HAL
open science

Génération Modulaire de Grammaires Formelles

Simon Petitjean

To cite this version:

Simon Petitjean. Génération Modulaire de Grammaires Formelles. Informatique et langage [cs.CL]. Universite d'Orléans, 2014. Français. NNT : . tel-01163150v1

HAL Id: tel-01163150

https://theses.hal.science/tel-01163150v1

Submitted on 12 Jun 2015 (v1), last revised 21 Sep 2015 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MIPTIS

MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE

présentée par :

Simon PETITJEAN

soutenue le 11 décembre 2014

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/Spécialité : **Informatique**

Génération Modulaire de Grammaires Formelles

THÈSE DIRIGÉE PAR :

Denys DUCHIER	Professeur, Université d'Orléans

ET CO-ENCADRÉE PAR :

Yannick PARMENTIER	Maître de Conférences, Université d'Orléans

RAPPORTEURS :

Laura KALLMEYER	Professeure, Université de Düsseldorf
Kim MENS	Professeur, Université catholique de Louvain

EXAMINATEURS :

Claire GARDENT	Directrice de Recherche CNRS, LORIA, *présidente du jury*
Olivier BONAMI	Maître de Conférences, Université Paris-Sorbonne
Eric DE LA CLERGERIE	Chargé de Recherche, INRIA Rocquencourt
Ann COPESTAKE	Professeure, Université de Cambridge

Remerciements

J'aimerais tout d'abord remercier ceux sans qui cette thèse n'aurait pas existé, à savoir la région Centre, qui m'a accordé l'indispensable bourse de thèse, et mes encadrants, qui ont eu confiance en moi pour cette mission. Pour réaliser à quel point il est enviable de travailler avec Denys et Yannick, il m'a suffit de citer leurs noms lors de chacun de mes déplacements, et de voir les yeux de mes interlocuteurs s'illuminer. Je leur suis en particulier très reconnaissant pour leur grande patience pendant les derniers mois de rédaction, et leur disponibilité dans tous les moments où je les ai sollicités.

Cela a été un immense honneur d'exposer mon travail devant un jury tel que celui ci. Merci aux membres de ce jury d'avoir accepté cette tâche, et d'avoir fait de la soutenance un moment si agréable, notamment grâce à des questions des plus pertinentes.

Je remercie mes collaborateurs, avec qui travailler a été un très grand plaisir : Brunelle, Emmanuel, Nicola, Timm, Alexander, Bojana. Mes discussions avec eux ont réellement été les récréations de ma thèse. Lorsqu'on s'attaque à une épreuve de cette taille, il n'est pas rare d'avoir la sensation d'être seul et perdu. Les personnes que j'ai rencontrées au cours d'écoles d'été ou de conférences ont pu me montrer que je n'étais pas seul, et qu'être perdu faisait partie du jeu.

Ces années passées au LIFO sont passées beaucoup trop vite, essentiellement parce que ce laboratoire est peuplé de personnes formidables. Merci à tous ceux qui m'ont si bien accueilli, et qui m'ont appris tant de choses. Tout spécialement, je remercie Julien, Hélène, Nico, Sylvain, Anthony, Bastien, Mathieu, Matthieu, Maxime, Davide, Florent, Guillaume, Ahmed, Yohan, Ioan, Ali. Je pense que la seule façon de rendre cette liste moins incomplète est d'adresser mes remerciements à tous ceux qui ont participé à quelques uns de ces (parfois étonnants) moments de détente dans la salle du même nom. Merci aussi à Florence, Isabelle et Brigitte d'avoir rendu toutes les tâches administratives si faciles.

Merci à ceux sans qui vivre à Orléans aurait souvent été bien triste : Rabot, Benoît, David, François, Virginie, d'autres qui ont déjà été cités, et beaucoup d'autres qu'il serait trop difficile de lister sans que ce chapitre ne soit le plus long de ce manuscrit. J'ai notamment passé de très bons moments grâce à deux associations formidables, l'ADSO et l'ADDOSHS. Des journées à oublier se sont transformées en excellents souvenirs grâce à tous ces amis qui ne m'ont jamais abandonné. J'aimerais également remercier Leila, sans qui je n'aurais probablement jamais eu l'idée ni le courage de me lancer dans une aventure aussi ambitieuse.

Enfin, il est évident que je remercie tous les membres de ma famille, pour leur soutien permanent, et leur foi inébranlable. Rien ne laissait penser que mon destin était d'être un jour docteur en informatique. Le chemin a été long, mais mes parents ont tout donné pour le rendre moins difficile, et surtout pour me laisser tracer celui que je voulais. Cette thèse leur est naturellement dédiée.

Simon

Sommaire

Introduction **ix**

I État de l'Art 1

Chapitre 1 : Des langages dédiés au Traitement Automatique des Langues **3**
1.1 Objet d'étude : la langue naturelle 3
1.2 Des outils formels pour décrire la langue 4
1.2.1 Plusieurs niveaux d'analyse 4
1.2.2 Des interfaces entre les niveaux 8
1.3 Un besoin de langages dédiés 13

Chapitre 2 : Décrire la Morphologie **15**
2.1 Introduction 15
2.2 Analyse morphologique par unification 16
2.2.1 PFM 16
2.2.2 Network Morphology 17
2.3 Analyse morphologique par automates à états finis 18
2.3.1 Morphologie à Deux Niveaux 19
2.3.2 PARSLI 20
2.4 Conclusion 21

Chapitre 3 : Décrire la Syntaxe **23**
3.1 Introduction 23
3.2 Syntaxe générative par réécriture de chaînes 24
3.2.1 Grammaires hors contexte 24
3.2.2 Grammaires Lexicales Fonctionnelles 29
3.2.3 Grammaire Syntagmatique Guidée par les Têtes 31
3.3 Syntaxe générative par réécriture d'arbres : la Grammaire d'Arbres Adjoints . . 35
3.3.1 La Grammaire d'Arbres Adjoints 35
3.3.2 Une approche basée sur les metarègles 38
3.3.3 Des approches métagrammaticales 39
3.3.4 eXtensible MetaGrammar 41
3.4 Syntaxe basée sur la théorie des modèles 50
3.4.1 eXtensible Dependency Grammar 50
3.4.2 Grammaire de Propriétés 54

3.5 Conclusion 55

Chapitre 4 : Construction de compilateurs **57**
4.1 Introduction 57
4.2 Compilateurs de Compilateurs 58
4.2.1 CDL3 58
4.3 Génération de Composants Spécifiques de Compilateurs 60
4.3.1 Génération d'analyseurs syntaxiques avec Yacc 60
4.3.2 Génération d'analyseurs syntaxiques avec Parsec 61
4.4 Construction modulaire de DSL 62
4.4.1 Neverlang 62
4.5 Conclusion 64

II Un Compilateur Modulaire **65**

Introduction **67**

Chapitre 5 : Différents niveaux de modularité **69**
5.1 Introduction 69
5.2 Un besoin de modularité 69
5.3 Différents types d'utilisateurs 70
5.4 Une architecture modulaire 71
5.5 Modularité linguistique 72
5.6 Modularité d'assemblage 74
5.7 Modularité du compilateur et de l'exécuteur 75
5.7.1 Compilateur 75
5.7.2 Exécuteur 75
5.8 Conclusion 76

Chapitre 6 : Assemblage modulaire d'un langage dédié **77**
6.1 Introduction 77
6.2 Définir un langage dédié 77
6.3 Enrichir un langage 78
6.4 Des briques de langage 80
6.4.1 Définir des briques 80
6.4.2 Connecter des briques 81
6.4.3 Assemblage d'un DSL 84
6.5 Conclusion 85

Chapitre 7 : Assemblage modulaire d'un compilateur **87**
7.1 Introduction 88
7.1.1 Étapes de la compilation 88
7.1.2 Répartir les règles dans des briques 89
7.1.3 EDCG 90
7.1.4 Variables attribuées 91

7.2 Étape d'analyse . . . 93
7.2.1 Analyse syntaxique . . . 93
7.2.2 Analyse lexicale . . . 95
7.2.3 Connecter des briques . . . 95
7.3 Étape de Typage . . . 97
7.3.1 Principe . . . 97
7.3.2 Typage des données . . . 98
7.3.3 Typage du langage de contrôle . . . 100
7.3.4 Typage des classes . . . 101
7.3.5 Typage des dimensions . . . 102
7.3.6 Algorithme de typage . . . 104
7.3.7 Implantation . . . 106
7.4 Étape de dépliage . . . 107
7.4.1 Principe . . . 107
7.4.2 Formalisation du dépliage . . . 108
7.4.3 Exemples . . . 111
7.4.4 Implantation . . . 113
7.4.5 Extensibilité . . . 113
7.5 Étape de génération . . . 114
7.5.1 Principe . . . 114
7.5.2 Formalisation et implantation . . . 115
7.5.3 Extensibilité . . . 118
7.5.4 Code spécifique aux classes . . . 119
7.6 Constructeurs de briques . . . 121
7.7 Point d'entrée de l'exécution . . . 122
7.7.1 Lancer l'évaluation du code généré . . . 122
7.7.2 Définir les accumulateurs . . . 123
7.7.3 Code généré . . . 124
7.8 Exemple : XMG-1 . . . 125
7.9 Conclusion . . . 127

Chapitre 8 : Assemblage modulaire d'un générateur 131
8.1 Introduction . . . 131
8.1.1 Architecture du générateur . . . 131
8.1.2 Exemple . . . 132
8.2 Étape d'exécution . . . 133
8.3 Étape de résolution . . . 134
8.3.1 Solveurs et principes . . . 134
8.3.2 Solveurs . . . 135
8.3.3 Principes et plugins . . . 135
8.3.4 Brique de solveur . . . 135
8.3.5 Associer les solveurs aux dimensions . . . 136
8.4 Étape de sérialisation . . . 137
8.4.1 Fonctionnement . . . 137
8.4.2 Extensibilité . . . 138
8.5 Conclusion . . . 138

Chapitre 9 : Comparaison et conclusion **141**
9.1 Introduction 141
9.2 Comparaison 141
9.2.1 Similitudes architecturales 141
9.2.2 Comparaison contrastive 143
9.3 Conclusion 145

III Application de l'assemblage modulaire de DSL à la description de ressources linguistiques **147**

Introduction **149**

Chapitre 10 : Application à la syntaxe **151**
10.1 Grammaires d'Arbres Adjoints 151
10.1.1 Problématique linguistique 151
10.1.2 Création de briques 152
10.1.3 Assemblage du compilateur 154
10.2 Grammaires d'Interaction 154
10.2.1 Problématique linguistique 154
10.2.2 Création de briques 156
10.2.3 Assemblage du compilateur 157
10.3 Conclusion 158

Chapitre 11 : Application à la morphologie **161**
11.1 Problématique linguistique : morphologie verbale en Ikota 161
11.1.1 Blocs élémentaires 163
11.1.2 Exemple de dérivation 164
11.1.3 Phonologie de surface 165
11.2 Création des briques 165
11.2.1 Une brique pour décrire des champs ordonnés 165
11.2.2 Une brique pour la description morphologique 166
11.3 Assemblage du compilateur 168
11.4 Conclusion 169

Chapitre 12 : Application à la sémantique **171**
12.1 Problématique linguistique 171
12.1.1 Factorisation de squelettes d'arbres et de frames 172
12.1.2 une nouvelle dimension <frame> 173
12.2 Création de briques 173
12.2.1 Une brique pour définir des hiérarchies de types 173
12.2.2 Une brique pour décrire des frames 174
12.3 Assemblage du compilateur 175
12.4 Conclusion 176

Conclusion générale **179**

Bilan 179
Perspectives 181

Annexes **183**

Annexe A. Outils d'extension **185**
Introduction 185
Commandes 185
Compilateurs 185
Création d'une brique 186
Contributions 186

