de sortie non nul, JSON malformé), le serveur synthétise lui-même une réponse `{ok: false}` portant un message de niveau infrastructure (`PluginTimeoutError`, `PluginProtocolError`, etc.). Le SDK auteur n'a donc pas à se préoccuper de ces cas — la `serve` se charge de la propagation normale, le reste est intercepté côté serveur.

B.13. Récapitulatif des garanties de sécurité

Les sections précédentes décrivent en détail les mécanismes du sous-système. Le tableau ci-dessous les ré-articule sous la forme d'un récapitulatif des garanties — chaque ligne énonce une propriété de sécurité et le mécanisme technique qui la porte.

Garantie	Mécanisme
Authenticité	Signature **Ed25519** vérifiée à l'installation contre une clé publique inscrite dans `<PLUGIN_TRUST>`. La clé privée ne circule jamais.
Intégrité au repos	Le digest signé est enregistré dans `.registry.json` à l'installation, et **revérifié à chaque chargement** contre le fichier sur disque. Défense anti-TOCTOU.
Privilège (Python)	`subprocess` par défaut, isolé du processus serveur. `inproc` exige la **double garde** (`tier = high` et `plugins_trusted = 1`).
Isolation langage (VScript)	Aucune I/O exposée par le langage → **sandbox par défaut**. L'objet `ws` n'est injecté que sous la même double garde, et de surcroît `env: true` doit figurer dans le manifeste signé.
Non-usurpation	Le dispatch consulte `BUILTINS` avant `ctx.plugins`. La collision avec un nom de builtin est rejetée à l'installation. Un plugin ne peut donc jamais redéfinir un builtin.
Bornage runtime	Timeout dur, signal de stop coopératif, `cwd` confiné à `{ws}/files`, environnement nettoyé. Les bornages logiques (`ctx.set_timeout`, `ctx.should_stop`) s'appliquent aussi en `inproc`.
Gardes admin	Toutes les commandes `plugin-trust-*` exigent `[proadmin]` connecté. Toute tentative non-admin lève `PermissionError`.

Figure B.10 — *Récapitulatif des sept garanties de sécurité du sous-système — chacune est portée par un mécanisme indépendant, ce qui multiplie les barrières en cas de compromission partielle.*

B.14. Limites connues et perspectives

Le sous-système couvre l'ensemble des risques applicatifs mais ne résout pas, dans son état actuel, l'ensemble des risques systèmes. Trois limites méritent d'être connues.

Isolation OS du sous-processus

Le mode `subprocess` isole le plugin du processus serveur en mémoire, mais ne lui impose pas un bac à sable OS : il a accès au réseau, et au système de fichiers au-delà de son `cwd` (que rien n'empêche, sous Linux, de quitter par chemin absolu). Le jail applicatif (`ctx.resolve_path`) protège les builtins du serveur, pas le sous-processus, qui exécute du code Python natif. Sur **Windows**, la situation est identique : le sous-processus n'est pas isolé par un job object ou un conteneur.

Une isolation plus stricte (conteneur Linux léger, job object Windows, sandbox-exec sur macOS) constitue un chantier ouvert. Tant qu'elle n'est pas mise en place, on recommandera de ne déclarer `tier = high` qu'à des auteurs dont le code lui-même est considéré comme inoffensif — la garantie cryptographique ne fait que prouver l'origine, pas l'innocuité.

Modes privilégiés : code à privilèges serveur

Le mode `inproc` (Python) et l'option `env: true` (VScript) exécutent leur code avec les **privilèges du processus serveur**. Toute la discipline cryptographique précédente vise à garantir que ce privilège n'est accordé qu'à un code dont l'origine est certaine ; elle ne contrôle pas ce que ce code fait. La double garde est donc, in fine, une **décision humaine** déléguée à l'administrateur du serveur (qualité de la clé) et à celui du workspace (acceptation des privilèges) ; le système assure la cohérence de cette décision, pas sa pertinence.

Désactivation propre en l'absence de cryptography

Le sous-système requiert la bibliothèque Python `cryptography` (primitives Ed25519). Si cette dépendance est absente d'un build figé — par exemple, dans un déploiement embarqué sans accès à PyPI —, le sous-système se **désactive proprement** : `[plugin-install]`, `[plugin-load]` et l'ensemble des commandes `plugin-trust-*` lèvent une `PluginsUnavailableError` explicite. Aucun mécanisme de fallback n'autorise l'exécution d'un plugin sans vérification — l'indisponibilité de la vérification est traitée comme un échec dur, jamais comme une licence d'exécution.

B.15. Carte des fichiers du sous-système

À titre de référence pour les développeurs du cœur de VAILS, le tableau ci-dessous recense les fichiers principaux impliqués dans le sous-système plugin. La colonne de droite indique brièvement le rôle de chaque fichier ; les détails du moteur VScript (vendorisé depuis VDOM) sont traités à l'Annexe C.

Côté plugins

Fichier	Rôle
`builtin/plugins.py`	Implémentation de l'ensemble des keywords du sous-système : magasin de confiance (**`plugin-trust-*`**), cycle de vie (**`plugin-install`**, **`plugin-remove`**, **`plugin-load`**, **`plugin-unload`**), introspection (**`plugins-list`**, **`plugin-info`**).
`vvl_plugin_sdk.py`	SDK auteur, expose **`serve`**, **`ok`**, **`fail`**. Distribué avec les bundles ou installé côté auteur.
`scripts/vvl-plugin-sign.py`	Outil hors-ligne : sous-commandes **`keygen`** (paire Ed25519) et **`sign`** (canonicalisation, calcul de digest, signature, écriture du bundle).
`src/vails/vvl_config.py`	Persistance : **`WorkspaceConfig.plugins_trusted`**, lecture/écriture du bloc **`<PLUGIN_TRUST>`**, fonctions du magasin (ajout, suppression, mise à jour atomique).
`src/vails/vvl_core.py`	Intégration moteur : champ **`ctx.plugins`**, branche de dispatch après **`BUILTINS`** et **`ctx.keywords`**, propagation par référence dans **`clone()`** et l'expansion de macro.

Figure B.11 — *Fichiers principaux du sous-système plugin — la séparation entre persistance (vvl_config.py), runtime (vvl_core.py) et keywords (builtin/plugins.py) est strictement maintenue.*

Côté VScript (extrait — détails en Annexe C)

Fichier	Rôle
`vscript/`	Cœur du langage vendorisé depuis le runtime VDOM (branche **`dev_py3`**, moteur 1.3) : lexer/parser PLY, subtypes, library, engine, plus les extensions **`regexp`**, **`xml`** (réécrite en XPath) et **`vails_ext`**. Voir **`vscript/PATCHES.md`**.
`src/vails/vvl_vscript.py`	Façade VAILS : compile-une-fois + cache des programmes VScript, marshalling VVL ↔ VScript, point d'entrée **`run_vscript(host=…)`**.
`src/vails/vvl_vscript_env.py`	Objets hôtes injectés au moment du chargement quand **`env: true`** : **`ws.files`**, **`ws.csv`**, **`ws.db`**, **`ws.mem`**, **`ws.log`**, **`ws.vvl`**.

Figure B.12 — *Fichiers principaux du sous-système VScript — la façade VAILS isole le cœur vendorisé du reste du serveur.*

Le moteur VScript proprement dit, sa grammaire, sa sémantique d'évaluation, ses modernisations Tier 1 (littéraux tableau/dictionnaire, opérateur `??`, assignations composées, `IIf`, court-circuit d'`And`/`Or`) ainsi que la spécification complète de l'objet `ws` sont décrits à l'**Annexe C — VScript**. La présente annexe se limite à la plomberie de chargement et de garde qui s'applique de manière uniforme aux deux langages de plugin.

Annexe C — Le langage VScript

Mots-clés : VScript, [vscript], compile-une-fois, args, result, marshalling, bytecode CPython, sandbox structurel, lexical, types, opérateurs, classes, ws, interopérabilité OO, modernisations Tier 1, Tier 3

VScript est le langage de script embarqué dans VAILS. Hérité du runtime VDOM dont il est vendorisé (paquet `vscript/`, branche `dev_py3`, moteur 1.3), il offre à l'auteur VVL un second langage à syntaxe impérative — proche de VBScript mais **distinct**, on ne l'appelle jamais « VBScript » — qui se prête particulièrement à l'écriture de routines de calcul, de manipulation de chaînes, de traitement XML ou de classes ad hoc, là où l'expressivité purement homoiconique de VVL deviendrait verbeuse.

Cette annexe couvre l'intégralité du langage et de son intégration dans VAILS : la structure lexicale et les types, les opérateurs et le contrôle de flux, les procédures et l'orienté objet, la bibliothèque standard et les deux extensions embarquées (`RegExp`, XML), puis les particularités d'intégration côté VVL — `[vscript "…"]` inline, libraries partagées via `[use …]`, objet hôte `ws`, et enfin l'interopérabilité OO qui fait communiquer instances VVL et instances VScript comme des **proxys vivants**. La chaîne de chargement des plugins écrits en VScript (signature Ed25519, install, double garde, `env: true`) reste traitée à l'**Annexe B — Système de plugins** et n'est pas reprise ici.

> **Note —** L'annexe est volontairement référentielle plutôt que tutorielle. Pour des exemples exécutables, on consultera les huit tutoriels `vvl/vscript/01..08` distribués avec VAILS — chaque chapitre cite explicitement le ou les tutoriels qui l'illustrent.

C.1. Nature et origine du langage

VScript ne descend pas d'une lignée Microsoft VBScript classique : il est issu de VDOM (Visual Dynamic Object Model), un environnement applicatif web français des années 2000 dont VAILS hérite par filiation directe. Son moteur a été **vendorisé** dans le sous-paquet `vscript/` du serveur, puis porté Python 3 sur la branche `dev_py3` (moteur 1.3). Les patchs de portage, qui touchent essentiellement l'unification chaînes/bytes et le décodage Unicode, sont documentés dans `vscript/PATCHES.md` ; la spécification de facto du comportement attendu est portée par la suite de tests `vscript/tests/` (1269 cas verts).ynamic

Une transpilation, pas un interpréteur arbre-marchant

À la différence d'un interpréteur traditionnel qui parcourt un arbre de syntaxe à chaque évaluation, VScript suit un pipeline en quatre étapes (figure C.1). Le code source est d'abord lexé puis parsé par PLY (Python Lex-Yacc) selon la grammaire définie dans `syntax.py` / `lexemes.py`. L'AST obtenu est ensuite composé en source Python équivalente, qui est passée à `compile()` pour produire un **code object CPython natif** (du bytecode). C'est ce bytecode — et lui seul — qui est conservé dans la référence appelable retournée par `[vscript "…"]`.



Figure C.1 — *Pipeline d'exécution VScript — les quatre étapes lexer/parser/composition/compile() ne sont parcourues qu'une fois ; chaque invocation [$ref a b] ne ré-exécute que la dernière étape (exec).*

Sandbox par construction

La vendorisation a été sélective : seuls le **cœur du langage** et deux extensions (`regexp` et une API XML pratique réécrite en XPath) sont embarqués. Les wrappers ASP/VDOM historiques — `Server`, `Request`, `Response`, `Session`, `Databases`, `Mailing`, `FreePDF`, … — n'ont pas été repris. Conséquence directe et essentielle : **aucune primitive d'entrée/sortie n'est exposée par le langage**. Pas de lecture ou d'écriture de fichier, pas de socket réseau, pas d'invocation de processus, pas d'accès au système d'exploitation. Tout accès à l'environnement doit passer par l'objet hôte `ws` (chapitre C.23), injecté par la façade VAILS.

> **Limite —** Une **unique exception** mérite d'être connue : le caractère backtick ( ) injecte du Python brut jusqu'à la fin de ligne. C'est une surface de sécurité à part entière — réservée aux administrateurs du serveur, à éviter dans tout code soumis par un utilisateur. Elle n'est mentionnée ici que pour exhaustivité.

C.2. Les deux portes d'entrée dans VAILS

VScript est accessible depuis VVL par deux portes d'entrée, qui partagent toute la mécanique d'exécution (le même moteur, le même bytecode) mais diffèrent par leur durée de vie et leur contexte de sécurité.

Forme inline — [vscript "code"]

La forme **inline** est un keyword VVL : `[vscript "code"]` compile le code une seule fois et renvoie une **référence appelable** (`VVL_BoundMethod`, sur le modèle de `[lambda]`). Cette référence se stocke dans une variable VVL et se réinvoque ensuite avec la syntaxe d'appel ordinaire.

```
$double = [vscript "result = args(0) * 2"]
[$double 21]                          # => 42  (compilé à la création de $double)
[$double 100]                         # => 200 (réutilise la référence, pas de recompilation)
$sum = [vscript "result = args(0) + args(1) + args(2)"]
[$sum 10 20 12]                       # => 42
```

Le code inline s'exécute **in-process**, dans le même processus que le serveur, au même titre que `[evaluate]` ou `[lambda]`. Il bénéficie donc d'un accès complet à l'objet hôte `ws` (le code inline est, par construction, du code serveur — voir le tutoriel `01-inline-refs.vvl`).

Forme plugin — lang: "vscript"

La seconde porte d'entrée est le **plugin signé**. Un plugin dont le manifeste déclare `lang: "vscript"` voit son fichier d'entrée (un `.vscript`) compilé au moment du chargement (`[plugin-load]`) puis réutilisé à chaque invocation. Tout le reste — signature Ed25519, vérification anti-TOCTOU, magasin de clés `<PLUGIN_TRUST>`, idiomes `[plugin-install]` / `[plugin-load]`, double garde `env: true` pour exposer `ws` à un plugin VScript — relève de l'**Annexe B — Système de plugins** et n'est pas réintroduit ici.

Concrètement, un plugin VScript est signé par l'outil habituel :

```
python scripts/vvl-plugin-sign.py sign --key author.pem --key-id prosuite-2026 \
    --name greet --lang vscript --entry greet.vscript \
    --arity-min 1 --arity-max 1 --out greet.vvlplugin
```

Et utilisé côté workspace de manière transparente :

```
[plugin-install "greet.vvlplugin"]
[plugin-load "greet" as greet]
[greet "world"]                       # => Hi world!  (greet.vscript: result = "Hi " &
args(0) & "!")
```

> **Note —** Un plugin VScript s'exécute **in-process** comme la forme inline, mais **sans la double garde** requise par le mode `inproc` Python : le langage étant sandboxé par construction, aucune élévation de privilège n'est en jeu. L'accès à `ws` reste, lui, conditionné à `env: true` + double garde (voir Annexe B).

C.3. Contrat d'appel et marshalling VVL ↔ VScript

L'interface entre VVL et le code VScript repose sur un contrat minimal en deux noms réservés : `args` pour l'**entrée** et `result` pour la **sortie**.

Entrée : la variable args, indexation 0-based

Lorsqu'une référence est invoquée avec `[$ref a b c …]`, le serveur empile les arguments évalués dans une variable `args` accessible au code VScript. L'indexation est **0-based** : `args(0)` est le premier argument, `args(1)` le deuxième, et ainsi de suite. Le nombre d'arguments effectifs est connu via `UBound(args)` (qui retourne l'indice maximum, donc `nb_args - 1`) ou via la variable d'aide `argc` (synonyme `UBound(args) + 1`).

> **Note —** Attention au piège classique : VScript est globalement **1-based** pour les chaînes (`Mid(s, 1, 2)` lit deux caractères à partir du premier), mais `args` et les tableaux littéraux `[…]` sont **0-based**. Le mélange est cohérent avec la convention historique de VBScript et n'est pas négociable.

Sortie : la variable result

Le code VScript pose sa valeur de retour dans la variable `result`. Si le résultat est un **objet** (une instance de classe créée par `New`, par exemple), l'affectation doit utiliser `Set` :

```
' Valeur scalaire : affectation simple
result = "Hi " & args(0) & "!"

' Objet : Set obligatoire
Set p = New Point
p.Init args(0), args(1)
Set result = p
```

L'absence de `Set` lors de l'affectation d'un objet à `result` est une **erreur d'exécution** ; cette discipline est héritée de VBScript et reste invariante. Si `result` n'est jamais affecté, la référence retourne `Empty`.

Marshalling automatique

La frontière VVL ↔ VScript est traversée avec **conversion automatique de types** : aucune sérialisation explicite n'est nécessaire. Le tableau ci-dessous synthétise les correspondances.

Côté VVL	Côté VScript	Sens du marshalling
`String`	`String`	Bidirectionnel, identique.
`Integer` / `Float`	`Integer` / `Double`	Bidirectionnel. VScript distingue `Integer` et `Double` ; VVL aligne sur ses propres types numériques.
`Boolean`	`Boolean`	Bidirectionnel. Vrai VScript = `-1` côté numérique ; le marshalling le ramène à `true` VVL.
Liste VVL	`Array` VScript	Bidirectionnel. L'accès par indice est conservé (0-based des deux côtés).
Dictionnaire VVL	`Dictionary` VScript	Bidirectionnel. L'accès par clé chaîne est conservé.
Instance VVL	Proxy vivant	Voir C.24 — l'objet n'est pas copié, il est exposé par référence.
`Null` / `Nothing`	`Null` / `Empty` / `Nothing`	VScript distingue trois absences ; VVL en présente une seule (`null`).

Figure C.2 — *Correspondances de types entre VVL et VScript — l'opération est transparente pour les scalaires et les collections, et préserve l'identité pour les objets (voir D.24).*

Le marshalling se fait à **chaque entrée et sortie** de la référence : modifier un élément du tableau côté VScript ne modifie pas la liste VVL d'origine — sauf pour les objets (instances de classe), qui traversent la frontière sous forme de **proxys vivants** dont les mutations sont reflétées des deux côtés (voir le chapitre C.24 pour le détail de cette sémantique de référence partagée).

C.4. Structure lexicale

La structure lexicale de VScript est volontairement traditionnelle, dans la lignée de la famille Basic. Les éléments suivants en constituent l'ossature normative.

Élément	Règle
Casse	Le langage est **insensible à la casse** : `Total`, `total` et `TOTAL` désignent le même identifiant, et les mots-clés s'écrivent indifféremment `if`, `If`, `IF`. Cette insensibilité s'étend aux membres d'instance (voir C.24 pour ses implications dans l'interopérabilité OO).
Commentaires	Une apostrophe `'` introduit un commentaire jusqu'à la fin de la ligne. La forme alternative `Rem …` est acceptée pour compatibilité historique. Il n'existe **pas** de syntaxe de commentaire de bloc.
Continuation de ligne	Un caractère _ (souligné) placé en fin de ligne joint physiquement cette ligne à la suivante au moment du lexing. Utile pour aérer une expression longue sans rompre son intégrité syntaxique.
Séparateurs d'instructions	Le saut de ligne sépare les instructions ; le caractère `:` permet d'en aligner plusieurs sur une même ligne (`a = 1 : b = 2 : c = 3`).
Identifiants	Les identifiants respectent `[A-Za-z][A-Za-z0-9_]*`. Aucun caractère étendu Unicode n'est admis en première position.
Backtick — Python brut	Le caractère introduit une ligne de **Python brut** qui sera insérée telle quelle dans la composition (C.1). Surface de sécurité majeure — usage réservé aux administrateurs.

Figure C.3 — *Éléments lexicaux fondamentaux de VScript — la casse insensible et la convention de commentaire restent les deux particularités qui surprennent le plus un développeur venant de Python ou de Java.*

C.5. Littéraux et constantes intrinsèques

VScript admet une palette complète de littéraux, des plus classiques (entiers, chaînes, dates) aux ajouts récents des modernisations Tier 1 (tableaux, dictionnaires) et Tier 3 (hexadécimaux et binaires numériques, voir le tutoriel `08-tier3.vvl`).

Littéraux numériques

Trois formes sont disponibles : entiers décimaux (`42`, `-7`), nombres à virgule flottante en notation classique ou scientifique (`3.14`, `1.5e3`), et — depuis Tier 3 — littéraux hexadécimaux et binaires préfixés respectivement par `&H` et `&B` :

```
x = &H1F                          ' x = 31  (hexadécimal)
y = &B1010                        ' y = 10  (binaire)
z = &HFF + &B10                   ' z = 257 (mélangeable, conversion immédiate)
```

> **Note —** Une régression historique de l'implémentation initiale produisait un **caractère** (via `Chr()`) à partir de `&H…`. La modernisation Tier 3 a rétabli la sémantique numérique — désormais `&H1F` est strictement l'entier 31.

Littéraux chaîne et chaîne interpolée

La forme classique `"texte"` admet un seul échappement : le guillemet double interne se redouble (`"il dit ""oui"""` → `il dit "oui"`). En particulier, **`\n` et `\t` ne sont pas interprétés** dans cette forme — les backslashes restent littéraux, ce qui préserve les chemins Windows tels quels (`"C:\Users\…"`).

La forme moderne — chaîne interpolée — s'écrit `$"texte"` et autorise à la fois l'insertion d'expressions entre accolades et un jeu d'échappements C-like (`\n`, `\t`, `\r`, `\\`, `\"`). Voir le chapitre D.6 pour le détail.

Littéraux dates

Une date littérale est encadrée par des dièses : `#2026-05-22#` ou `#10:30:00#`. Aucune ambiguïté de format locale ne s'introduit puisque la grammaire impose un format ISO interprété sans dépendance régionale.

Littéraux tableau et dictionnaire (Tier 1)

Les modernisations Tier 1 (VAILS 5.18 / VVL 4.12) ont ajouté deux familles de littéraux composites, qui rendent inutile la combinaison `Array()` + `Dictionary()` pour construire des données :

```
a = [10, 20, 30]                    ' tableau, accès a(0) .. a(2)
d = {"id": 7, "name": "z"}          ' dictionnaire, accès d("id"), d("name")
rows = [{"id": 1}, {"id": 2}]       ' tableau de records (composition naturelle)

' Dans les API ws/db : passer un record en une seule expression
ws.csv("clients").add {"id": "7", "name": "z"}
```

> **Note —** Ces littéraux s'affectent avec `=` (et **non** `Set`), comme tout littéral non-objet. L'itération `For Each k, v In d` (chapitre D.12) déstructure un dictionnaire en paires clé/valeur. Voir le tutoriel `03-modern-features.vvl`.

Constantes intrinsèques

VScript expose les familles de constantes nommées héritées de VBScript, sous la forme d'identifiants préfixés `vb*`. Elles sont toujours disponibles, n'ont pas à être importées, et restent **insensibles à la casse** comme tout identifiant.