Annexe B. Application à la musique **187**
Introduction 187
Le compilateur 188
Une dimension <music> 188
Principes 189
Des interactions entre dimensions 189
Sortie 190
Une métagrammaire pour les accords musicaux 190
Intuition 190
Décrire les notes 190
Décrire des Intervalles 192
Décrire des Accords 193
Progressions d'accords 195

Bibliographie **197**

Table des figures

1.1 Une grammaire hors contexte pour un fragment du français 6
1.2 Un arbre de dérivation pour la phrase "Un chat mange une souris" 6
1.3 Un graphe de dépendance pour la phrase "Un chat mange une souris" 7
1.4 Une représentation sémantique sous la forme d'un graphe 8
1.5 Représentation d'un groupe nominal 10
1.6 Représentation du groupe nominal "une souris" 10

2.1 Un réseau d'héritage DATR 17
2.2 Une grammaire HFST 19
2.3 Flexion du verbe *Balayer* décrite avec PARSLI 20

3.1 Exemple de règles PATR II 28
3.2 Exemple de lexique PATR II 28
3.3 Grammaire LFG, c-structure et f-structure pour la phrase "John loves Mary" 30
3.4 Une hiérarchie d'héritage pour les noms 31
3.5 La substitution en TAG 35
3.6 L'adjonction en TAG 36
3.8 Verbe avec subject canonique et objet canonique ou extrait 36
3.7 Une partie de la famille des verbes transitifs en TAG 37
3.9 Exemple de classe MGCOMP (issu de [Thomasset et Clergerie, 2005]) 40
3.10 Une construction active avec sujet canonique et objet relatif 41
3.11 Fragments d'arbres TAG pour un sujet canonique, un objet relatif et la morphologie verbale active 42
3.12 Une construction active avec sujet canonique et objet canonique 42
3.13 Combinaisons sans contraintes 46
3.14 Règles d'unification pour les couleurs. 46
3.15 Arbres élémentaires colorés 47
3.16 Combinaisons filtrées par couleurs 47
3.17 Fragments décorés avec le principe *rank* 48
3.18 Unique modèle après filtrage par le principe *rank* 48
3.19 Exemple d'analyse par dépendance (extrait de [Debusmann, 2006]) 51
3.20 Exemple d'entrée lexicale (extrait de [Debusmann, 2006]) 51
3.21 Entrée lexicale pour "wants" (extraite de [Debusmann, 2006]) 52
3.22 Entrée lexicale pour le mot *wants* dans le formalisme XDG 52
3.23 Fragment d'une grammaire de propriétés pour le français (constructions verbales de base) 55

4.1 La définition d'une *slice* Neverlang définissant la syntaxe et la sémantique d'une somme 63

4.2 La définition d'un langage Neverlang 64

5.1 L'architecture modulaire de XMG 71
5.2 L'architecture du compilateur 75
5.3 L'architecture de l'exécuteur 75

6.1 Un exemple de fragment d'arbre 78
6.2 Un exemple de fragment d'arbre décoré par des structures de traits 78
6.3 Une grammaire hors contexte pour des descriptions d'arbres décorées par des structures de traits 79
6.4 Connexion de briques au moyen de non-terminaux externes 81
6.5 Un assemblage utilisant plusieurs instances d'une même boîte 82
6.6 Connexion de briques pour des structures récursives 83
6.7 Assemblage de briques pour la description de structures de traits intégrant des disjonctions atomiques 83

7.1 Le compilateur 88
7.2 L'architecture du compilateur 88
7.3 L'architecture du compilateur et les langages manipulés 89
7.4 Une bibliothèque de variables attribuées pour les structures de traits 92
7.5 Extrait du code d'un typer (syn) 107
7.6 extrait du code d'un unfolder (avm) 114
7.7 Code généré pour une description syntaxique 118
7.8 Exemple de classe contenant une description syntaxique 119
7.9 Code généré pour une classe 120
7.10 Code généré pour une classe (après interprétation des EDCG) 120
7.11 Exemple de fichier `edcg.yap` généré 124
7.12 L'assemblage d'un compilateur recréant le XMG existant 125
7.13 Connexions entre briques 126
7.14 L'architecture d'une brique (avm) 127
7.15 L'architecture d'une brique (syn) 128
7.16 Structure d'une brique d'assemblage (synsem) 129

8.1 L'exécuteur 132
8.2 L'architecture du générateur 132
8.3 L'architecture du générateur de XMG-1 133
8.4 Les étapes de la résolution 136
8.5 Les étapes de la résolution avec plugins 136
8.6 Structure d'une brique de solveur (tree) 139
8.7 L'architecture d'une brique (syn) 139
8.8 Structure d'une brique d'assemblage (synsem) 140

9.1 L'architecture modulaire de XMG 142
9.2 L'architecture modulaire de Neverlang 143

10.1 Arbres élémentaires colorés 152
10.2 Code XMG pour les arbres élémentaires colorés 153
10.3 Fichier `compiler.yaml` pour l'assemblage du compilateur `synsem` 155
10.4 Un extrait de grammaire d'interaction 156

10.5 Un extrait de grammaire d'interaction (suite) 157
10.6 Une analyse par une grammaire d'interaction 158
10.7 Code XMG pour une description d'arbre en grammaire d'interaction 159
10.8 Fichier `lang.def` de la brique `polAvm` 159
10.9 Fichier `compiler.yaml` pour l'assemblage du compilateur `ig` 160

11.1 Métagrammaire de la morphologie verbale de l'Ikota 164
11.2 Une dérivation avec succès 165
11.3 Une dérivation avec échec: conflits sur temps et prog 165
11.4 Le fichier `lang.def` de la brique `fields` 166
11.5 Exemple de définition d'une suite de champs ordonnés 166
11.6 Exemple de contribution à la dimension `<morph>` 167
11.7 Fichier `lang.def` de la brique `morph` 167
11.8 Fichier `compiler.yaml` pour l'assemblage du compilateur `morphtf` 168

12.1 Un patron d'arbre associé à un syntagme, sa contrepartie sémantique et la hiérarchie de type associée 172
12.2 Factorisation métagrammaticale du patron d'arbre et de la frame de la figure 12.1. . 174
12.3 Spécification de la frame de POConstr 175
12.4 Exemple de spécification de contraintes sur les types de structures de traits 175
12.5 Fichier `lang.def` pour les hiérarchies de types 176
12.6 Fichier `lang.def` pour la dimension `<frame>` 177
12.7 Fichier `compiler.yaml` pour l'assemblage du compilateur `synframe` 178

1 Les points de modularité de XMG 180

2 Une représentation arborescente d'une séquence d'accords 189
3 Combinaison de fragments pour un accord en XMG 190
4 Les 21 notes générées 192
5 Les 17 tierces majeures générées 193
6 Les 18 tierces mineures générées 193
7 Les 17 triades majeures et mineures générées 194
8 Les 15 accords de septième diminuée générés 195
9 Do majeur, et son relatif, la mineur 195

Introduction

La communauté se développant autour d'un logiciel est un élément décisif pour son évolution. Elle l'est en particulier dans le monde du logiciel libre, dans lequel les utilisateurs peuvent participer au développement de nouvelles fonctionnalités, rapporter des erreurs, ou simplement distribuer le logiciel pour étendre la communauté.

Dans le cas des langages de programmation, les différentes communautés peuvent se former selon les paradigmes de programmation, puis selon les langages qui implémentent ces paradigmes. Selon les besoins de l'utilisateur, le langage le plus adapté, soit celui qui lui permet de décrire le plus instinctivement son programme, peut différer. Il peut également devoir alterner entre plusieurs langages ou les combiner lorsque leur expressivité n'est plus adaptée.

L'objectif de la thèse est de montrer que l'idée d'adapter le langage à l'utilisateur, plutôt que de laisser l'utilisateur s'adapter au langage, est viable.

Le principal soucis de cette idée est que le développement de nouveaux compilateurs est une tâche pointue, qui demande de l'expertise. Notre hypothèse est que la modularité est la clé de ce problème. Ainsi, la méthode que nous proposons est basée sur le découpage d'un compilateur en briques composables et réutilisables.

Construire un compilateur adapté à ses besoins, basé sur un langage assemblé par ses soins, revient, si l'on s'appuie sur cette modularité, à combiner des briques. Le codage des processus de compilation est intégralement situé dans les briques, ce qui signifie que l'on peut créer un compilateur pour un nouveau langage sans devoir connaître précisément son comportement lors des différentes phases de la compilation, ni la structure des objets qui y sont manipulés.

L'objectif est donc double : d'une part, il s'agit de prouver que l'on peut assembler un compilateur de façon modulaire. D'autre part, l'objectif est de justifier que le découpage du compilateur en modules peut être essentiellement basé sur le langage qu'il reconnaît.

L'extensibité des compilateurs est également facilitée par cette approche. Coder une nouvelle fonctionnalité du compilateur, par exemple le support de nouvelles instructions, correspond à la création d'une brique. Atteindre une indépendance totale entre les briques signifie qu'une nouvelle brique peut être conçue sans devoir prendre en considération son interface avec les briques existantes. L'assemblage de cette nouvelle brique à un compilateur existant constitue une extension.

Une méthode de génération de compilateurs est en général orientée vers un type spécifique de compilateurs. Dans cette thèse, nous proposons un compilateur de compilateurs dans le cadre de la génération de ressources langagières. Cette tâche implique de pouvoir décrire des structures, parfois complexes ou nombreuses, permettant de modéliser la langue naturelle. Les langages reconnus par ces compilateurs sont donc des langages dédiés (ou DSL, Domain Specific Languages). Nous détaillons l'approche permettant de décomposer un compilateur de ce type en

parties élémentaires. Nous donnons également les méthodes pour contribuer à ce compilateur, en expliquant comment les briques sont constituées.

Les travaux présentés dans cette thèse n'ont pour essentielle ambition que de contribuer au domaine de la génération de ressources linguistiques. Les langages dédiés que nous assemblerons sont spécifiques à la descriptions des règles qui composent les langues. En conséquence, le nombre d'instructions que manipulent les compilateurs assemblés par notre méthode ainsi que leur optimisation en terme de performances sont des critères que nous ne considérons pas comme prioritaires.

Dans une première partie, nous présenterons l'état de l'art, et le contexte motivant le besoin de modularité. Dans la deuxième partie, nous nous détaillerons la contribution, c'est à dire la méthode de découpage des langages de description et de la chaîne de compilation. Enfin, dans la troisième partie, nous nous concentrerons sur la validation de l'approche, en composant modulairement plusieurs compilateurs dédiés à différentes taches de génération de ressources langagières.

Première partie

État de l’Art

Des langages dédiés au Traitement Automatique des Langues

1

Sommaire

1.1 Objet d'étude : la langue naturelle **3**

1.2 Des outils formels pour décrire la langue **4**

1.2.1 Plusieurs niveaux d'analyse 4

1.2.2 Des interfaces entre les niveaux 8

1.3 Un besoin de langages dédiés **13**

Ce chapitre présente les problématiques liées à l'utilisation de langages dédiés dans le cadre du traitement automatique des langues. Dans un premier temps, nous définirons l'objet d'étude, soit le traitement de la langue naturelle. Dans un second temps, nous insisterons sur les différents aspects faisant de la description de ressources langagières une tâche complexe, nécessitant des outils dédiés.

1.1 Objet d'étude : la langue naturelle

Le cadre de la thèse est le traitement automatique des langues (TAL, voir [Clark *et al.*, 2010]), dont le but ultime est de permettre à la machine de manipuler le même langage que les êtres humains. Ceci implique un grand nombre d'applications, qui nécessitent des informations sur la langue. Pour pouvoir mener à bien ces tâches, on peut envisager deux approches principales. Dans la première (bottom up), on part d'exemples pour en extraire la connaissance linguistique par apprentissage (voir par exemple [Mitchell, 1997]). Dans la seconde, on donne une description de la connaissance linguistique, par exemple des règles de grammaires élaborées par un spécialiste (voir par exemple [Mitkov, 2003, chap. 4, 8, 12, 18, 26]).

Pour l'approche bottom up, l'avantage est de disposer de nombreux corpus, représentatifs de la langue, et d'être aujourd'hui capable de manipuler efficacement les gros volumes de données que constituent ces corpus. Par exemple, dans le cadre de la traduction automatique, les approches de ce type sont plutôt répandues, en particulier depuis les années 1990 (avec [Brown *et al.*, 1993]). Elles reposent généralement sur un apprentissage depuis deux corpus alignés, c'est à dire des bases de phrases de la langue source et leurs traductions dans la langue cible. Malgré son efficacité, une approche basée sur l'apprentissage est néanmoins limitée, notamment en raison de la forte dépendance au corpus.

Dans le cas du top down, la représentation de la langue est disponible, mais il faut préalablement la formaliser pour qu'elle soit manipulable par la machine. Dans la suite du document c'est à des approches de ce type que nous nous intéresserons.

Qu'est ce que décrire la langue ? La réponse à cette question dépend du niveau de représentation linguistique où l'on choisit de se placer.

Le premier niveau que l'on pourrait citer est la morphologie (voir par exemple [Spencer et Zwicky, 2001]), qui consiste à étudier la formation des mots. L'une de ses applications est l'analyse morphologique, qui vérifie qu'une forme d'un mot est valide, et en extrait de l'information. Par exemple, "suivraient" est une forme du verbe suivre, son mode est le subjonctif, son temps le présent, et elle correspond à la troisième personne du pluriel. La tâche inverse est la génération de formes fléchies. Un programme dédié à cette tâche prend en entrée une description de mot, et construit tous les mots la réalisant. Par exemple, on peut souhaiter générer toutes les formes plurielles du verbe "suivre" (soit à tous les modes, tous les temps, toutes les personnes).

L'étude de la formation de phrases, soit la syntaxe, est un domaine qui tient une place importante dans le traitement automatique des langues. Elle est notamment essentielle dans des applications très répandues, comme la traduction automatique ([Koehn, 2010]), les systèmes de dialogue ([Łupkowski et Purver, 2010]), les systèmes de question/réponse ([Harabagiu et Moldovan, 2003]), la correction grammaticale ...

La sémantique désigne l'étude du sens des phrases, ou des mots. Elle est l'un des éléments qu'il peut être intéressant de construire lors d'une analyse syntaxique, ou morphologique. C'est le cas dans les systèmes de dialogue, où il est courant d'évaluer le sens d'une phrase pour pouvoir y répondre de façon pertinente. Cette réponse peut d'ailleurs être générée à partir d'une représentation sémantique.

Pour pouvoir manipuler le langage naturel, l'ordinateur doit utiliser des représentations des éléments linguistiques. Il doit par exemple être capable, pour une phrase donnée en entrée, d'extraire les informations nécessaires à une tâche, qui peut par exemple être une traduction de cette phrase vers une autre langue. Les informations typiquement extraites sont une analyse syntaxique de la phrase (reconnaître les fonctions des différents constituants de la phrase) ou/et une représentation du sens de la phrase.

1.2 Des outils formels pour décrire la langue

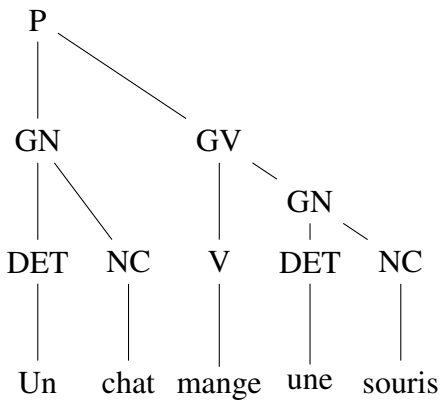
Pour parvenir à une description de la langue, dont nous avons évoqué les différents niveaux de représentation, il est nécessaire d'avoir des structures adaptées. Nous verrons que ces structures sont hétérogènes, de par cette diversité des niveaux d'étude, mais également celle des théories qui leurs sont associées. Enfin, nous discuterons le problème des interactions entre les niveaux d'étude.

1.2.1 Plusieurs niveaux d'analyse

Même après avoir choisi un niveau de représentation linguistique, il existe plusieurs outils pour le décrire. Nous illustrerons ceci sur deux exemples de représentation de la syntaxe et de la sémantique, puis nous analyserons les cloisonnements entre les niveaux de représentation.

Analyse par Arbre Syntagmatique

Une façon courante de représenter la syntaxe est ce que l'on appelle l'analyse par arbres syntagmatiques (ou par constituants, voir [Chomsky, 1957]), qui associe à chaque constituant de la phrase sa catégorie syntaxique. La phrase "un chat mange une souris" peut être analysée par un arbre dont la racine représente la phrase (de catégorie syntaxique *P*), et les feuilles les mots. Les noeuds non-feuilles correspondent aux syntagmes (par exemple, un syntagme nominal, qui peut se réaliser comme ici par un déterminant suivi d'un nom commun).



Cette analyse signifie que la phrase est composée d'un syntagme nominal (noeud *GN*) suivi d'un syntagme verbal (*GV*). Le syntagme nominal se compose d'un déterminant (*DET*) et d'un nom commun (*NC*), correspondant respectivement aux mots *un* et *chat*. Le syntagme verbal regroupe un verbe (*V*), relié au mot *mange*, et un groupe nominal, réalisé de la même façon que le précédent.

Pour arriver à une telle analyse, un certain nombre de théories ont été proposées. Avant d'analyser quelques unes d'entre elles en détail (dans le chapitre 3), prenons l'exemple d'un formalisme simple : les grammaires hors-contexte.

Grammaires hors-contexte. Parmi les systèmes de réécriture les plus connus, nous pouvons citer les grammaires algébriques (également appelées grammaires hors-contexte, décrites formellement dans [Autebert *et al.*, 1997]). Ces grammaires contiennent des règles de la forme $V \rightarrow w$, où V est un symbole dit non terminal, et w une chaîne composée de symboles dits terminaux et/ou non terminaux. Dans ce système de réécriture, on part d'un symbole non-terminal particulier, appelé axiome, et on essaye d'appliquer les règles jusqu'à produire une suite de symboles terminaux. Cette suite de symboles correspond à un énoncé défini par notre grammaire, en d'autres termes, à un énoncé grammatical.

De manière plus générale, on définit une grammaire algébrique au moyen des éléments suivants :

— un ensemble fini de symboles terminaux, notés conventionnellement par des minuscules (ici les mots de notre vocabulaires),
— un ensemble fini de symboles non-terminaux, notés conventionnellement par des majuscules (ici les catégories syntaxiques des constituants de la phrase, tels que Groupe Verbal, Groupe Nominal, etc),
— un élément de l'ensemble des non-terminaux appelé axiome (ici P pour Phrase),

— un ensemble de règles de production, qui sont des paires formées d'un non-terminal et d'une suite de terminaux et de non-terminaux comme vu ci-dessus.

Pour fixer les idées, considérons l'exemple de grammaire algébrique ci-dessous. Celui-ci permet notamment d'engendrer la phrase *un chat mange une souris* :

P	→	GN GV	GN	→	DET NC
GV	→	V GN	DET	→	un
DET	→	une	NC	→	chat
NC	→	souris	V	→	mange

FIGURE 1.1 — Une grammaire hors contexte pour un fragment du français

Dérivation. À partir de cette grammaire, la phrase *un chat mange une souris* est engendrée en appliquant les règles de réécritures, en partant de l'axiome (symbole *P*). Lors de l'application successive des règles de réécriture, on peut construire une analyse appelée arbre de dérivation. Cet arbre a pour racine l'axiome de la grammaire, et pour feuilles les mots de la phrase. Les règles de réécriture sont transcrites dans cet arbre : un noeud correspond à la partie gauche, ses fils à la partie droite.

L'arbre de dérivation menant à cette phrase est donné ci-dessous :

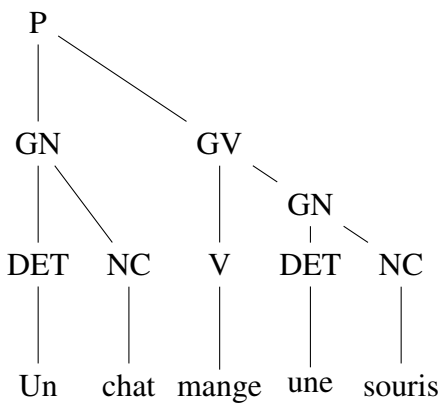


FIGURE 1.2 — Un arbre de dérivation pour la phrase "Un chat mange une souris"

Comme nous le verrons plus tard, les grammaires hors contexte ne disposent pas d'une expressivité suffisante pour décrire certains phénomènes linguistiques. C'est pourquoi des formalismes plus expressifs, et plus complexes, ont fait leur apparition.

Parmi eux, on peut citer les grammaires d'arbres adjoints, basées sur la réécriture d'arbres, que nous présenterons dans la section 3.3.

Analyse par Dépendances

Une alternative à l'analyse par constituants est l'analyse par dépendances (voir [Tesnière, 1959]). Dans le cas des grammaires de dépendances, l'analyse est un arbre dont la racine correspond au mot principal de la phrase (traditionnellement, le verbe), et chaque arête à une relation de

dépendance d'un constituant par rapport à un autre. Les arêtes sont étiquetées par la fonction grammaticale qu'elles symbolisent.

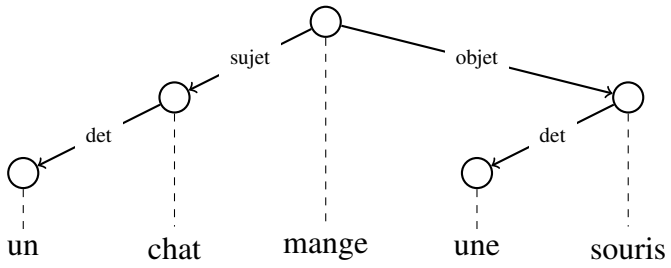


FIGURE 1.3 — Un graphe de dépendance pour la phrase "Un chat mange une souris"

On indique sur cette figure que *chat* et *souris* sont respectivement le sujet et l'objet du verbe *mange*, et que *un* et *une* sont les déterminants de *chat* et *souris*.

Une analyse en dépendance nécessite des structures différentes des règles de réécriture présentées précédemment (présentées dans la sous-section 3.4.1), ce qui contribue encore une fois à une hétérogénéité des ressources.

Description de la Sémantique

La sémantique traite de la représentation du sens d'un énoncé. Dans un contexte de traitement automatique, cette représentation est formalisée, dans le but de permettre son traitement informatique. Il est courant d'utiliser une formule logique, qui associe des prédicats aux concepts sémantiques (voir [Blackburn et Bos, 2005]). Dans notre exemple, les concepts à représenter sont la notion d'appartenance pour un individu à la race des chats ou des souris (par le biais des prédicats *chat* et *souris*), ainsi que l'action de manger.

$$chat(X), souris(Y), mange(X, Y)$$

où le prédicat *mange* s'applique à deux variables logiques X et Y, représentant l'acteur et l'objet de l'action "manger".

Une autre approche propose de modéliser le sens sous forme de liens entre concepts. C'est le cas par exemple des réseaux sémantiques ([Sowa, 1991]). On peut ainsi représenter le sens de l'énoncé précédent à l'aide d'un graphe, dont les arêtes indiquent les relations sémantiques entre les objets.