Famille	Constantes
Chaînes utilitaires	**`vbCrLf`**, **`vbCr`**, **`vbLf`**, **`vbNewLine`**, **`vbTab`**, **`vbVerticalTab`**, **`vbFormFeed`**, **`vbNullChar`**, **`vbNullString`**
Booléens classiques	**`vbTrue`** (−1), **`vbFalse`** (0)
Comparaison	**`vbBinaryCompare`** (0), **`vbTextCompare`** (1), **`vbDatabaseCompare`** (2)
Formats de date	**`vbGeneralDate`**, **`vbLongDate`**, **`vbShortDate`**, **`vbLongTime`**, **`vbShortTime`**
Jours de semaine	**`vbSunday`** … **`vbSaturday`**, **`vbUseSystemDayOfWeek`**
Première semaine	**`vbFirstJan1`**, **`vbFirstFourDays`**, **`vbFirstFullWeek`**, **`vbUseSystem`**, **`vbUseDefault`**

Figure C.4 — *Familles de constantes intrinsèques — leur présence est garantie par la bibliothèque standard, sans import.*

C.6. Interpolation de chaîne et formatage par pipe

L'**interpolation de chaîne** (`$"…"`) est une modernisation majeure qui mérite un chapitre dédié : elle remplace avantageusement la concaténation `&` répétée et offre un mécanisme de formatage déclaratif par chaîne de fonctions.

Forme générale

À l'intérieur d'une chaîne `$"…"`, toute occurrence de `{expr}` est substituée par la valeur de l'expression VScript qu'elle contient. Le résultat est **toujours une chaîne** — les conversions implicites (`CStr`) sont appliquées. Les accolades littérales s'obtiennent par doublement : `{{` et `}}`.

```
msg = $"Bonjour {name}, vous avez {count} message(s)"
total = $"{a} + {b} = {a + b}"              ' expression complète, conversion auto

mxn = $"matrice {{r,c}} de taille {n}x{n}"   ' produit : matrice {r,c} de taille 3x3
```

Échappements C-like — uniquement dans $"..."

Les séquences `\n`, `\t`, `\r`, `\\` et `\"` sont reconnues uniquement à l'intérieur des chaînes interpolées. Cette différence est délibérée : les chaînes classiques `"..."` conservent les backslashes littéraux, ce qui évite d'avoir à doubler chaque séparateur de chemin Windows.

Formatage par pipe — { expr | f | g(args) }

Le mécanisme le plus distinctif de l'interpolation est le **pipe** `|` : à l'intérieur d'une accolade `{ … | f | g(arg) }`, la valeur de gauche est passée en **premier argument** de chaque fonction listée à droite, dans l'ordre. La chaîne est arbitrairement longue ; le résultat de chaque étape devient la valeur d'entrée de la suivante.

```
prix  = $"prix: {price | FormatCurrency}"        ' FormatCurrency(price)
taux  = $"{ratio | FormatPercent(1)}"            ' FormatPercent(ratio, 1)
nom   = $"{name | Trim | UCase}"                 ' UCase(Trim(name))
nbr   = $"{x | FormatNumber(2)}"                 ' FormatNumber(x, 2)
hex   = $"hex={n | Hex}"                         ' Hex(n)
```

Tout identifiant nommant une fonction VScript visible à l'endroit de l'interpolation est admissible — pas seulement les `Format*` de la bibliothèque standard. Une fonction définie par l'utilisateur convient également : `{ data | normalize | summarize }`.

> **Note —** Limites de la v1. À l'intérieur d'une accolade `{expr}`, on ne peut pas inclure d'accolade fermante littérale (sauf par échappement `}}`), ni d'accolade imbriquée. Pour un cas complexe, on **extrait** l'expression dans une variable temporaire évaluée avant l'interpolation.

C.7. Subtypes et tests de type

À la manière de VBScript, VScript n'a qu'un seul type de variable au niveau lexical (le `Variant`) ; ce qui distingue dynamiquement deux valeurs est leur subtype, accessible à l'introspection. Le tableau ci-dessous recense les subtypes observables.

Subtype	Description	Test idiomatique
`Integer`	Entier signé (taille machine).	`IsNumeric(x) + TypeName(x) = "Integer"`
`Double`	Flottant double précision IEEE 754.	`IsNumeric(x) + TypeName(x) = "Double"`
`String`	Chaîne UTF-8 immutable.	`TypeName(x) = "String"`
`Boolean`	Vrai (−1 numérique) ou faux (0).	`TypeName(x) = "Boolean"`
`Date`	Instant ou date pure, encodage interne homogène.	`IsDate(x)`

Subtype	Description	Test idiomatique
`Array`	Tableau dimensionné (créé par `Array()`, `[…]`, ou `Dim a(n)`).	`IsArray(x)`
`Dictionary`	Table associative ordonnée clé → valeur.	`IsDictionary(x)`
`Object`	Instance de classe (utilisateur ou hôte).	`IsObject(x)`
`Empty`	Variable déclarée mais jamais affectée.	`IsEmpty(x)`
`Null`	Absence de valeur déclarée (≠ `Empty`).	`IsNull(x)`
`Nothing`	Référence d'objet non liée (après `Set x = Nothing`).	`IsNothing(x)` ou `x Is Nothing`

Figure C.5 — *Subtypes observables — les trois absences (Empty, Null, Nothing) jouent des rôles distincts : Empty pour les variables non initialisées, Null pour les valeurs explicitement « absentes », Nothing pour les références d'objet libérées.*

Les fonctions d'introspection `TypeName(x)` (chaîne du nom de subtype) et `VarType(x)` (code numérique historique) restent disponibles. Les versions du moteur sont exposées via `ScriptEngine`, `ScriptEngineMajorVersion`, `ScriptEngineMinorVersion`, `ScriptEngineBuildVersion` — utiles pour conditionner du code sur la version du moteur vendorisé.

C.8. Variables, constantes et déclarations

La déclaration explicite des variables n'est pas obligatoire en VScript : toute affectation à un identifiant non préalablement déclaré crée la variable à la volée. Néanmoins, la déclaration via `Dim` est recommandée pour la lisibilité et pour documenter l'intention — notamment dans les classes (D.14), où les champs **doivent** impérativement être déclarés par `Dim`.

Dim, ReDim, Const, Erase

La déclaration porte sur une variable simple, ou sur un tableau dimensionné. Le `ReDim` permet de redimensionner dynamiquement, avec la variante `Preserve` qui conserve le contenu existant. `Erase` réinitialise. `Const` introduit une constante figée pour la durée du module.

```
Dim a, b, total                    ' déclaration simple, multi-variable
Dim grid(9), m(3, 3)               ' tableaux 1D et 2D dimensionnés
ReDim arr(10)                      ' redimensionne (perd le contenu)
ReDim Preserve arr(20)             ' redimensionne en gardant les éléments existants
Erase arr                          ' réinitialise à un tableau vide
Const PI = 3.14159                 ' constante figée
```

Affectation par valeur vs Set : la règle d'or

La distinction la plus piégeuse pour un développeur venant d'un langage moderne est la différence entre l'affectation par valeur (`=`) et l'affectation d'objet (`Set`). Toute valeur **non-objet** s'affecte avec `=` ; toute valeur **objet** (instance créée par `New`, retournée par une fonction, exposée par l'hôte) s'affecte obligatoirement avec `Set`.

```
Dim n : n = 42                     ' OK — entier, affectation simple
Dim s : s = "hello"                ' OK — chaîne
Dim a : a = [1, 2, 3]              ' OK — tableau (littéral, non-objet)

Dim obj
Set obj = New MyClass              ' OBLIGATOIRE — instance de classe
Set rs  = ws.db("app").query("…")  ' OBLIGATOIRE — collection objet retournée par ws
obj = New MyClass                  ' ERREUR — affectation d'objet sans Set
```

> **Note —** Cette discipline est héritée de VBScript et ne sera pas relâchée. Elle se traduit aussi du côté de la frontière VVL/VScript : pour renvoyer un objet à VVL via la variable `result`, on écrit toujours `Set result = obj` (voir D.3 et D.24).

Passage de paramètres : ByVal et ByRef

Les paramètres des procédures (`Sub`, `Function`) sont **`ByRef` par défaut** — c'est la convention historique. Une procédure peut donc modifier une variable passée par l'appelant, ce qui peut surprendre quand on attend une copie. Le qualificatif `ByVal` impose explicitement le passage par valeur.

```
Sub Bump(ByRef n)                   ' par défaut : modification visible chez l'appelant
  n = n + 1
End Sub

Sub Compute(ByVal x)                ' explicite : copie locale
  x = x * 2                         ' modification invisible chez l'appelant
End Sub

Dim k : k = 5
Bump k                              ' k vaut maintenant 6
Compute k                           ' k vaut toujours 6
```

On retiendra la règle pratique : en cas de doute, préfixer explicitement `ByVal` — cela évite une classe entière de bugs subtils où une procédure utilitaire mute sans le vouloir une variable de l'appelant.

C.9. Opérateurs et précédence

Le tableau de précédence ci-dessous est exhaustif et normatif : il liste, du plus prioritaire au moins prioritaire, l'ensemble des opérateurs reconnus par la grammaire. Toute expression peut être désambiguïsée par parenthésage explicite.

Niveau	Opérateurs	Catégorie
1	`^`	Puissance (exposant réel).
2	`+x -x`	Unaires (signe).
3	`* /`	Multiplication, division — `/` **retourne toujours un Double**.
4	`\`	Division **entière** (le résultat est un entier tronqué).
5	`Mod`	Modulo (entier).
6	`+ -`	Addition, soustraction.
7	`&`	Concaténation de chaînes — toujours une chaîne, conversions auto.
8	`= <> < > <= >= Is`	Comparaisons (`=` selon contexte ; `Is` = identité d'objet).
9	`Not`	Négation logique.
10	`And`	Conjonction logique (avec court-circuit numérique, voir D.10).
11	`Or`	Disjonction logique (idem).

Niveau	Opérateurs	Catégorie
12	`Xor`	Ou exclusif.
13	`??`	Coalescing — **l'opérateur le plus faible** (voir D.11).

Figure C.6 — *Précédence des opérateurs — la division entière \ et le modulo Mod sont distincts de la division flottante / (qui produit toujours un Double même sur des entiers).*

Trois points méritent un soulignement particulier. **Premièrement**, le signe `=` est syntaxiquement polymorphe : à gauche d'une affectation il affecte, à l'intérieur d'une expression il compare. La grammaire désambiguïse par contexte sans possibilité de confusion. **Deuxièmement**, `<>` est l'opérateur d'inégalité — il n'existe pas de `!=`. **Troisièmement**, `Is` est l'opérateur d'**identité d'objet** (deux références désignent-elles la même instance ?), à ne pas confondre avec `=` qui compare par valeur — sur les objets, `=` est généralement non défini et peut lever.

Indexation : la convention 0-based vs 1-based

L'indexation des collections est **0-based** : tableaux (`a(0)`, `a(1)`, …), dictionnaires par chaîne de clé (`d("name")`), et la variable `args` (`args(0)` est le premier argument). En revanche, les fonctions de manipulation de chaînes historiques sont **1-based** : `Mid(s, 1, 2)` lit deux caractères à partir du premier caractère. Cette dichotomie ne doit pas être interprétée comme une incohérence — c'est l'alignement strict sur la convention VBScript que VScript préserve.