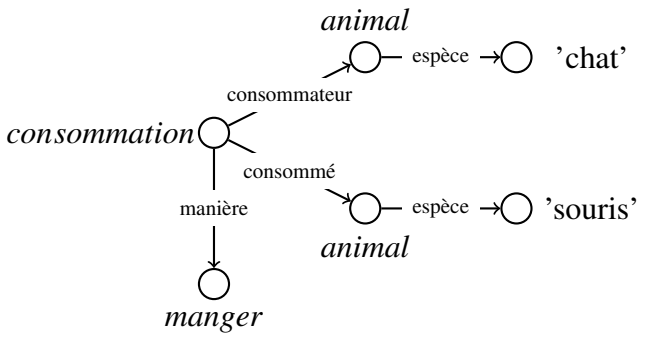


FIGURE 1.4 — Une représentation sémantique sous la forme d'un graphe

Cette représentation de concepts cognitifs représente un évènement, une action de consommation (la racine). Elle a pour manière *manger*, et ses acteurs sont deux animaux (le consommateur et le consommé). Les espèces de ces deux animaux sont respectivement "chat" et "souris". On peut noter que ces deux approches (formules logiques et réseaux sémantiques) ne sont pas si éloignées puisqu'un réseau sémantique peut être décrit par une formule logique puis interprété (voir par exemple [Kumar, 1990]).

1.2.2 Des interfaces entre les niveaux

Nous avons observé la richesse dans les manières de décrire les niveaux, nous allons maintenant voir que les descriptions de plusieurs niveaux sont souvent liées et nécessitent des interfaces.

La morphologie interagit avec la syntaxe, notamment lorsque l'on souhaite tenir compte des accords entre les constituants. En effet, malgré le fait que la phrase "un chat manges une souris" présente la même suite de catégories syntaxiques pour ses mots que la phrase initiale, elle n'est pas syntaxiquement correcte (le sujet est à la première personne du singulier, le verbe à la deuxième).

Pour cette raison, les règles de syntaxe sont donc en général augmentées d'information morphosyntaxique, reflétant les contraintes supplémentaires apportées par la morphologie des mots. Cette information est habituellement présente sous la forme de traits.

Les Structures de Traits

Les structures de traits sont des ensembles de couples attribut-valeur, assimilables aux enregistrements dans les langages de programmation. La représentation que nous utiliserons pour les structures de traits est également appelée matrice de traits (Attribute Value Matrix, ou AVM).

Les structures de traits suivantes donnent des informations morphosyntaxiques sur le déterminant "le" et le nom commun "chat".

$$\text{UN}\begin{bmatrix}\text{CAT} & \text{det} \\ \text{ACCORD} & \begin{bmatrix}\text{GEN} & \text{m} \\ \text{NUM} & \text{sg}\end{bmatrix}\end{bmatrix} \qquad \text{CHAT}\begin{bmatrix}\text{CAT} & \text{n} \\ \text{ACCORD} & \begin{bmatrix}\text{GEN} & \text{m} \\ \text{NUM} & \text{sg}\end{bmatrix}\end{bmatrix}$$

"un" a pour catégorie syntaxique (*CAT*) *det* (pour déterminant), et pour accord une nouvelle structure de traits. Cette dernière indique que le genre (*GEN*) est masculin (*m*) et le nombre (*NUM*) singulier (*sg*). "chat" a pour catégorie syntaxique *n* (pour nom) et les mêmes traits d'accord.

Pour représenter le partage d'information entre différents traits (éventuellement issus de structures différentes), on utilise des variables de coréférence, que l'on représente par des nombres encadrés.

$$\text{DET}\begin{bmatrix} \text{CAT} & \text{det} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \boxed{1} \\ \text{NUM} & \boxed{2} \end{bmatrix} \end{bmatrix} \qquad \text{N}\begin{bmatrix} \text{CAT} & \text{n} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \boxed{1} \\ \text{NUM} & \boxed{2} \end{bmatrix} \end{bmatrix}$$

où $\boxed{1}$ et $\boxed{2}$ sont des variables de coréférence indiquant l'identité entre les genres et les nombres du nom et du déterminant.

Les structures de traits sont associées aux structures syntaxiques, et peuvent être utilisées comme des filtres : quand deux structures de traits sont incompatibles, les structures auxquelles elles sont associées (par exemple un noeud de l'arbre syntaxique) le sont aussi. Ce mécanisme d'unification entre les traits est le concept fondamental des grammaires d'unification, dont nous présenterons quelques uns des représentants les plus célèbres dans le chapitre 3.

Enrichir les structures. Les structures de traits et leur mécanisme d'unification permettent de ne pas se soucier de l'ordre des informations (la structure n'est pas ordonnée), et de ne donner que l'information connue sur les structures, sans avoir à mentionner de valeurs indéterminées pour les traits dont on ignore la valeur. La description partielle ainsi donnée peut être enrichie par unification. C'est par exemple le cas pour la forme nominale *souris*, pour laquelle on ne peut définir le nombre que d'après le contexte. Le déterminant *une* a quand a elle une structure de traits complètement spécifiée concernant son genre et son nombre.

$$\begin{bmatrix} \text{CAT} & \text{det} \\ \text{ACCORD} & \begin{bmatrix} \text{NUM} & \text{sg} \\ \text{GEN} & \text{f} \end{bmatrix} \\ \text{LEX} & \text{une} \end{bmatrix} \qquad \begin{bmatrix} \text{CAT} & \text{n} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \text{f} \end{bmatrix} \\ \text{LEX} & \text{souris} \end{bmatrix}$$

La structure correspondant à un groupe nominal contient des variables de coréférence, permettant, dès qu'une variable est instanciée (par unification), de propager les valeurs dans d'autres traits.

$$\begin{bmatrix} \text{CAT} & \text{gn} \\ \text{ACCORD} & \begin{bmatrix} \text{NUM} & \boxed{1} \\ \text{GEN} & \boxed{2} \end{bmatrix} \\ \text{DET} & \begin{bmatrix} \text{CAT} & \text{d} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \boxed{2} \\ \text{NUM} & \boxed{1} \end{bmatrix} \end{bmatrix} \\ \text{NOM} & \begin{bmatrix} \text{CAT} & \text{n} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \boxed{2} \\ \text{NUM} & \boxed{1} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

FIGURE 1.5 — Représentation d'un groupe nominal

Ici, on s'assure que les genres et nombres du groupe nominal, du déterminant, et du nom sont cohérents (dans le cas présent, identiques).

Le groupe nominal "une souris" est associé à la structure suivante, calculée par unification (la valeur de *det* avec la structure de *une*, celle de *nom* avec la structure de *souris*, et les valeurs de *gen* et *num* du nom, du déterminant et du groupe nominal entre elles).

$$\begin{bmatrix} \text{CAT} & \text{gn} \\ \text{ACCORD} & \begin{bmatrix} \text{NUM} & \text{sg} \\ \text{GEN} & \text{m} \end{bmatrix} \\ \text{DET} & \begin{bmatrix} \text{CAT} & \text{d} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \text{f} \\ \text{NUM} & \text{sg} \end{bmatrix} \\ \text{LEX} & \text{une} \end{bmatrix} \\ \text{NOM} & \begin{bmatrix} \text{CAT} & \text{n} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \text{f} \\ \text{NUM} & \text{sg} \end{bmatrix} \\ \text{LEX} & \text{souris} \end{bmatrix} \end{bmatrix}$$

FIGURE 1.6 — Représentation du groupe nominal "une souris"

En replaçant ceci dans le contexte des règles de réécriture, la règle hors contexte initiale

$$GN \rightarrow DET \;\; N$$

peut être enrichie pour traiter l'accord en :

$$GN \rightarrow \begin{bmatrix} \text{CAT} & \text{gn} \\ \text{ACCORD} & \begin{bmatrix} \text{NUM} & \boxed{1} \\ \text{GEN} & \boxed{2} \end{bmatrix} \\ \text{DET} & \boxed{3} \\ \text{NOM} & \boxed{4} \end{bmatrix} \qquad DET \; \boxed{3}\begin{bmatrix} \text{CAT} & \text{det} \\ \text{ACCORD} & \begin{bmatrix} \text{NUM} & \boxed{1} \\ \text{GEN} & \boxed{2} \end{bmatrix} \end{bmatrix} \qquad N \; \boxed{4}\begin{bmatrix} \text{CAT} & \text{n} \\ \text{ACCORD} & \begin{bmatrix} \text{NUM} & \boxed{1} \\ \text{GEN} & \boxed{2} \end{bmatrix} \end{bmatrix}$$

Les règles lexicales, c'est à dire celles auxquelles on a associé un élément du lexique, sont adaptées de la même manière, par exemple :

$$N \rightarrow \begin{bmatrix} \text{CAT} & \text{n} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \text{f} \end{bmatrix} \\ \text{LEX} & \text{souris} \end{bmatrix} \qquad souris \; \begin{bmatrix} \text{CAT} & \text{n} \\ \text{ACCORD} & \begin{bmatrix} \text{GEN} & \text{f} \end{bmatrix} \\ \text{LEX} & \text{souris} \end{bmatrix}$$

La syntaxe et la morphologie ne sont pas les seuls niveaux de description linguistique qui fonctionnent de pair. Lors d'une étude morphologique, il est parfois indispensable de s'intéresser également à la phonologie. C'est le cas lorsque la forme fléchie (par exemple, un verbe conjugué) n'est pas identique à la combinaison des morphèmes qui devraient la composer, et que des transformations sont nécessaires pour passer de l'une de ces formes à l'autre. Par exemple, la règle du pluriel en français, qui consiste à ajouter le suffixe "s", produit des formes telles que "oss" ou "viss" pour les mots terminant en "s" au singulier. C'est une règle phonologique qui crée la forme fléchie.

Interface syntaxe-sémantique

Influencés par l'héritage de Montague ([Montague, 1970]), les travaux concernant la sémantique des phrase reposent généralement sur des formules logiques calculées à partir de la syntaxe. Pour cela, [Montague, 1970] propose d'associer à chaque entrée lexicale de la grammaire une représentation sémantique, et à chacune de ses règles syntaxiques une règle de calcul sémantique. Cette règle permet d'obtenir à partir des représentations associées aux constituants une représentation du sens de leur combinaison.

Le calcul sémantique présenté dans ce même travail est basé sur le lambda-calcul. Les représentations sémantiques sont alors des lambda-termes (contenant généralement une lambda-abstraction), et les règles sémantiques des applications fonctionnelles permettant d'obtenir un lamba-terme par l'opération de réduction.

On pourrait associer à l'entrée lexicale "mange" le lambda-terme suivant :