C.10. Court-circuit logique And et Or

Une des **déviations assumées** de VScript par rapport à VBScript concerne l'évaluation des opérateurs `And` et `Or`. VBScript évalue toujours les deux opérandes, ce qui interdit l'idiome « tester d'abord, déréférencer ensuite » fréquent dans les langages modernes. VScript ajoute un **court-circuit conditionnel** qui résout ce problème sans casser la logique trivaluée héritée.

La règle exacte — court-circuit limité aux gauches numériques

Le court-circuit s'active **uniquement** lorsque l'opérande gauche est numérique et déterminant : `0` pour `And` (rien ne peut être vrai), `-1` pour `Or` (tout est déjà vrai). Dans ces deux cas, l'opérande droit n'est pas évalué. Pour toute autre gauche — booléen (`True` / `False`), `Null`, `Empty`, chaîne, expression de comparaison comme `n <> 0` — l'évaluation est **complète**, ce qui préserve la logique à trois valeurs autour de `Null`.

```
1  Dim n : n = args(0)
2  Dim msg
3  If n And (100 \ n) > 0 Then          ' n = 0 : court-circuit, 100\n jamais évalué
4    msg = "nonzero"                     ' n = 5 : évaluation complète, 100\5 = 20 > 0
5  Else
6    msg = "zero (droite ignorée)"
7  End If
8  result = msg
9
10 ' Test n=0  -> "zero (droite ignorée)"   pas de division par zéro
11 ' Test n=5  -> "nonzero"
```

> **Limite —** Si l'on écrit `(n <> 0) And (100 \ n) > 0`, l'opérande gauche est un booléen — la gauche n'est pas considérée comme « numérique déterminant », et la division par zéro réapparaît. C'est l'idiome paradoxal et utile à connaître : pour bénéficier du court-circuit, on teste directement la valeur entière (`n`), pas la comparaison.

C.11. Coalescing ?? et optional chaining ?.

Deux opérateurs modernes — empruntés à la famille C# / TypeScript — rendent l'écriture d'expressions tolérantes aux absences considérablement plus directe que les imbrications de `If … Then` traditionnelles.

Coalescing ?? — valeur par défaut paresseuse

L'expression `a ?? b` retourne `a` sauf si `a` vaut `Null` ou `Empty`, auquel cas elle retourne `b`. La **droite est paresseuse** : `b` n'est évalué que si nécessaire. L'opérateur est associatif à droite, ce qui permet d'enchaîner naturellement plusieurs défauts.

```
name = user.name ?? "anonyme"
addr = profile.address ?? user.address ?? "adresse inconnue"

' Précédence : ?? est le plus faible, donc le parenthésage est rarement nécessaire
total = a + b ?? 0                    ' = (a + b) ?? 0
```

Optional chaining ?. — accès null-safe par maillon

L'expression `obj?.member` retourne `Empty` au lieu de lever si `obj` est `Null`, `Empty` ou `Nothing`. L'opérateur est **chaînable** mais doit être posé sur **chaque maillon** potentiellement nul — la sémantique est par maillon, et non globale à l'expression.

```
' Sécurisation de chaque maillon
city = customer?.address?.city ?? "inconnue"

' Erreur subtile : seul a est gardé contre Null ; si b est Null, b.c lève
risky = a?.b.c

' Forme correcte : ?. à chaque maillon
safe  = a?.b?.c
```

L'opérateur `?.` ne garde que contre les **trois absences** (`Null`, `Empty`, `Nothing`) — il ne capture pas les erreurs de type (par exemple, accéder à un `member` qui n'existe pas sur un objet réel lève normalement). Le point classique `.` reste inchangé et conserve sa sémantique stricte. Voir le tutoriel `03-modern-features.vvl` (exemple 8) pour un cas d'usage avec une classe `Acct`.

C.12. Contrôle de flux

VScript offre un éventail complet de structures de contrôle, allant des formes classiques `If/Then/Else` à `Select Case`, plusieurs variantes de boucles, et — pour les sorties — `Exit` ainsi que le `Return` moderne (chapitre D.13).

Conditionnelles : If et Select Case

La forme `If` admet trois variantes : en ligne (une instruction sur la même ligne), en bloc avec `Then` seul (terminé par `End If`), et la forme étendue `If/ElseIf/Else`. La forme `Select Case` regroupe par valeur — elle accepte plusieurs valeurs par branche (`Case 1, 2, 3`) et une clause finale `Case Else`.

```
' If en ligne
If x > 0 Then result = "pos"
If x > 0 Then result = "pos" Else result = "neg"

' If en bloc
If x > 0 Then
  result = "pos"
ElseIf x = 0 Then
  result = "zero"
Else
  result = "neg"
End If

' Select Case — équivalent à une chaîne de If/ElseIf
Select Case n
  Case 1
    result = "un"
  Case 2, 3
    result = "deux ou trois"
  Case Else
    result = "autre"
End Select
```

Boucles : For, For Each, Do, While

Trois familles de boucles coexistent. **`For i = a To b Step s`** itère un compteur numérique entre deux bornes (le `Step` est optionnel et vaut `1` par défaut, ou peut être négatif pour itérer à rebours). **`For Each item In collection`** itère sur les éléments d'un tableau, d'un dictionnaire ou d'une collection retournée par l'objet `ws`. La variante **`For Each k, v In dict`** (Tier 3) déstructure un dictionnaire en paires clé/valeur. Les boucles **`Do ... Loop`** existent en variantes pré-test (`Do While`, `Do Until`) et post-test (`Loop While`, `Loop Until`). La forme historique `While … Wend` reste disponible.

```
' For numérique
For i = 1 To 10 Step 2 : Log i : Next       ' 1 3 5 7 9

' For Each sur un tableau
For Each item In [10, 20, 30] : total = total + item : Next

' For Each k, v sur un dictionnaire (Tier 3)
Dim d : d = {"a": 1, "b": 2, "c": 3}
For Each k, v In d
  out = out & k & "=" & CStr(v) & " "
Next                                          ' out = "a=1 b=2 c=3 "

' Do While / Do Until
Do While n > 0 : n = n - 1 : Loop
Do Until done : ... : Loop

' Post-test (au moins une exécution garantie)
Do : process() : Loop While more
```

Sorties anticipées : Exit et Return

Chaque structure de boucle ou de procédure dispose d'une instruction de sortie anticipée dédiée — `Exit For`, `Exit Do`, `Exit Function`, `Exit Sub`, `Exit Property`. La syntaxe moderne `Return` (chapitre D.13) sort d'une `Function` ou d'une `Sub` avec une éventuelle valeur de retour.

```
For Each row In rs
  If row("id") = target Then
    found = True
    Exit For                          ' interrompt la boucle dès la première occurrence
  End If
Next

Function Find(items, target)
  For Each it In items
    If it = target Then Return True
  Next
  Return False                        ' équivalent moderne de Find = False : Exit Function
End Function
```

C.13. Procédures : Sub, Function, Return moderne

VScript distingue deux familles de procédures : les **`Sub`** (qui ne retournent pas de valeur) et les **`Function`** (qui en retournent une). La syntaxe d'appel diffère : une `Sub` s'appelle directement par son nom (avec ou sans `Call`), une `Function` s'utilise dans une expression. Le retour de fonction obéit à deux conventions cohabitantes : la forme classique (assignation au nom de la fonction) et la forme moderne (`Return expr`).

Forme classique — nom_de_fonction = valeur

La convention héritée consiste à affecter la valeur de retour au nom de la fonction. Tant que la procédure n'a pas atteint son `End Function`, l'affectation peut être modifiée — c'est la dernière valeur affectée qui sera retournée. Pour sortir en cours de route, on combine avec `Exit Function`.

```
Function Square(x)
  Square = x * x                      ' valeur de retour
End Function

Function Fact(n)
  If n <= 1 Then
    Fact = 1
  Else
    Fact = n * Fact(n - 1)            ' récursion
  End If
End Function

Log Fact(5)                          ' 120
```

Forme moderne — Return [expr]

Le keyword `Return` est une **modernisation Tier 1**, sucre syntaxique sur la forme classique : `Return expr` est strictement équivalent à `nom_de_fonction = expr` suivi de `Exit Function`, en atomique. Le bénéfice est double : on évite l'oubli classique du `Exit`, et on n'a plus à répéter le nom de la fonction à chaque point de retour.

```
Function Grade(n)
  If n < 0 Then Return "invalid"      ' pose le résultat ET sort
  If n >= 90 Then Return "A"
  If n >= 80 Then Return "B"
  Return "C"
End Function

Sub Notify(msg)
  If msg = "" Then Return             ' Return nu = Exit Sub
  ws.log msg
End Sub
```

> **Note —** Les deux formes peuvent coexister dans une même fonction ; la modernisation est purement additive et n'invalide aucun code existant. `Return` hors `Function` ou `Property Get` est une erreur de compilation.

Assignations composées — sucre arithmétique

Pour simplifier les schémas d'accumulation, les opérateurs `+= -= = /= \= &=** sont disponibles. La règle est mécanique : x op= e se réécrit en x = x op (e) — avec **parenthésage automatique de la droite**, qui évite les surprises de priorité.

```
Dim t : t = 0
t += 5                                ' t = 5
t *= 2                                ' t = 10
t -= 3                                ' t = 7

Dim s : s = "Hello"
s &= ", World"                        ' s = "Hello, World"

' Le parenthésage automatique de la droite :
t -= a + b                            ' équivaut à t = t - (a + b), pas t = t - a + b
```

C.14. Programmation orientée objet : Class et Property

VScript dispose d'un **système de classes complet** : déclaration avec `Class`, champs, méthodes, propriétés à accesseurs (`Property Get/Let/Set`), constructeur implicite et **héritage** par `Inherits`.

Déclaration d'une classe

La syntaxe canonique d'une classe combine champs (déclarés par `Dim` — jamais par `Private x` ou `Public x` au niveau variable, qui sont rejetés par la grammaire), méthodes (`Sub`/`Function`) et propriétés.

```
Class Counter
  Dim n                              ' champ (toujours déclaré par Dim)

  Public Sub Init(start)             ' « constructeur » conventionnel
    n = start
  End Sub

  Public Sub Bump()
    n = n + 1
  End Sub

  Public Property Get Value()        ' accesseur en lecture
    Value = n
  End Property
End Class

Set c = New Counter
c.Init 40
c.Bump
c.Bump
Log c.Value                        ' 42
```

> **Note —** VScript n'a **pas** de constructeur avec arguments à l'instanciation : `New Counter(40)` n'est pas légal. L'idiome canonique est d'utiliser `New` seul, puis d'appeler une méthode d'initialisation conventionnellement nommée `Init` (ou `Class_Initialize` historique, sans paramètre, exécutée automatiquement à `New`).