$$mange \rightarrow \lambda x.manger(x)$$

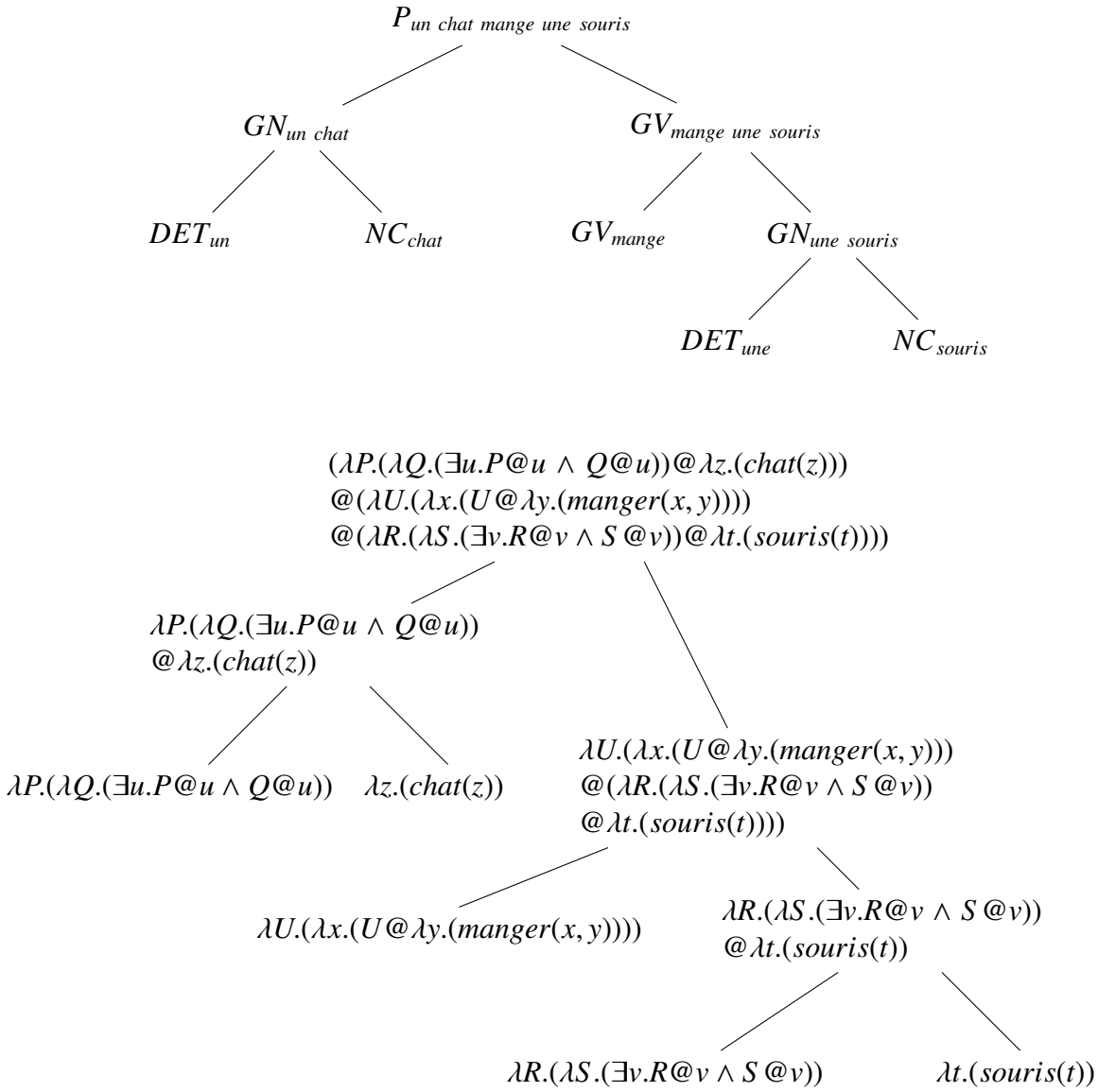
dans lequel le prédicat *manger* est abstrait par la variable x, symbolisant son argument.

Le nom commun *chat* peut quant à lui être associé à l'abstraction suivante :

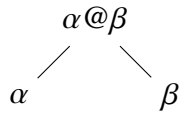
$$chat \rightarrow \lambda P.P@chat$$

où @ est l'application fonctionnelle.

L'analyse sémantique est faite à partir d'un arbre syntaxique qui doit être binaire (pour cela, la grammaire hors-contexte utilisée doit être en forme normale). Pour la phrase "le chat mange la souris", on peut obtenir la structure suivante :



Le principe du calcul sémantique est le suivant : pour chaque noeud non feuille, la formule qui lui est associée est le résultat d'une application fonctionnelle entre celles associées à ses deux fils. Ceci peut être schématisé de la manière suivante :



La formule associée à la racine peut maintenant être simplifiée par β-réduction. Cette opération correspond au remplacement de toutes les occurences d'une variable abstraite par un argument. La β-réduction produit dans notre cas la formule :

$$(\exists x.chat(x) \wedge (\exists y.souris(y) \wedge manger(x, y)))$$

Pour une analyse fine, plusieurs niveaux d'analyses doivent donc cohabiter, rendant nécessaires différents types de ressources. La diversité de ces ressources les rend difficiles à décrire de manière homogène. L'hétérogénéité des structures implique donc souvent l'hétérogénéité des outils de génération et celle des formats dans lesquels ces structures sont décrites.

1.3 Un besoin de langages dédiés

On peut imaginer que l'utilisation d'un format générique, comme XML ou JSON, est la solution à l'hétérogénéité que nous avons notée. Cependant, si ils permettent en effet de décrire la plupart des ressources linguistiques (les treebanks par exemple), leur syntaxe n'est pas adaptée à l'utilisateur. L'écriture des balises et la redondance importante sur l'ensemble d'un fichier en font une ressource qu'il est préférable de générer automatiquement. Si cette génération ne se fait pas par apprentissage sur un corpus, elle doit se faire d'après une description de la grammaire, dans un langage dédié.

Au même titre que dans le cadre du développement de programmes informatiques, avoir un cadre de développement adapté, proposant des outils d'abstraction et un langage de description adapté aux structures que l'on manipule offre un confort de développement et de maintenance bien supérieur, d'autant plus si la taille de ressource est importante.

Nous avons montré que les ressources nécessaires aux tâches du traitement automatique des langues pouvaient prendre des formes très diverses, notamment du fait de l'absence de consensus quant à leur représentation. Nous avons également noté que les cloisonnements entre les niveaux de description linguistique étaient difficile à définir, voire inexistants. C'est donc vers une solution adaptable, configurable, permettant de décrire et de faire interagir plusieurs niveaux, que nous souhaitons nous diriger.

Dans les chapitres suivants, nous nous intéresserons à deux niveaux de description linguistique en particulier : la morphologie et la syntaxe. Pour chacun de ces deux niveaux, nous citerons un ensemble de théories linguistiques utilisant différents types de structures. Pour chacune de ces théories, nous étudierons un ensemble d'approches proposées pour leur description. Enfin, le dernier chapitre concernera les techniques proposées pour la génération de compilateurs, et en particulier les langages de description qu'elles impliquent.

Décrire la Morphologie

2

Sommaire

2.1 Introduction **15**

2.2 Analyse morphologique par unification **16**

2.2.1 PFM 16

2.2.2 Network Morphology 17

2.3 Analyse morphologique par automates à états finis **18**

2.3.1 Morphologie à Deux Niveaux 19

2.3.2 PARSLI 20

2.4 Conclusion **21**

2.1 Introduction

Historiquement, et avant que la syntaxe ne la remplace dans ce rôle, la morphologie a longtemps été considérée comme le principal domaine de la linguistique. Elle peut être caractérisée comme l'étude des relations entre son et sens à l'intérieur du mot [Fradin, 2003]. La morphologie peut se diviser en deux tâches. La première est appelée morphologie flexionnelle ([Geeraerts *et al.*, 2010]) , et consiste à étudier les modifications opérées sur les mots pour qu'ils s'accordent avec le reste d'un énoncé. Si l'on prend l'exemple du verbe "terrasser", on souhaite que "je le terrasse" soit considéré comme valide, mais pas "je le terrassons".

La seconde tâche est la morphologie constructionnelle, ou dérivationnelle (voir par exemple [Lieber et Štekauer, 2014]). Elle désigne l'étude des constructions de nouveaux mots à partir de mots existants. Le nouveau mot possède des traits différents (la catégorie syntaxique par exemple) ou un sens modifié. Par exemple, à partir du nom "fait", on forme l'adjectif "factuel", puis l'adverbe "factuellement".

Principalement, deux théories morphologiques se sont succédées : la morphologie morphématique combinatoire, ou morphologie classique, et la morphologie lexematique. L'unité minimale de sens, essentielle pour l'aspect sémantique, change avec ces théories. Le morphème, en morphologie morphématique combinatoire, est assemblé avec d'autres morphèmes. Cette approche est adaptée à la description de langues agglutinantes, mais connaît des limites lorsque certaines affixes ne correspondent pas à des unités de sens (morphe vide) ou qu'un sens est exprimé sans substrat phonique (morphème zéro). Les morphèmes zéro apparaissent notamment dans des

constructions où les patrons sont fixes, comme dans la conjugaison espagnole :

$$cant - a - \emptyset - mos \qquad cantamos\ (nous\ chantons)$$
$$cant - \emptyset - \emptyset - o \qquad canto\ (je\ chante)$$

où Øcorrespond au morphème zéro. La deuxième position contient la marque du mode, ici indicatif (non exprimée à la première personne du singulier), la troisième celle du temps, ici présent (qui n'a pas de marque à l'indicatif).

En morphologie lexematique, le morphème laisse sa place d'unité minimale de sens au lexème, modifié par des fonctions.

Nous avons vu que les manières de décrire chaque niveau de représentation étaient diverses, nous allons dans ce chapitre observer plus en détails un certain nombre d'approches pour la description des phénomènes morphologiques. Si dans le chapitre précédent, nous ne nous sommes intéressés qu'à l'outil formel de description, nous irons ici plus loin en donnant un ensemble d'outils implémentant ces approches, et les langages de description qui leurs sont associés.

2.2 Analyse morphologique par unification

2.2.1 PFM

Le modèle Paradigm Function Morphology [Stump, 2001a] considère que le système flexionnel d'une langue peut être modélisé par une fonction. Cette fonction associe à un lexème et un ensemble de traits morphosyntaxiques une flexion. La fonction paradigmatique se compose d'un ensemble de règles de réalisation organisées en blocs. Ces règles se classent en trois catégories : les règles de choix du stem de référence, les règles d'exposants, et les règles de renvoi.

Un exemple de règle d'exposant est celle d'apparition en anglais du "s" à la troisième personne du singulier au présent. Le radical, par exemple "run", reçoit le suffixe "s" dans le cas où ses traits comprennent `pers=3`, `num=sg` et `tense=present`.

Intéressons nous maintenant à un outil implémentant cette formalisation, PFME, fourni par Cat's Claw (Computer-Assisted Technology Service Computational Linguist's Automated Workbench, http://www.cs.uky.edu/~raphael/linguistics/claw.html).

La déclaration d'un lexème dans cet environnement se fait comme dans l'exemple suivant :

```
Lexeme: EAT
Meaning: eat
Syntactic category: V
Inflection class: n
```

On associe à un lexème une sémantique, éventuellement composée de plusieurs mots, une catégorie syntaxique, et un ensemble de classes d'inflexion.

Les racine du lexème sont exprimées comme dans l'exemple suivant, en associant à un lexeme et des contraintes morphosyntaxiques une forme :

```
Root(<EAT,{past}>) = ate
Root(<EAT,{perfect/futPerf}>) = eaten
Root(<EAT,{}>) = eat
```

Ces règles signifient que le lexème `EAT` a pour racine `ate` au passé simple, `eaten` au passé et au futur parfaits, et *eat* dans les autres cas. Le choix de l'alternative utilisée se fait par précédence de Panini, c'est à dire sur l'alternative la plus restrictive (celle qui a le plus de contraintes parmi celles compatibles).

Un bloc exprimant des règles d'exposant consiste en un identifiant, et un ensemble de règles formées à gauche de l'identifiant du bloc, de la lettre X désignant l'entrée et de contraintes morphosyntaxiques, et à droite de la forme modifiée par l'exposant de l'entrée.

```
Block I
  I, X,{3 sg present} -> Xs
  I, X,{perfect/past/futPerf} -> Xed
```

Dans cet exemple, on exprime le fait qu'au présent, un verbe se voit associé le suffixe "s" à la troisième personne au présent, et le préfixe "ed" au passé simple et au passé parfait, et au futur parfait.

PFME permet de décrire les règles propres au modèle PFM efficacement, notamment grâce à la possibilité d'utiliser des disjonctions.

2.2.2 Network Morphology

La Network Morphology [Brown et Hippisley, 2012] est une approche déclarative à la morphologie flexionnelle.

Cette approche s'inspire de DATR ([Evans et Gazdar, 1989]), qui est un langage formel pour la description de réseaux d'héritage. Son concept central est nommé Théorie. Il s'agit d'un réseau de noeuds qui contiennent et partagent de l'information. Lorsque l'on définit un noeud, on définit sa place dans le réseau d'héritage. L'idée est de pouvoir capturer la redondance par des généralisations dans les entrées similaires. Le réseau présenté dans la figure 2.1 montre par exemple comment trois verbes (Walk, Jump et Run) peuvent hériter de traits communs fournis par Verb.