Propriétés — Get, Let, Set

Le bloc `Property` définit une propriété calculée avec des accesseurs distincts. `Property Get` retourne une valeur ; `Property Let` reçoit une valeur (assignation par `=`) ; `Property Set` reçoit une référence d'objet (assignation par `Set`). Les trois portent le même nom et peuvent coexister sur une même propriété.

```
Class Account
  Dim m_balance
  Dim m_owner

  Public Property Get Balance()
    Balance = m_balance
  End Property

  Public Property Let Balance(v)
    m_balance = v
  End Property

  Public Property Get Owner()
    Set Owner = m_owner              ' renvoyer un OBJET : Set obligatoire
  End Property

  Public Property Set Owner(o)
    Set m_owner = o                  ' recevoir un OBJET : Set obligatoire
  End Property
End Class
```

Héritage — Inherits, Me, MyBase, MyClass

L'héritage est mono-classe et déclaré par `Inherits` après le nom de la sous-classe. Trois pseudo-références sont disponibles dans le corps des méthodes : **`Me`** désigne l'instance courante (équivalent de `self` / `this`) ; **`MyBase`** désigne la classe parente (pour appeler une méthode redéfinie) ; **`MyClass`**

désigne la classe définissante de la méthode courante (utile pour ignorer une redéfinition dans une sous-classe).

```
Class Animal
  Dim name
  Public Sub Init(n) : name = n : End Sub
  Public Function Label() : Label = "[" & name & "]" : End Function
End Class

Class Dog : Inherits Animal
  Public Sub Bark()
    Log MyBase.Label & " woof!"    ' invoque Label() de la classe parente
  End Sub
End Class

Set d = New Dog
d.Init "Rex"
d.Bark                              ' [Rex] woof!
```

Le bloc With — économiser des répétitions

Le bloc `With obj` permet d'omettre le préfixe `obj.` pour tous les accès à membre placés à l'intérieur. C'est un raccourci syntaxique apprécié pour configurer un objet en série.

```
Set p = New Point
With p
  .X = 10
  .Y = 20
  .Init "origin"
End With
```

D.15. Enum — constantes nommées qualifiées

Le bloc `Enum … End Enum` (modernisation Tier 3) définit un ensemble de constantes nommées accessibles par qualification du nom de l'énumération : `EnumName.Member`. L'attribution est automatiquement incrémentielle à partir de zéro, mais peut être forcée sur n'importe quel membre — les membres suivants reprennent l'incrément à partir de la dernière valeur explicite.

```
Enum Color
  Red                               ' 0
  Green = 5                         ' 5
  Blue                              ' 6 (auto-incrément à partir du précédent)
End Enum

c = Color.Green                     ' 5
all = Color.Red & "/" & Color.Blue  ' "0/6"

' Accès indirect par chaîne (utile pour le pilotage par config)
v = Color("green")                  ' 5
```

L'accès par chaîne `Color("green")` retourne la même valeur que `Color.Green`. Cette forme indirecte est commode lorsque le membre est connu dynamiquement (lu depuis un fichier de configuration, par exemple) — elle évite une cascade de `Select Case`. Voir le tutoriel `08-tier3.vvl`.

D.16. Fonctions de première classe : AddressOf

L'opérateur `**AddressOf nom**` crée une référence appelable sur une fonction ou une sub nommée — référence qui peut ensuite être stockée dans une variable, passée en argument à une autre procédure, ou invoquée comme une fonction ordinaire. Cette mécanique (modernisation Tier 3) ouvre la voie au style fonctionnel d'ordre supérieur sans recourir à `Eval` ou à une indirection par chaîne.

```
Function Inc(x) : Inc = x + 1 : End Function
Function Double_(x) : Double_ = x * 2 : End Function

' Stockage et appel direct
Set fn = AddressOf Inc
Log fn(5)                          ' 6

' Passage à une fonction d'ordre supérieur
Function MapSum(fn, arr)
  Dim s : s = 0
  For Each it In arr
    s = s + fn(it)                 ' invocation indirecte
  Next
  MapSum = s
End Function

Log MapSum(AddressOf Inc,     [1, 2, 3])        ' 1+1 + 2+1 + 3+1 = 9
Log MapSum(AddressOf Double_, [1, 2, 3])        ' 2 + 4 + 6 = 12
```

> **Note —** `AddressOf` ne référence que des fonctions ou sub **nommées** au niveau supérieur ou dans une classe — les lambdas anonymes inline (de type `Function(x) … End Function` capturé dans une expression) ne sont pas encore couvertes par cette modernisation.

D.17. Gestion d'erreurs : Try / Catch / Finally / Throw

VScript adopte la gestion structurée d'erreurs (`Try/Catch/Finally/End Try`) plutôt que le modèle historique `On Error` de VBScript. La forme `On Error Resume Next` est **délibérément exclue** : elle masque les erreurs sans les traiter, pratique reconnue depuis longtemps comme une source de bugs durables. Cette exclusion est normative et ne sera pas réintroduite.

Forme générale

Le bloc `Try` regroupe le code surveillé ; un ou plusieurs blocs `Catch` interceptent les erreurs (avec liaison de variable et filtrage par type optionnel) ; un bloc `Finally` optionnel exécute du code de nettoyage quelle que soit l'issue.

```
Try
  result = SomeRiskyOperation()
Catch DivisionByZeroError As e
  Log "division par zéro : " & e.Message
Catch FileNotFoundError As e
  Log "fichier manquant : " & e.Path
Catch e                            ' attrape-tout (lié à e)
  Log "erreur inconnue : " & e.Message
Finally
  Cleanup()                        ' toujours exécuté
End Try
```

Catch typé et binding

Chaque clause `Catch` peut être typée (filtrage par classe d'erreur) et lier l'erreur capturée à une variable (`As e`). La résolution est séquentielle : la première clause dont le type correspond capture l'erreur ; une clause `Catch` sans type sert d'attrape-tout (à placer en dernier pour rester accessible).

Throw — lever ou relancer

`Throw expr` lève une erreur dont la valeur est `expr` (généralement une instance d'une classe d'erreur). `Throw` nu, à l'intérieur d'un bloc `Catch`, relance l'erreur courante telle quelle — pour la propager après un nettoyage local.

```
Function Divide(a, b)
  If b = 0 Then Throw New DivisionByZeroError
  Divide = a / b
End Function

Try
  Log Divide(10, 0)
Catch DivisionByZeroError As e
  ws.log "alerte : " & e.Message
  Throw                              ' relance pour traitement supérieur
End Try
```

> **Note — Forme simple en pratique.** Pour neutraliser une opération risquée en un coup, un `Try` minimal sans typage ni `Finally` suffit largement — c'est l'idiome qu'on retrouve dans le tutoriel `02-language-tour.vvl` : `Try : r = args(0) / args(1) : Catch : r = -1 : End Try`.

D.18. Bibliothèque standard

La bibliothèque standard regroupe les fonctions globales directement disponibles dans tout code VScript, sans import ni qualification. Le tableau ci-dessous les organise par domaine. Toutes les fonctions sont insensibles à la casse comme le reste du langage.

Domaine	Fonctions
Chaînes	`Len`, `StrComp`, `Replace`, `Split`, `LCase`, `UCase`, `InStr`, `InStrRev`, `Left`, `Right`, `Mid`, `Trim`, `LTrim`, `RTrim`, `Space`, `String`, `StrReverse`, `Escape`, `Unescape`
Mathématiques	`Abs`, `Sgn`, `Round`, `Exp`, `Int`, `Fix`, `Log`, `Sqr`, `Atn`, `Cos`, `Sin`, `Tan`, `Rnd`, `Randomize`
Dates	`Date`, `Time`, `Now`, `Timer`, `DateSerial`, `TimeSerial`, `DateValue`, `TimeValue`, `DatePart`, `Year`, `Month`, `Day`, `Weekday`, `Hour`, `Minute`, `Second`, `DateAdd`, `DateDiff`, `MonthName`, `WeekdayName`
Conversions	`CBool`, `CDate`, `CSng`, `CDbl`, `CByte`, `CInt`, `CLng`, `CStr`, `Hex`, `Oct`, `Chr`, `Asc`, `RGB`
Tests de type	`IsArray`, `IsDictionary`, `IsDate`, `IsEmpty`, `IsNothing`, `IsNull`, `IsNumeric`, `IsObject`
Formatage	`FormatDateTime`, `FormatNumber`, `FormatCurrency`, `FormatPercent`
Tableaux et dictionnaires	`Array`, `Dictionary`, `OrderedDictionary`, `SubArray`, `LBound`, `UBound`, `Filter`, `Join`
Introspection	`VarType`, `TypeName`, `ScriptEngine`, `ScriptEngineBuildVersion`, `ScriptEngineMajorVersion`, `ScriptEngineMinorVersion`
Sortie et utilitaires	`Log` (vers stdout), `IIf(cond, a, b)` (if inline — **eager** : les deux branches sont évaluées)

Figure C.7 — *Bibliothèque standard groupée par domaine — toutes les fonctions sont globales, accessibles sans import, et insensibles à la casse.*

Conventions d'arguments à connaître

Quelques conventions sont source de confusion récurrente. Le tableau ci-dessous récapitule les plus utiles à mémoriser.

Fonction	Signature et particularité
`Mid(string, start, length)`	**1-based** (premier caractère = `1`). `length` optionnel.
`InStr([start,] haystack, needle)`	**1-based** ; retourne `0` si non trouvé. Le `start` optionnel se met en premier.
`Left(string, n) / Right(string, n)`	Retourne les `n` premiers/derniers caractères. Tolère `n` plus grand que la longueur.
`Replace(string, find, repl)`	Remplace **toutes** les occurrences (forme simple, sans regex).
`Split(string, delimiter)`	Retourne un tableau **0-based**. Le `delimiter` peut être multi-caractères.
`Round(x[, digits])`	Arrondi banquier (vers le pair) — pas l'arrondi commercial.
`Int(x) / Fix(x)`	`Int` arrondit vers `-∞` ; `Fix` tronque vers `0`. Différents pour les négatifs : `Int(-3.5) = -4`, `Fix(-3.5) = -3`.
`IIf(cond, a, b)`	**Eager** — les deux branches sont toujours évaluées (contrairement à `If/Then`). Pour la paresse, utiliser `??` ou un `If` bloc.

Figure C.8 — *Conventions d'arguments fréquentes — l'indexation 1-based des fonctions de chaîne et l'évaluation eager de IIf méritent une attention particulière.*

D.19. Méthodes membres fluides

En complément des fonctions globales, VScript expose une **API fluente par méthodes membres** sur les trois subtypes composites — `String`, `Array`, `Dictionary`. La notation est celle d'un appel de méthode (`s.upper`, `arr.count`) et chaque méthode est chainable : `s.trim.upper` se lit naturellement de gauche à droite. Les fonctions globales restent disponibles ; les méthodes membres sont additives.

Méthodes sur String

Méthode	Description
`s.upper / s.lower`	Conversion de casse (équivalents à `UCase(s)` / `LCase(s)`).
`s.trim / s.ltrim / s.rtrim`	Suppression des espaces de bord (tout / gauche seul / droite seule).
`s.len`	Longueur de la chaîne.
`s.reverse`	Chaîne inversée.
`s.contains(x)`	Booléen : `s` contient-il `x` ?
`s.startswith(x) / s.endswith(x)`	Booléen : `s` commence/finit-il par `x` ?
`s.replace(a, b)`	Remplace toutes les occurrences de `a` par `b`.
`s.left(n) / s.right(n)`	Préfixe/suffixe des `n` premiers/derniers caractères.
`s.split(sep)`	Découpe en tableau (équivalent à `Split(s, sep)`).
`s.indexof(x)`	Position **1-based** de la première occurrence, `0` si absent.

Figure C.9 — *Méthodes membres sur String — les méthodes 0-arg (upper, trim, len, reverse) s'écrivent sans parenthèses ; les méthodes avec arguments les exigent.*

Méthodes sur Array et Dictionary

Type	Méthodes
Array	**`a.count`** — taille. **`a.contains(x)`** — booléen. **`a.indexof(x)`** — position 1-based, 0 si absent. **`a.join(sep)`** — concatène en chaîne. **`a.first`** — premier élément. **`a.last`** — dernier élément.
Dictionary	**`d.count`** — nombre d'entrées. **`d.has(k)`** — booléen sur l'existence d'une clé. **`d.keys`** — collection des clés. **`d.values`** — collection des valeurs. Lookup d'une clé : **`d("k")`** (l'accès parenthésé reste l'idiome principal).

Figure C.10 — *Méthodes membres sur Array et Dictionary — pour l'accès par clé d'un Dictionary, la forme parenthésée d("k") reste idiomatique ; les méthodes count/has/keys/values complètent l'introspection.*

> **Note — Règle des parenthèses.** Une méthode membre sans argument s'invoque **sans parenthèses** (`s.upper`, `a.count`) — elle est auto-appelée. Une méthode membre avec arguments s'invoque avec parenthèses (`s.replace(a, b)`, `a.contains(x)`). Cette convention est cohérente avec les propriétés (`Property Get`) qu'on lit aussi sans parenthèses. Les noms `get` et `set` étant réservés, il n'existe pas de membre portant ces noms.

D.20. Extension RegExp

L'extension `RegExp` est embarquée dans le runtime VScript. Elle expose une classe unique — `RegExp` — instanciée par `New`, dont on configure le motif et les drapeaux avant d'invoquer `Test`, `Execute` ou `Replace`.

Configuration de l'instance

Trois propriétés mutables gouvernent le comportement : **`Pattern`** (l'expression régulière, chaîne), **`Global`** (booléen — appliquer à toutes les occurrences), **`IgnoreCase`** (booléen — comparaison sans tenir compte de la casse).

```
Set re = New RegExp
re.Pattern = "[0-9]+"
re.Global = True
re.IgnoreCase = False

Log re.Test("abc123")              ' True (au moins une occurrence)
Log re.Replace("a1b22c333", "#")   ' a#b#c#  (Global = True)
```

Itération sur les occurrences

La méthode `Execute(s)` retourne une **collection de `Match`** dont on peut interroger la taille (`.Count`) et que l'on peut parcourir avec `For Each`. Chaque `Match` expose `.Value` (le texte de l'occurrence), `.FirstIndex` (position 0-based dans la chaîne) et `.Length`.

```
Set re = New RegExp
re.Pattern = "\w+@\w+"
re.Global = True

Set matches = re.Execute("contacts: alice@dev, bob@ops")
Log matches.Count                  ' 2

For Each m In matches
  Log m.Value & " @ index=" & m.FirstIndex
Next
' alice@dev @ index=10
' bob@ops @ index=21
```

D.21. Extension XML — API pratique XPath

L'extension XML embarquée s'appuie sur `ElementTree` (Python) sous le capot, exposée à VScript par une API navigable alignée sur la syntaxe XPath restreinte du module. Deux points d'entrée : **`XmlParse(txt)`** parse une chaîne XML et retourne le nœud racine ; **`XmlNew(tag)`** crée un nouveau document avec l'élément racine donné. Tous deux retournent un nœud navigable.

Surface du nœud — propriétés et navigation

Membre	Rôle
`.Tag`	Nom de l'élément (chaîne).
`.Text`	Contenu textuel direct (chaîne).
`.Count`	Nombre d'enfants directs.
`.Children`	Collection des nœuds enfants directs (`For Each`).
`.Attr(name[, default])`	Lecture d'un attribut, avec valeur par défaut optionnelle.
`.SetAttr(name, value)`	Définition ou modification d'un attribut.
`.HasAttr(name)` / `.RemoveAttr(name)`	Test et suppression d'attribut.
`.Find(xpath)`	Premier nœud correspondant au XPath relatif au nœud courant.
`.FindAll(xpath)`	Collection de tous les nœuds correspondants.
`.Child(tag)`	Raccourci : premier enfant direct avec ce tag.
`.Add(tag[, text])`	Crée un nouvel enfant à la fin, optionnellement avec un texte.
`.Append(node)`	Ajoute un nœud existant comme enfant.
`.Remove(node)`	Retire un nœud enfant.
`.ToString([pretty])`	Sérialise en chaîne XML, indentée si `pretty = True`.

Figure C.11 — *Surface XML — l'API combine navigation XPath (Find/FindAll), mutation (SetAttr/Add/Remove) et sérialisation (ToString) ; le module est suffisant pour transformer un document complet sans dépendance externe.*

Exemple de bout en bout

```
' Parser et lire
Set doc = XmlParse("<root><item id='1'>hello</item><item id='2'>world</item></root>")
Set first = doc.Find("item")
Log first.Tag & " = " & first.Text          ' item = hello

' Itérer sur tous les items
For Each it In doc.FindAll("item")
  Log it.Attr("id") & ": " & it.Text
Next                                          ' 1: hello / 2: world

' Construire et modifier
Set newdoc = XmlNew("books")
Set b = newdoc.Add("book", "Le Guide")
b.SetAttr "author", "Adams"
b.SetAttr "year", "1979"
Log newdoc.ToString(True)
' <books>
'   <book author="Adams" year="1979">Le Guide</book>
' </books>
```

> **Note — Le XPath est relatif.** `doc.Find("item")` cherche un élément `item` parmi les enfants directs de `doc`. Pour des expressions plus complexes (`book/title`, `item[@id='2']`), c'est la syntaxe XPath restreinte d'`ElementTree` qui s'applique — le `//descendant` absolu n'est en revanche pas supporté.

D.22. Bibliothèques réutilisables : [use $lib1 $lib2 ...]

Une référence `[vscript "..."]` peut jouer le rôle de **bibliothèque** : ses définitions de haut niveau — `Function`, `Sub`, `Class`, `Const` — deviennent visibles à un autre script qui la déclare avec `[use $lib]`. Ce mécanisme évite la duplication de code utilitaire entre plusieurs scripts d'un même workspace et permet de constituer de véritables modules réutilisables.

Syntaxe et exemple canonique

La déclaration `[use $lib1 $lib2 ...]` se place dans la clause d'options du `[vscript ...]` qui importe — soit en dernière position de la forme inline. Chaque `$lib` reste lui-même appelable normalement (`[$lib args]`) ; c'est sa double nature de référence appelable et de magasin de définitions qui rend l'idiome extensible.

```
# Une bibliothèque mathématique : une fonction + une constante + une classe
$mathlib = [vscript "function square(x)
  square = x * x
end function
const PI = 314
class Vec
  dim mx
  public sub Init(a)
    mx = a
  end sub
  public function Norm()
    Norm = mx * mx
  end function
end class"]

# Un script importateur qui utilise square et PI
$geo = [vscript "result = square(args(0)) + PI" [use $mathlib]]
[$geo 3]                              # => 323  (3*3 + 314)

# Un autre importateur qui utilise la classe Vec
$norm = [vscript "set v = new Vec
v.Init args(0)
result = v.Norm" [use $mathlib]]
[$norm 5]                             # => 25
```

Règles de fusion

Le mécanisme de `[use]` obéit à trois règles strictes qu'il est essentiel de connaître pour éviter les surprises.

Règle	Conséquence
Ré-exécution par appel	Au moment de chaque invocation **`[$script ...]`**, le code de chaque bibliothèque est **ré-exécuté** dans le namespace de l'importeur **avant** son propre code. Conséquence : pas d'**état partagé** entre appels (les variables globales de la lib sont réinitialisées à chaque fois) — chaque invocation est isolée.
Ordre d'injection	Les bibliothèques sont injectées dans l'**ordre de la liste `[use $a $b $c]`**. Si plusieurs libs définissent un identifiant du même nom (fonction, classe, constante), c'est la **dernière** qui l'emporte (**`$c`** écrase **`$a`** et **`$b`**).

Règle	Conséquence
Priorité de l'importeur	Le code de l'importeur s'exécute en **dernier**. Toute redéfinition locale par l'importeur **écrase** une définition de bibliothèque de même nom. C'est ce qui permet de spécialiser localement un comportement importé sans modifier la lib.

Figure C.12 — *Règles de fusion des bibliothèques — la ré-exécution par appel garantit l'isolation, l'ordre garantit la composition prévisible, la priorité de l'importeur garantit la spécialisation locale.*

Usage avec plusieurs bibliothèques

```
$strlib = [vscript "function shout(s)
  shout = UCase(s) & \"!\"
end function"]

$combo = [vscript "result = $\"{shout(args(0))} sq={square(args(1))}\"" [use $mathlib $strlib]]
[$combo "hi" 4]                          # => HI! sq=16
                                         # square vient de $mathlib, shout de $strlib
```

> **Note —** Une bibliothèque est, en pratique, aussi une référence appelable. Si l'on définit `$mathlib = [vscript "function square(x)… end function"]` (sans `result =`), l'invocation `[$mathlib …]` retourne `Empty` mais reste sans erreur. La double utilisation comme lib et comme fonction reste cohérente.

D.23. L'objet hôte ws — surface complète

VScript étant **sandboxé par construction** (chapitre D.1), tout accès à l'environnement du workspace passe par un objet hôte unique nommé **`ws`**, injecté par la façade VAILS au démarrage de l'exécution. L'Annexe B a posé les conditions de sécurité (toujours disponible en `[vscript]` inline ; sous double garde `env: true` + `tier=high` + `plugins_trusted` pour les plugins VScript) ; le présent chapitre détaille la surface effectivement exposée.

L'objet `ws` est une façade jailée sur les builtins existants de VVL — il hérite donc des mêmes verrous (DB, fichiers), de la même résolution de chemin (`ctx.resolve_path`), et des mêmes contrôles d'accès. Sa surface est composée de sept membres principaux.

ws.name — identifier le workspace courant

```
Log "workspace=[" & ws.name & "]"
' workspace=[mon_projet]
```

Propriété en lecture seule retournant le nom du workspace dans lequel le code s'exécute. Utile pour journaliser le contexte, ou pour conditionner un comportement selon l'environnement (développement, recette, production).

ws.files — fichiers, jailés à {ws}/files

L'accès au système de fichiers est strictement confiné au sous-répertoire `{ws}/files`. Aucune sortie par chemin absolu ni par `..` n'est possible : la résolution est faite par `ctx.resolve_path`, qui rejette tout chemin sortant du jail.

Méthode	Effet
`ws.files.write(name, text)`	Écrit (ou crée) le fichier nommé avec le texte fourni. Atomique : écriture en **`temp + rename`**.
`ws.files.read(name)`	Retourne le contenu textuel du fichier. Lève si absent.
`ws.files.exists(name)`	Retourne **`True`** ou **`False`**.

Méthode	Effet
`ws.files.delete(name)`	Supprime le fichier. Idempotent — pas d'erreur si absent.

```
ws.files.write "out.txt", "hello from vscript"
Log ws.files.read("out.txt")                        ' hello from vscript
Log ws.files.exists("out.txt")                      ' True
ws.files.delete "out.txt"
Log ws.files.exists("out.txt")                      ' False
```

ws.db(name) — SQLite jailée

L'accès à une base SQLite par workspace s'obtient via `ws.db("nom")`, qui retourne un objet connexion vers `{ws}/db/<name>.sqlite`. Cette base est créée à la demande si elle n'existait pas. Trois méthodes principales : **`exec`** pour les DDL/DML sans résultat, **`query`** pour les `SELECT` (avec paramètres liés), et **`tables`** pour lister les tables.