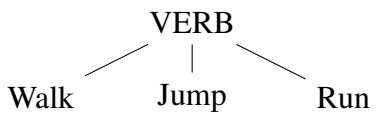


FIGURE 2.1 — Un réseau d'héritage DATR

Les deux entrées DATR suivantes permettent la construction d'un réseau d'héritage :

```
VERB:
<syntactic category> == verb
<present participle> == "<root>" ing
<past> == "<root>" ed.

Walk:
<> == VERB
<root> == walk.
```

Dans un premier temps, on définit l'entrée *VERB*, en lui donnant une catégorie syntaxique, ainsi que deux règles de suffixation, que nous analyserons dans ce qui suit. La seconde entrée, celle du verbe *Walk*, hérite de l'entrée *VERB* (`<> == VERB`), c'est à dire comprend toutes ses règles, en lui ajoutant celle de la racine verbale (`<root> == walk`).

Une requête DATR est l'association d'un lexème et d'une séquence de propriétés. Son évaluation consiste à appliquer la requête à toutes les règles hébergées par un noeud. Si deux règles ont une partie gauche identique, la seconde écrase la première (cette stratégie est utilisée dans DATR pour que les noeuds plus spécifiques puissent écraser de l'information contenue par des noeuds plus généraux lors de l'héritage).

Ajouter à la Théorie les deux entrées suivantes permet de tirer partie de la hiérarchie d'héritage pour éviter la redondance.

```
Jump:
<> == VERB
<root> == jump.

Run:
<> == VERB
<root> == run
<past> == ran
<past participle> == <root>.
```

Le réseau d'héritage résultant de ces nouvelles définitions est celui que nous avions présenté dans la figure 2.1.

Si DATR a été utilisé pour la description de la morphologie, certains aspects rendent son utilisation compliquée. Les traits morphosyntaxiques doivent par exemple être ordonnés, ce qui est une contrainte importante lors d'un passage à l'échelle (lorsque le nombre de traits devient important).

KATR ([Finkel et Stump, 2002]) est une extension de DATR, qui résout ce problème, et facilite la description de phénomènes de morphologie flexionnelle pour lesquels le langage DATR s'avérait trop peu adapté (notamment les exposants de genre, nombre et personne en Hébreu, voir [Finkel *et al.*, 2002]). Il intègre notamment des expressions régulières, et permet d'exprimer les traits sans se soucier de leur ordre. Sa syntaxe reste proche de celle de DATR, et son expressivité est identique.

Peu de ressources de tailles conséquentes ont été développées avec ces outils, contrairement aux approches que l'on explorera dans la section suivante. Ces approches sont basées sur des structures différentes : les automates finis.

2.3 Analyse morphologique par automates à états finis

Les approches purement informatiques à la morphologie flexionnelle sont principalement basées sur les automates d'état fini (XFST, [Beesley et Karttunen, 2003], ou Foma, [Hulden, 2009]). [Karttunen, 2003] a prouvé que si l'on ne s'intéresse qu'à la simple génération de paradigmes morphologiques, les approches formelles peuvent se réduire à des systèmes équivalents à des automates d'état fini.

```
Alphabet

a b c d e f g h i j k l m n o p q r s t u v w x y z N:m N:n ;

Sets
Consonant = b c d f g h j k l m n p q r s t v w x z m n ;
Vowel = a e i o u y ;

Definitions

ClosedSyllable = :Vowel+ [ ~:Vowel ]+ ;

Rules

"N:m␣before␣input-character␣p"
N:m <=> _ p: ;

"Degradation␣of␣p␣to␣m␣after␣input-character␣N"
p:m <=> N: _ ;
```

FIGURE 2.2 — Une grammaire HFST

2.3.1 Morphologie à Deux Niveaux

Les méthodes utilisant le concept de morphologie à deux niveaux, initialement introduit par [Koskenniemi, 1983], présentent l'avantage d'être bi-directionnelles, c'est à dire de permettre aussi bien l'analyse morphologique que la génération. La morphologie à deux niveaux consiste à considérer deux niveaux de représentation (appelés niveau de surface et niveau profond), et à établir des contraintes entre ces niveaux. Ces contraintes sont ensuite compilées en un automate d'état fini. L'avantage des contraintes sur les transformations est que l'ordre dans lequel on pose les contraintes n'influe pas sur le résultat. Dans le cas de transformations, l'ordre d'application est problématique.

Un exemple de phénomène que l'on peut représenter grâce aux deux niveaux est la transformation du *n* en *m* lorsqu'il est suivi d'un *p* (notamment pour le préfixe *in* qui devient *im* dans *impossible*). Pour cela, on utilise un caractère n'appartenant qu'au niveau profond (*N*), se réalisant en surface en *m* lorsqu'il est suivi de *p* et en *n* dans les autres cas. La forme profonde *iNcapable* se réalise ainsi en *incapable* et *iNpossible* en *impossible*.

Nous illustrerons ces approches avec l'outil HFST-TwolC ([Lindén *et al.*, 2009]). La définition d'une grammaire TwolC débute par celle de son alphabet, puis celle de ses diacritiques, de ses ensembles (de caractères), de ses définitions d'expressions régulières, et enfin de ses règles. La chaîne d'entrée des règles est celle correspondant à la forme profonde, et la forme de sortie celle correspondant à la forme de surface.

L'exemple de la figure 2.2 est basé sur celui de la documentation de HSFT (https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstTwolC) :

L'alphabet est une suite de couples de caractère d'entrée et de caractère de sortie (remplacés par un caractère unique dans le cas où ils sont identiques). Ici, `N` est donc un caractère n'ap-

partenant qu'au niveau profond, qui se réalise en surface en `m` ou `n`. `ClosedSyllable` est une expression régulière qui correspond à tous les motifs dont la forme de sortie débute par une suite de voyelles, soit les caractères appartenant à l'ensemble `Vowel`, suivie par une suite de consonnes (~ exprime la négation).

La règle nommée `"N:m␣before␣input-character␣p"` signifie qu'un `N` doit se réaliser en surface en `m` si il est suivi dans la forme profonde par un `p`.
`"Degradation␣␣of␣p␣to␣m␣after␣input-character␣N"` indique quant à elle qu'un `p` suivant immédiatement un `N` en forme profonde doit se réaliser en `m`. Ces règles, appliquées en parallèle, permettent de réaliser la forme lexicale *kaNpan* en *kamman* (génitif singulier du nom commun *kampa* , *peigne* en finnois).

Bien qu'ils soient capables de générer des paradigmes corrects, ces outils sont critiqués pour le fait que les descriptions qu'ils manipulent ne reposent pas suffisamment sur des intuitions linguistiques, soit l'expression de régularités ou d'irrégularités.

2.3.2 PARSLI

PARSLI (PARadigm Shape Lexicon Interface, [Walther, 2012]), dont l'implémentation se nomme Alexina$_{PARSLI}$ ([Sagot et Walther, 2013]), combine l'efficacité des approches informatiques et un cadre formel de description permettant d'exprimer les intuitions linguistiques qui font défaut à l'approche précédente.

PARSLI utilise le concept de zones de réalisation. Cette innovation permet de ne pas associer aux entrées lexicales une classe d'inflexion, mais plutôt un ensemble de zones de réalisation. Ces zones contiennent les règles de réalisation dédiées à des partitions spécifiques de son paradigme.

Chaque zone de réalisation appartient à une couche de réalisation. Ces couches, dont le nombre peut varier en fonction du cadre désiré, constituent une représentation structurée de la réalisation des formes. Elles appartiennent à trois catégories. L'une d'entre elles est la couche de stem. Le reste de l'architecture inclut un nombre arbitraire (éventuellement nul) de couches de thème et d'exposition. Pour obtenir la réalisation d'une forme, une règle de réalisation est appliquée dans chacune des couches.

L'utilité de ce concept s'illustre dans l'entrée de la figure 2.3, décrivant la flexion du verbe *Balayer* (exemple extrait de [Sagot et Walther, 2013]).

$$\begin{bmatrix} \text{I-PHON} & \text{balayer} \\ \text{I-CAT} & \text{verb} \\ \text{MSF} & \text{standard} \\ \text{S-STEM} & \text{(empty)} \\ \text{S-FORM} & \text{(empty)} \\ \text{I-PAT} & \begin{bmatrix} (z^{s}_{ay}, id), (z^{exp}_{v1,1}, id) \\ (z^{s}_{ay}, id), (z^{exp}_{v1,2}, id) \\ (z^{s}_{ai}, id), (z^{exp}_{v1,2}, id) \end{bmatrix} \end{bmatrix}$$

FIGURE 2.3 — Flexion du verbe *Balayer* décrite avec PARSLI

On utilise le trait `I-PAT` (pour inflexion-pattern) les règles de réalisation pour les deux couches. Ce trait est associé à un ensemble de sous-patterns, eux mêmes composés de couples de réalisation. Un couple de réalisation associe une zone de réalisation à une fonction de transfert. Dans notre exemple, cette fonction est la fonction identité (*id*).

Les deux stems alternatifs de l'entrée *balai*– (valide pour certaines personnes, comme "je balaie", "ils balaient", mais pas "nous balaieons") et *balay*– (pour toutes les personnes, "je balaye", "nous balayons") sont combinés avec les deux types d'exposants. Les deux ensembles d'exposants pour le présent de l'indicatif sont les suivants : $z^{exp}_{v1,1}$ contient toutes les terminaisons (soit *e*, *es*, *e*, *ent*) valides pour les deux stems, et $z^{exp}_{v1,2}$ celles qui ne le sont que pour *ay* (*ons*, *ez*). La combinaison des exposants de ce second ensemble avec z^{s}_{ai} est la seule interdite par l'entrée.

2.4 Conclusion

Dans ce chapitre, nous avons présenté un ensemble de théories morphologiques pour lesquelles il existe des langages de description informatiques. Les langages dédiés à chacune de ces théories demandent de pouvoir décrire des structures hétérogènes (des réseaux d'héritage, des automates, des structures de traits, etc), d'où la présence de multiples outils.

Dans le chapitre suivant, nous nous appliquerons à montrer que l'hétérogénéité des langages est également présente dans les théories syntaxiques, pour lesquelles la réflexion sur les langages de description est amorcée depuis longtemps. De plus, ce niveau de description est l'un de ceux, avec la morphologie et la sémantique, l'un de ceux utilisés pour la validation de l'approche que nous proposons.

Décrire la Syntaxe

3

Sommaire

3.1 Introduction **23**

3.2 Syntaxe générative par réécriture de chaînes **24**

3.2.1 Grammaires hors contexte 24

3.2.2 Grammaires Lexicales Fonctionnelles 29

3.2.3 Grammaire Syntagmatique Guidée par les Têtes 31

3.3 Syntaxe générative par réécriture d'arbres : la Grammaire d'Arbres Adjoints **35**

3.3.1 La Grammaire d'Arbres Adjoints 35

3.3.2 Une approche basée sur les metarègles 38

3.3.3 Des approches métagrammaticales 39

3.3.4 eXtensible MetaGrammar 41

3.4 Syntaxe basée sur la théorie des modèles **50**

3.4.1 eXtensible Dependency Grammar 50

3.4.2 Grammaire de Propriétés 54

3.5 Conclusion **55**

Après avoir noté la diversité des langages de description morphologiques, intéressons nous à la description de la syntaxe. Dans ce chapitre, nous présentons un ensemble de théories ayant été proposées pour cette tâche, parmi les plus utilisées. Pour chacun des formalismes grammaticaux allant de pair avec ces théories, nous observerons un ou plusieurs langages informatiques dédiés à son implémentation, le but étant de cerner les besoins des DSL en terme d'expressivité.

3.1 Introduction

La description de la syntaxe a connu un tournant à la fin des années 50 avec les travaux de Chomsky [Chomsky, 1957], introduisant la syntaxe dite générative. Une telle description repose sur un système de réécriture permettant de générer l'ensemble des énoncés grammaticaux, et se prête particulièrement à une représentation informatique.

Avec la grammaire générative, Chomsky s'oppose aux théories structuralistes. Ces théories constituaient le courant le plus répandu à l'époque en Europe, où le structuralisme est né dans les travaux de Ferdinand de Saussure ([de Saussure, 1915]), ainsi qu'aux États-Unis, où Leonard

Bloomfield puis Zellig Harris contribuèrent au développement des théories distributionnalistes ([Bloomfield, 1933],[Harris, 1955]).