Méthode	Description
`db.exec(sql)`	Exécute un DDL/DML (CREATE, INSERT, UPDATE, DELETE). Les valeurs littérales sont à intégrer directement dans la requête (pas de paramètres liés sur `exec`).
`db.query(sql, …params)`	Exécute un SELECT avec **paramètres liés `?`**. Retourne une collection itérable de lignes, chacune sous forme de `Dictionary` indexé par nom de colonne.
`db.tables`	Collection des noms de tables présentes dans la base.

Figure C.13 — *Sous-objet ws.db — la séparation exec/query reflète la sécurité : exec accepte n'importe quel DDL/DML littéral, query expose les paramètres liés pour neutraliser les injections.*

```
Set db = ws.db("app")
db.exec "create table if not exists t(id integer, name text)"
db.exec "insert into t values (1, 'alice')"
db.exec "insert into t values (2, 'bob')"

Set rs = db.query("select name from t where id >= ? order by id", 1)
Log rs.Count                                        ' 2
For Each row In rs
  Log row("id") & " => " & row("name")
Next
' 1 => alice
' 2 => bob

Set names = db.tables                               ' collection des noms de table
For Each t In names : Log t : Next
```

ws.csv(name) — tables CSV de premier niveau

L'accès aux tables CSV du workspace passe par `ws.csv("nom")`, qui retourne une instance avec une API riche de neuf méthodes. Le fichier sous-jacent est `{ws}/<nom>.csv` ; les modifications restent en mémoire jusqu'à l'appel explicite de `.save`.

Méthode	Description
`t.all`	Collection de toutes les lignes (en dictionnaires).
`t.find(key)`	Première ligne dont la clé primaire vaut `key`. **Exige `t.key "col"`** au préalable pour déclarer la clé primaire.

Méthode	Description
`t.where(col, op, val)`	Sous-collection filtrée. Opérateurs : `=`, `!=`, `>`, `<`, `>=`, `<=`, `contains`, `startswith`, `endswith`.
`t.add(record)`	Ajoute une ligne (dictionnaire `{"col": val, …}` accepté grâce aux littéraux Tier 1).
`t.update(key, fields)`	Met à jour la ligne identifiée par sa clé primaire.
`t.delete(key)`	Supprime la ligne identifiée.
`t.save`	Persiste les modifications sur disque.
`t.clear`	Vide la table (en mémoire ; `.save` pour persister).
`t.count`	Nombre de lignes.

Figure C.14 — *Sous-objet ws.csv — l'API combine lecture, filtrage, mutation et persistance. Les opérations qui nécessitent une clé primaire exigent une déclaration préalable t.key "colonne".*

```
Set t = ws.csv("clients")
t.key "id"                                      ' déclare la clé primaire

Log t.count                                    ' 42
Set rows = t.all                                ' toutes les lignes
Set r = t.find("42")                            ' ligne avec id="42"
Set hits = t.where("age", ">", 30)              ' filtrage

t.add {"id": "43", "name": "Zoé", "age": 28}    ' littéral Tier 1
t.update "42", {"name": "Alice mise à jour"}
t.delete "42"
t.save                                          ' persiste sur disque
```

ws.mem(name) — magasin clé-valeur léger

Pour les besoins de cache ou de registre simple qui ne justifient pas une table SQLite, `ws.mem("nom")` retourne un magasin clé-valeur persisté dans `{ws}/db/`. Deux méthodes seulement : **`put`** pour écrire, **`value`** pour lire. Les noms `get` et `set` sont réservés par le langage et ne peuvent donc pas servir d'identifiants de méthode — c'est la raison du choix `value`/`put`.

```
ws.mem("cache").put "last_run", Now
Set t = ws.mem("cache").value("last_run")
Log FormatDateTime(t, vbGeneralDate)            ' 2026-05-22 04:42:17
```

ws.log msg — journalisation

Le sous-objet `ws.log` écrit une entrée dans le journal du workspace, avec horodatage automatique. Il s'utilise comme une `Sub` à un argument.

```
ws.log "import terminé : " & CStr(n) & " lignes"
```

ws.vvl(code) — rappel VVL depuis VScript

Le point d'orgue de l'objet `ws` est le **rappel VVL** : `ws.vvl("…code…")` évalue une chaîne VVL depuis l'intérieur du code VScript, dans le contexte serveur courant. Cela permet à un script VScript d'invoquer n'importe quel builtin VVL — y compris d'autres plugins, des keywords métier, ou l'évaluation de fragments dynamiques.

```
Dim r : r = ws.vvl("[+ 1 2]")                    ' r = 3 (entier)

ws.log "calcul VVL depuis VScript"
Dim taux : taux = ws.vvl("[tva-applicable $client]")
                                                  ' invoque un keyword métier VVL
```

> **Limite —** Comme `[evaluate]` côté VVL, `ws.vvl` ouvre la possibilité d'évaluer du code dynamique — avec les précautions d'usage : ne jamais y interpoler directement une entrée utilisateur non assainie, sous peine d'injection de code VVL équivalent à une `eval()` ouverte.

D.24. Interopérabilité OO — proxys vivants VVL ⇄ VScript

L'un des points les plus raffinés de l'intégration VScript dans VAILS est sa gestion de l'**interopérabilité orientée objet**. Une instance d'une classe VVL transmise à un code VScript — et réciproquement — n'y est ni copiée ni stringifiée : elle est exposée sous forme de **proxy vivant**, c'est-à-dire d'un objet de surface qui délègue lecture, écriture et appel à l'objet sous-jacent. L'identité est préservée, les mutations sont reflétées des deux côtés.

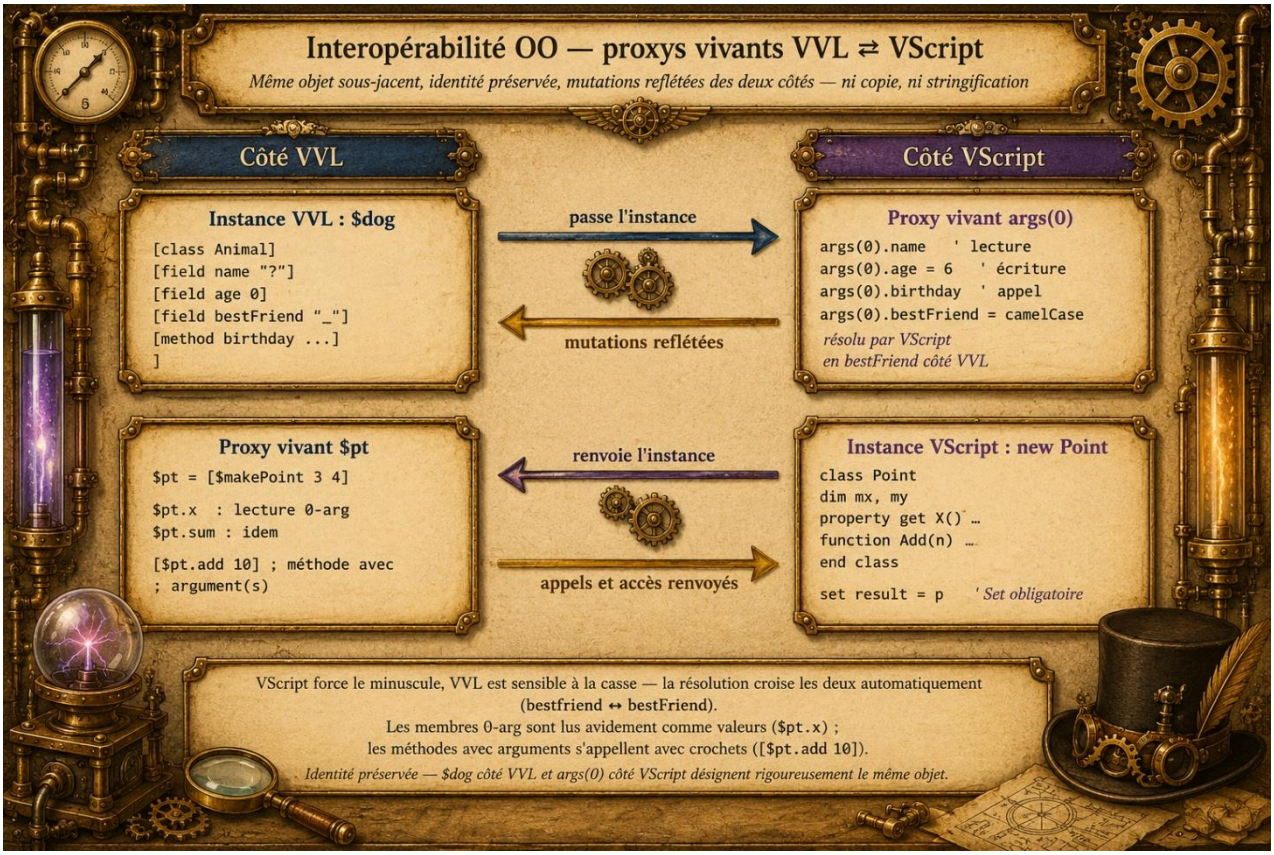


Figure C.15 — *Interopérabilité OO — passage d'une instance VVL à VScript (haut) et inversement (bas) ; dans les deux cas l'identité est préservée et les mutations traversent la frontière.*

Sens 1 — instance VVL exposée à VScript

Une instance VVL passée comme argument à une référence `[vscript …]` devient un objet VScript ordinaire dans `args`. On y lit les champs avec la notation membre `.`, on y appelle les méthodes de la même manière (l'appel ré-entre l'évaluateur VVL en transparence), et on peut même y écrire — la modification est immédiatement reflétée dans l'instance VVL d'origine.

```
# Côté VVL — définition et instanciation
[class Animal
    [field name "?"]
    [field age 0]
    [field bestFriend "Buddy"]
    [method birthday [] $self.age = ($self.age + 1)]
]
$dog = [new Animal "Rex" 5]

# Côté VScript — lecture, appel, écriture
$readname = [vscript "result = args(0).name"]
[$readname $dog]                                    # => Rex

$grow = [vscript "args(0).age = 6
result = args(0).age"]
[$grow $dog]                                        # => 6
$dog.age                                            # => 6 (côté VVL, même objet !)

$bday = [vscript "args(0).birthday
result = args(0).age"]
[$bday $dog]                                        # => 7 (méthode VVL invoquée depuis
VScript)
$dog.age                                            # => 7
```

Le pont de casse — bestFriend ↔ bestfriend

VScript étant **insensible à la casse** (D.4), tout identifiant qu'il manipule est ramené en minuscules avant comparaison. VVL, en revanche, est sensible à la casse — un champ déclaré `bestFriend` n'est pas le même que `bestfriend`. Pour que la frontière ne devienne pas un cauchemar, le proxy de VAILS **résout entre les deux** : depuis VScript, on accède à `args(0).bestfriend`, et la résolution descend automatiquement vers le champ `bestFriend` du côté VVL.

```
# camelCase VVL accessible en lowercase VScript
$readbf = [vscript "result = args(0).bestfriend"]
[$readbf $dog]                                      # => Buddy
```

Sens 2 — instance VScript exposée à VVL

Réciproquement, un `New Foo` créé côté VScript et renvoyé via `Set result = obj` devient un objet VVL utilisable. L'accès aux membres suit les conventions évoquées au chapitre D.19 : les membres **0-argument** (propriétés `Property Get`, fonctions sans paramètre) sont lus **avidement comme valeurs** avec la notation pointée ordinaire `$obj.x` ; les méthodes **avec arguments** s'invoquent avec la syntaxe d'appel VVL `[$obj.method args …]`.

```
$makePoint = [vscript "class Point
  dim mx
  dim my
  public sub Init(a, b)
    mx = a
    my = b
  end sub
  public function Sum()
    Sum = mx + my
  end function
  public function Add(n)
    Add = mx + n
  end function
  public property get X()
    X = mx
  end property
end class
set p = new Point
p.Init args(0), args(1)
set result = p"]

$pt = [$makePoint 3 4]
$pt.x                                   # => 3   (propriété, lue comme valeur)
$pt.sum                                 # => 7   (fonction 0-arg, lue comme
valeur)
[$pt.add 10]                            # => 13  (méthode avec argument)
```

Récapitulatif des règles cruciales

Règle	Détail
Proxys, pas de copies	Les objets traversent la frontière par référence. Mutation côté A immédiatement visible côté B. Identité (`Is`) préservée.
Set obligatoire pour renvoyer un objet	Du côté VScript, `Set result = obj` est requis pour que l'instance soit correctement transmise à VVL. Sans `Set`, l'erreur est levée à l'évaluation.
Casse résolue	VScript minuscule, VVL sensible — le proxy croise les deux. `args(0).bestfriend` atteint `bestFriend` côté VVL.
0-arg vs avec arguments	Membres sans paramètre : lus avidement comme valeurs (`$obj.x`). Méthodes avec arguments : appelées avec crochets (`[$obj.add 10]`).
Aucune sérialisation	Aucune conversion en chaîne ou en JSON n'intervient. Le coût d'appel reste celui d'une indirection de fonction.

Figure C.16 — *Règles cruciales de l'interopérabilité OO — la sémantique est celle de la référence partagée, à mémoriser pour éviter les pièges classiques de la copie implicite.*

D.25. Modernisations VScript — récapitulatif vs VBScript

VScript se distingue de VBScript par une série d'ajouts purement additifs — la plupart introduits dans VAILS 5.18 (Tier 1) et VAILS 5.19 (Tier 3) — qui modernisent l'expression sans casser la compatibilité du code historique. Le tableau ci-dessous récapitule l'écart par rapport à VBScript classique. Aucune de ces modernisations ne casse de code VBScript existant : elles enrichissent strictement la grammaire.

Modernisation	Détail	Chapitre
Littéraux tableau `[…]` et dict `{…}`	Construction directe de données et de records sans `Array()` ni `Dictionary()`.	D.5

Modernisation	Détail	Chapitre
Chaîne interpolée `$"… {expr} …"`	Substitution d'expressions ; échappements C-like (`\n`, `\t`).	D.6
Formatage par pipe `{ x | f | g(a) }`	Chaîne de transformations sur la valeur interpolée.	D.6
Méthodes membres fluides	`s.upper.trim`, `arr.count`, `d.has(k)` — additives aux fonctions globales.	D.19
`Return [expr]`	Sucre atomique = affectation du nom + `Exit Function/Sub`.	D.13
Optional chaining `obj?.member`	Accès null-safe, retourne `Empty` au lieu de lever.	D.11
Coalescing `??`	`a ?? b` — `a` sauf si `Null`/`Empty` → `b` (droite paresseuse).	D.11
Assignations composées `+= -= … &=`	Sucre arithmétique avec parenthésage automatique.	D.13
`IIf(cond, a, b)`	If inline (eager — les deux branches sont évaluées).	D.18
`For Each k, v In dict`	Déstructuration clé/valeur sur dictionnaire.	D.12
`Enum … End Enum`	Constantes nommées qualifiées (`Color.Red`).	D.15
Fonctions de 1re classe `AddressOf`	Référence appelable stockable et passable.	D.16
`[use $lib1 $lib2 …]`	Bibliothèques réutilisables côté VAILS.	D.22
Court-circuit `And`/`Or` (déviation)	Activé sur gauche numérique déterminante (0 / −1) ; logique trivaluée préservée.	D.10
Littéraux `&H` et `&B` numériques	Correction d'une régression : `&H1F` = 31, pas `Chr(31)`.	D.5

Figure C.17 — *Modernisations VScript par rapport à VBScript — toutes additives, toutes optionnelles : le code VBScript de référence continue de fonctionner sans modification.*

D.26. Pièges classiques et déviations à connaître

L'expérience accumulée sur les huit tutoriels `vvl/vscript/01..08` permet d'inventorier les pièges récurrents qui surprennent un développeur découvrant le langage. Ce chapitre les regroupe par catégorie.

Échappement de la chaîne hôte VVL

Le code VScript voyage à l'intérieur d'une chaîne VVL **doublement quotée**. Tout guillemet double interne se redouble côté VScript (`""` → un guillemet) et doit en plus être échappé pour la chaîne VVL hôte avec `\"`. Le caractère `\` (utilisé en VScript pour la division entière) doit lui aussi être échappé en `\\`. Cette double couche d'échappement est la source la plus fréquente d'erreurs de débutant.

```
# VScript natif :  out = "hello" : div = 100 \ n
# Dans une chaîne VVL :
$x = [vscript "out = \"hello\"
div = 100 \\ n"]

# Astuce : passer par une variable VVL pour les chaînes complexes
$code = "out = "hello world""             # difficile : encore plus d'échappement
# Le plus simple : limiter les guillemets internes (préférer la concaténation
# de morceaux courts), ou utiliser les chaînes interpolées $"..." qui acceptent \n
```

Pièges spécifiques au langage

Piège	Symptôme et résolution
Casse insensible	VScript ramène tous les identifiants en minuscules. Conséquence : un champ VVL camelCase (`bestFriend`) doit s'écrire `bestfriend` côté VScript (le proxy résout, voir D.24).
`Set` obligatoire pour les objets	Affecter une instance sans `Set` lève à l'exécution. La règle est constante : valeur = `=`, objet = `Set`. S'applique aussi à `Set result = obj` pour renvoyer un objet à VVL.
`On Error Resume Next` exclu	L'idiome VBScript classique n'est pas supporté. Utiliser `Try/Catch/End Try` — y compris dans sa forme minimaliste `Try : … : Catch : … : End Try`.
`get` et `set` sont réservés	Conséquence : impossible de nommer un membre `get` ou `set`. C'est la raison pour laquelle `ws.mem` expose `value`/`put` plutôt que les noms attendus.
Méthode 0-argument sans parenthèses	Une méthode `Sub Bump()` s'invoque `obj.Bump`, sans parenthèses (idem pour les `Property Get`). Avec parenthèses, on tenterait d'appliquer le résultat — souvent une erreur.
`IIf` est eager	`IIf(cond, expr_a, expr_b)` évalue **toujours** les deux branches. Pour la paresse, utiliser `If/Then/Else` (bloc) ou `??` quand applicable.
`Int` vs `Fix` sur les négatifs	`Int(-3.5)` = −4 (arrondi vers −∞), `Fix(-3.5)` = −3 (troncature vers `0`). Les deux fonctions coïncident sur les positifs ; le piège est en zone négative.
Indexation mixte 0/1	Tableaux et `args` sont **0-based** ; `Mid`/`InStr`/`Left`/`Right` sont **1-based**. Pas négociable, hérité de VBScript.
Champs de classe par `Dim`	Les formes VBScript `Private x` / `Public x` au niveau variable sont **rejetées**. Utiliser `Dim` systématiquement dans une classe.

Figure C.18 — *Pièges classiques — la majorité se manifeste à l'exécution par des erreurs explicites ; quelques-uns (casse, eager IIf) sont silencieux et méritent une attention particulière.*

D.27. Tables de référence rapide

Pour clore l'annexe, trois tables condensées rassemblent les éléments les plus fréquemment consultés du langage. Elles servent de pense-bête lors de l'écriture d'un script — leur place est dans l'imaginaire collectif d'un livre de référence.

Mots-clés réservés

Catégorie	Mots-clés
Déclaration	`Dim`, `ReDim`, `Preserve`, `Const`, `Set`, `Erase`, `Public`, `Private`, `Default`
Contrôle de flux	`If`, `Then`, `Else`, `ElseIf`, `End`, `Select`, `Case`, `For`, `Each`, `To`, `Step`, `Next`, `In`, `Do`, `Loop`, `While`, `Until`, `Wend`, `Exit`, `Return`, `Call`, `With`
Procédures	`Sub`, `Function`, `Property`, `Get`, `Let`, `ByVal`, `ByRef`
OO	`Class`, `Inherits`, `New`, `Me`, `MyBase`, `MyClass`
Erreurs	`Try`, `Catch`, `Finally`, `Throw`
Énumérations	`Enum`
Opérateurs / valeurs	`And`, `Or`, `Not`, `Xor`, `Mod`, `Is`, `True`, `False`, `Empty`, `Null`, `Nothing`, `NaN`, `Infinity`
Spéciaux	`AddressOf`, `Rem`, `Log`, `IIf`

Figure C.19 — *Mots-clés du langage — l'ensemble est figé et insensible à la casse comme les identifiants. Un identifiant utilisateur ne peut pas réutiliser un de ces mots, qu'il soit en majuscules ou en minuscules.*

Opérateurs par précédence

Niveau	Opérateurs	Sémantique
1 — fort	`^`	Puissance.
2	`+x -x`	Unaires (signe).
3	`* /`	Multiplication, division flottante.
4	`\`	Division entière.
5	`Mod`	Modulo.
6	`+ -`	Addition, soustraction.
7	`&`	Concaténation de chaînes.
8	`= <> < > <= >= Is`	Comparaisons et identité d'objet.
9	`Not`	Négation logique.
10	`And`	Conjonction (court-circuit numérique).
11	`Or`	Disjonction (court-circuit numérique).
12	`Xor`	Ou exclusif.
13 — faible	`??`	Coalescing.

Figure C.20 — *Opérateurs par ordre de précédence décroissant — à mémoriser : / produit toujours un Double, \ produit toujours un entier, ?? est le plus faible.*

Conventions d'arguments à retenir

Fonction / collection	Convention
`args(i)`, tableaux `a(i)`, dictionnaires `d("k")`	**Indexation 0-based** pour les collections.
`Mid`, `Left`, `Right`, `InStr`, `InStrRev`	**Indexation 1-based** pour les fonctions de chaîne (héritage VBScript).
`IIf(cond, a, b)`	**Eager** — les deux branches sont évaluées. Pour la paresse, utiliser `If/Then` ou `??`.
`Replace(s, find, repl)`	Remplace **toutes** les occurrences (forme simple sans drapeau).
`Int(x)` vs `Fix(x)`	`Int` arrondit vers `-∞`, `Fix` tronque vers `0`. Identiques sur les positifs.
`Round(x[, n])`	Arrondi banquier (vers le pair) — pas l'arrondi commercial.
Méthode 0-arg vs avec arguments	Sans arguments → notation sans parenthèses (`s.upper`). Avec arguments → parenthèses (`s.replace(a, b)`).
`Set` vs `=`	Valeur → `=`. Objet → `Set`. **Toujours**.

Figure C.21 — *Conventions d'arguments — le mélange 0-based / 1-based et la règle Set/= sont les deux conventions à intérioriser pour écrire du VScript naturellement.*