Les grammaires structurales constituent un inventaire de formes et de constructions grammaticales apparaissant dans un corpus. Le fait que cet inventaire soit une ressource finie est une limite évidente de ces approches, puisqu'une grammaire devrait idéalement pouvoir analyser tous les énoncés (soit un nombre infini) dans une langue. La grammaire générative et transformationnelle introduit l'idée d'une langue fonctionnant sur deux niveaux : un niveau de surface et un niveau profond. On passe de la structure profonde à la structure de surface par le biais de transformations.

L'un des principaux atouts de ces approches en comparaison avec les approches structurales est la possibilité de décrire, à partir d'une ressource finie, un nombre infini de constructions. Les différentes évolutions des grammaires génératives ont engendré en 1990 le programme minimaliste ([Chomsky, 1992],[Chomsky, 1995]).

Les grammaires d'unification étendent les grammaires génératives à l'aide de structures de traits. Elles sont essentiellement nées des critiques sur la place de la sémantique (pour l'expression de laquelle la structure profonde n'était pas adaptée, sinon en la rendant de plus en plus abstraite) dans les grammaires transformationnelles, et de leurs transformations. Ces dernières posaient notamment problème lors des implémentations de la théorie pour l'analyse syntaxique : alors que les transformations permettaient de dériver une forme de surface à partir d'une forme profonde, c'est l'opération inverse de cette dérivation qui était nécessaire (lors d'une analyse, on souhaite obtenir à partir d'une forme de surface la structure profonde).

Dans les sections suivantes, nous présentons un ensemble de théories linguistiques appartenant aux grammaires d'unification, pour lesquelles un grand nombre d'outils de description existent. Nous distinguons deux grandes familles de formalismes basés sur la réécriture : les théories basées sur la réécriture de chaînes (section 3.2), puis celles basées sur la réécriture d'arbres (section 3.3). Pour chacune de ces théories, nous analyserons les besoins en terme d'expressivité des langages dédiés, ainsi que les langages de description existants.

Nous terminerons en section 3.4 par la présentation de théories syntaxiques appartenant à un autre courant, celui de la syntaxe basée sur la théorie des modèles (Model Theoretic Syntax, [Pullum et Scholz, 2001]). Nous nous intéressons à ces approches nées plus récemment, parce qu'il est important que la solution que nous proposerons soit adaptable à ces théories également (qui disposent en outre de leurs propres langages de description).

3.2 Syntaxe générative par réécriture de chaînes

3.2.1 Grammaires hors contexte

Les grammaires hors contexte, que nous avons présentées dans la sous-section 1.2.1 sont un premier exemple de grammaire générative.

Une grammaire hors contexte seule tend à sur-générer, puisque les seules contraintes apparaissant dans les règles de notre exemple concernent les catégories syntaxiques. Pour gérer les phénomènes dépendant de critères morphosyntaxiques, on peut utiliser des grammaires hors contexte augmentées d'information morphosyntaxique (par exemple, les structures de traits évoquées dans la sous-section 1.2.2.

Des implémentations de ces grammaires hors contexte enrichies ont été proposées à une époque où les affirmations de Chomsky sur l'insuffisance des grammaires hors contexte [1] étaient critiquées, et pour certaines, démontrées fausses ([Gazdar et Pullum, 1982]). Nous nous intéresserons dans ce qui suit à deux de ces implémentations (les DCG et PATR II)

Grammaires à Clauses Définies

Les Grammaires à Clauses Définies (DCG,[Pereira et Warren, 1980]) sont une manière d'exprimer une grammaire en programmation logique. Les grammaires logiques ainsi manipulées diffèrent des grammaires de réécriture (dans ce cas, les grammaires hors contexte) par la manipulation de termes en lieu et place des atomes

Leur intérêt est d'alléger la notation des clauses définies normales prolog, en les dispensant d'arguments (précisément, les listes de différence).

Par exemple, définir la grammaire hors contexte proposée précédemment en prolog sans utiliser les DCG reviendrait à :

```
p(L1,L3)  :- gn(L1,L2), gv(L2,L3).
gn(L1,L3) :- det(L1,L2), n(L2,L3).
gv(L1,L3) :- v(L1,L2), gn(L2,L3).
det([le|L],L).
det([la|L],L).
n([chat|L],L).
n([souris|L],L).
v([mange|L],L).
```

où les deux arguments de tous les prédicats correspondent aux choses suivantes : le premier est ce qu'il reste à analyser avant l'application du prédicat (`[le,chat,mange,la,souris]` par exemple avant l'application de toute règle), le second est ce qu'il reste à analyser après son application (`[]` après l'application de *p* sur notre exemple précédent).

Lors de l'évaluation du prédicat `p(A,[])`, *A* est unifié avec toutes les phrases (représentées par des listes de mots) grammaticales selon la grammaire hors contexte, soit `[la,souris,mange,le,chat]`, `[le,chat,mange,la,souris]` ou encore `[le,souris,mange,la,souris]` .

En employant le sucre syntaxique offert par les DCG, ce code peut devenir :

```
p --> gn, gv.
gn --> det, n.
gv --> v, gn.
det --> [le].
det --> [la].
n --> [chat].
n --> [souris].
v --> [mange].
```

Ces règles permettent de reconnaître si une phrase est valide, ou de générer l'ensemble des énoncés (si aucune règle n'est récursive). Si l'on souhaite obtenir une analyse syntaxique de

1. Les phénomènes linguistiques étant censés demander une capacité générative forte qu'elles n'offrent pas.

la phrase (un arbre, encodé comme un terme prolog), l'idée est de passer en paramètre des prédicats les représentation que l'on souhaite qu'ils construisent.

```
p(p(GN,GV)) --> gn(GN), gv(GV).
gn(gn(DET,N)) --> det(DET), n(N).
gv(gv(V,GN)) --> v(V), gn(GN).
det(det(le)) --> [le].
det(det(la)) --> [la].
n(n(chat)) --> [chat].
n(n(souris)) --> [souris].
v(v(mange)) --> [mange].
```

Une évaluation du prédicat `p(A,[le,chat,mange,la,souris],[])` unifie *A* avec le terme `p(gn(det(le),n(chat)),gv(v(mange),gn(det(la),n(souris))))`, équivalent à l'arbre de la figure 1.2.

On peut également utiliser des arguments supplémentaires pour construire une représentation sémantique, pour ajouter des informations contextuelles, ou pour ajouter de l'information morphosyntaxique dans les règles, pour gérer les accords par exemple. La surgénération constatée précédemment causée par l'absence de contrainte sur l'accord entre le déterminant et le nom (avec les phrases comme "la chat mange le souris") peut être évitée avec ce procédé.

```
p --> gn, gv.
gn --> det(G), n(G).
gv --> v, gn.
det(m) --> [le].
det(f) --> [la].
n(m) --> [chat].
n(f) --> [souris].
v --> [mange].
```

Les DCG constituent un sucre syntaxique précieux, avec une syntaxe intuitive, pour définir des grammaires hors contexte. En revanche, elles n'apportent pas d'avantage d'expressivité. Dans la section suivante, nous allons voir un langage de description présentant une avancée notable : la possibilité de définir des abstractions.

PATR II

PATR II est un l'un des premiers langages proposés pour la représentation de l'information linguistique, datant d'une trentaine d'années. Shieber présente son outil comme "la plus simple des grammaires d'unification" ([Shieber, 1984]). Le langage permet de représenter des formalismes syntaxiques de type grammaire de réécriture augmentée de structures de traits, et est conçu en particulier pour les systèmes faiblement lexicalisés.

PATR II fournit deux mécanismes permettant de gérer la redondance lexicale : les macros et les règles lexicales.

Les macros (*templates*) sont des outils pour capturer la redondance. Les traits communs à un ensemble d'entrées lexicales peuvent être factorisés par ce moyen. L'exemple suivant illustre ce phénomène en donnant les entrées lexicales pour verbes transitifs *aimer* et *manger*.

```
\w aimer
\c  v
\f  <arg0 cat> = np
    <arg1 cat> = np

\w manger
\c  v
\f  <arg0 cat> = np
    <arg1 cat> = np
```

Où \w précède l'entrée lexicale telle qu'elle sera lue dans la phrase, \c la catégorie syntaxique de l'entrée, et \f un ensemble de traits (additionnels à la catégorie syntaxique) propres à l'entrée lexicale. Lorsque la partie gauche d'un trait n'est pas atomique (par exemple <arg0 cat>), elle représente un chemin dans la structure de traits. On exprime ici le fait que les deux entrées sont de catégorie verbale et qu'ils sous-catégorisent deux arguments (un sujet et un objet) de catégorie nominale. La structure de traits associée à ces entrées est pour résumer :

$$\begin{bmatrix} \text{CAT} & \text{V} \\ \text{ARG0} & \begin{bmatrix} \text{CAT} & \text{np} \end{bmatrix} \\ \text{ARG1} & \begin{bmatrix} \text{CAT} & \text{np} \end{bmatrix} \end{bmatrix}$$

La redondance dans la sous-catégorisation, propre aux verbes transitifs, peut être factorisée par la macro suivante :

```
Let transitiveVerb be
  <arg0 cat> = np
  <arg1 cat> = np
```

puis appelée pour définir les deux entrées lexicales précédentes :

```
\w  aimer
\c  v
\f  transitiveVerb

\w  manger
\c  v
\f  transitiveVerb
```

Les règles lexicales permettent quant à elles de décrire les relations entre les entrées lexicales d'un même mot. L'exemple suivant, toujours tiré de [Crabbé, 2005] constitue la règle lexicale du mode passif.

```
Define passive as
  <out cat> = <in cat>
  <out morph> = passif
  <out arg0 cat> = pp
  <out arg1 cat> = <in arg0 cat>
```

```
Rule P -> GN GV
Rule GV -> v GN
Rule GN -> det nc
```

FIGURE 3.1 — Exemple de règles PATR II

```
\w chat
\c n
\f <num> = sg
   <gen> = m

\w souris
\c n
\f <num> = pl
   <gen> = f

\w  mange
\f  <cat> = v
    transitiveVerb

\w le
\f <cat> det

\w la
\f <cat> det
```

FIGURE 3.2 — Exemple de lexique PATR II

Cette règle lexicale prend en entrée l'entrée lexicale *in* et crée la nouvelle entrée lexicale *out*. `<out cat> = <in cat>` signifie l'identité entre les catégories syntaxiques des deux entrées. `<out morph> = passive` associe la morphologie passive à l'entrée lexicale en sortie. `<out arg0 cat> = pp` indique que le sujet de l'entrée lexicale créée (`arg0`) est réalisé par un groupe prépositionnel. enfin, `<out arg1 cat> = <in arg0 cat>` exprime le fait que l'objet de l'entrée créée est le sujet de l'entrée initiale.

La description des règles syntaxique se fait quant à elle grâce à la syntaxe :

```
Rule L -> R1...Rn
```

où `L` est un non terminal, et `R1...Rn` des terminaux ou non terminaux. La syntaxe d'une règle est donc littéralement celle des règles hors contexte précédemment discutées. On peut réécrire la grammaire de la figure 1.1 en définissant les règles comme dans la figure 3.1 et les entrées lexicale comme le montre la figure 3.2.

Avec les macros et les règles lexicales, PATR II apporte des mécanismes d'abstraction utiles pour développer des grammaires de taille plus importante. De façon similaire aux langages de programmation, les moyens de factorisation et de réutilisation deviennent disponibles.

Avec des grammaires algébriques, il est possible de représenter une partie de la grammaire d'une langue naturelle, si l'ordre de ses mots est contraint et que la morphologie est relativement

pauvre. Cependant, comme l'ont montré Bresnan et al [Bresnan, 1982] dans leurs travaux sur le hollandais, ou encore Shieber [Shieber, 1985] dans ses travaux sur le suisse germanique, certains phénomènes linguistiques ne sont pas descriptibles par les grammaires algébriques. Cela tient en partie au fait que ce formalisme ne permet pas de représenter des relations entre des constituants non contigus de la phrase (en effet, une règle hors contexte définit des contraintes entre constituants limitrophes, noeuds père ou frères dans l'arbre de dérivation).

3.2.2 Grammaires Lexicales Fonctionnelles

Le formalisme LFG suppose une indépendance entre les fonctions syntaxiques et les fonctions grammaticales, et sépare donc ces deux aspects lors de ses analyses. On manipule lors de celles ci deux types de structures : la structure de constituants, ou c-structure, et la structure fonctionnelle, ou f-structure. La première citée correspond à un arbre syntaxique. La f-structure est quant à elle un graphe, représenté par une structure de traits.

L'exemple de la figure 3.3 présente les règles pour l'analyse de la phrase "John loves Mary".

La grammaire jouet se compose de 5 règles de réécriture, enrichies de décorations. Ces décorations permettent la construction de la f-structure par unification. À l'intérieur de ces décorations apparaissent les signes $\uparrow$ et $\downarrow$ qui représentent respectivement la f-structure du non terminal de gauche et celle du symbole auquel la décoration est associée.

Par exemple, quand la règle (1) est appliquée, le trait $SUBJ$ de la f-structure de S est unifié avec la f-structure de NP, et les f-structures de S et VP sont unifiées.

Lors de l'analyse de la phrase "John loves Mary", la f-structure calculée est celle associé au non-terminal S, f_1. La valeur de son trait $SUBJ$ est unifié avec la f-structure de NP (d'après (1)), appelée f_2, et f_1 devient par unification la f-structure de VP. Les valeurs des traits $PRED$, NUM et $PERS$ de cette structure viennent de l'application de la règle (3). f_1 étant la f-structure de VP, l'application de (2) l'associe également à V, et son trait OBJ prend pour valeur la f-structure du NP de la règle, soit f_3. Les valeurs des traits de cette dernière proviennent des décorations de la règle (4). Enfin, le prédicat $Pred$ et le trait $TENSE$ de f_1 sont fournis par la règle lexicale (5).

Le Xerox Language Environment (XLE, [Kaplan et Maxwell, 1996]) est un outil de développement pour les grammaires lexicales fonctionnelles (LFG, [Bresnan et Kaplan, 1982] [Bresnan, 2001]). Il est célèbre pour être utilisé dans le projet ParGram (Parallel Grammar Project) initié par Xerox, dont le but est de fournir des grammaires à forte couverture pour un ensemble de langues.

XLE permet la description d'une règle LFG de la manière suivante :

```
Category --> Category1: Schemata1;
             Category2: Schemata2;
             Etc.
```

Les règles (1) et (2) de l'exemple précédent sont donc traduites en :

```
S --> NP : (^ SUBJ)=! ;
      VP : ^ = !.
VP --> V : ^ = ! ;
      NP : (^ OBJ)=!.
```

où ^ correspond à $\uparrow$ et ! à $\downarrow$.

Grammaire jouet:

(1)	S	→	NP $(\uparrow SUBJ) = \downarrow$	VP $\uparrow = \downarrow$
(2)	VP	→	V $\uparrow = \downarrow$	NP $(\uparrow OBJ) = \downarrow$
(3)	NP	→	John $(\uparrow PRED) = ' JOHN'$ $(\uparrow NUM) = SG$ $(\uparrow PERS) = 3$	
(4)	NP	→	Mary $(\uparrow PRED) = ' MARY'$ $(\uparrow NUM) = SG$ $(\uparrow PERS) = 3$	
(5)	V	→	loves $(\uparrow PRED) = ' LOVE\langle(\uparrow SUBJ)(\uparrow OBJ)\rangle'$ $(\uparrow TENSE) = PRESENT$	

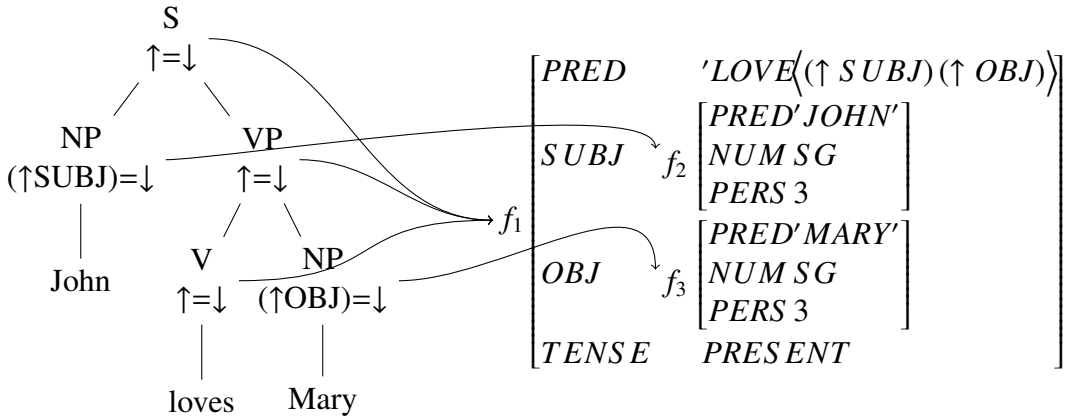


FIGURE 3.3 — Grammaire LFG, c-structure et f-structure pour la phrase "John loves Mary"

Les règles lexicales, soit celles dont la partie droite est un terminal (soit (3), (4) et (5) dans notre exemple), sont définies comme suit :

```
word Category1 Morphcode1 Schemata1;
     Category2 Morphcode2 Schemata2;
     Etc.
```

où Morphcode est un symbole indiquant si le mot utilise la morphologie XLE. Ici, nous utiliserons le symbole $*$, indiquant que la morphologie n'est pas utilisée. Les règles (3), (4) et (5) se traduisent en ce cas en :

```
John NP * (^ PRED)='JOHN' (^ NUM)=SG (^ PERS)=3.
Mary NP * (^ PRED)='MARY' (^ NUM)=SG (^ PERS)=3.
loves V * (^ PRED)='LOVE<(^ SUBJ) (^ OBJ)>'
          (^ TENSE)=PRESENT.
```

Les grammaires LFG apportent une expressivité plus forte (les langages qu'elles génèrent appartiennent à la classe des langages sensibles au contexte) pour la description des phénomènes linguistiques. En revanche, l'environnement offert par le formalisme est plutôt figé.

3.2.3 Grammaire Syntagmatique Guidée par les Têtes

Dans une grammaire syntagmatique guidée par les têtes [2] (HPSG, [Pollard et Sag, 1994]), le modèle de représentation de lexique utilisé est celui proposé par Dan Flickinger ([Flickinger *et al.*, 1985], [Flickinger, 1987]). Il repose sur une hiérarchie d'héritage, qui donne plus de liberté dans les généralisations que celle offerte par PATR II. Si dans ce dernier, seules les entrées lexicales initiales (celles qui n'ont pas été transformées par les règles lexicales) peuvent être factorisées, il est ici question de placer dans la hiérarchie d'héritage toutes les entrées lexicales.

L'exemple de la figure 3.4, extrait de [Flickinger, 1987] présente une hiérarchie d'héritage des noms.

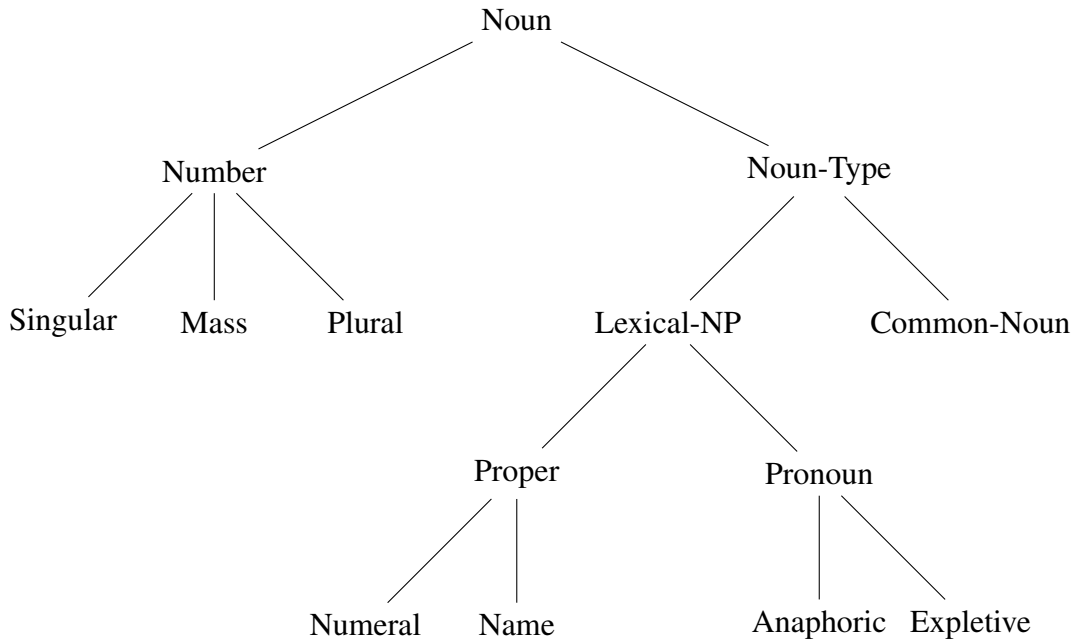


FIGURE 3.4 — Une hiérarchie d'héritage pour les noms

Cette hiérarchie signifie par exemple qu'une entrée lexicale de type numéral (associée à la classe Numeral) hérite de toutes les informations des classes Proper, Lexical-NP, Noun-Type et Noun. C'est donc le formalisme en lui même qui permet la factorisation, et non l'outil de description.

Dans une grammaire HPSG, tout les objets linguistiques manipulés sont des structures de traits, aussi bien pour le traitement de la syntaxe que pour la sémantique, le lexique et les phrases. Tout objet linguistique possède un type, et appartient donc à une hiérarchie, permettant l'héritage des traits.

2. Le classement des grammaires HPSG dans la catégorie des grammaires de réécriture peut être discuté. On peut en effet également considérer le formalisme comme basé sur la théorie des modèles, dans la mesure où l'unification permet de chercher un modèle satisfaisant en tenant compte des contraintes exprimées.

L'exemple simplifié (au niveau des notations, des traits utilisés) suivant, issu de [Abeillé, 1993], donne les règles HPSG pour deux mots et une règle de grammaire.

$$\begin{bmatrix} \textsc{Phon} & \backslash\text{jean}\backslash \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Categorie} & \text{N} \\ \textsc{Sous-cat} & <> \end{bmatrix} \\ \textsc{Semantique} & \text{'jean'} \end{bmatrix}$$

La description du mot "Jean" indique sa catégorie syntaxique (Nom), et l'absence de compléments (Sous-cat est une liste vide). On donne au mot pour sémantique l'individu 'jean', ainsi qu'une représentation phonétique (\jean\).

$$\begin{bmatrix} \textsc{Phon} & \backslash\text{dort}\backslash \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Categorie} & \text{V} \\ \textsc{Mode} & \text{ind} \\ \textsc{Sous-cat} & <[1]\begin{bmatrix} \textsc{Semantique} & [2] \end{bmatrix}> \end{bmatrix} \\ \textsc{Semantique} & \begin{bmatrix} \textsc{Relation} & \text{'dormir'} \\ \textsc{Agent} & [2] \end{bmatrix} \end{bmatrix}$$

Comme pour la description précédente, "dort" reçoit des traits concernant la syntaxe, précisant qu'il est un verbe à l'indicatif, et qu'il sous-catégorise un syntagme nominal (SN). Ce syntagme nominal est l'agent de la relation 'dormir', donnée comme sémantique, par le biais d'une coindexation.

$$\begin{bmatrix} \textsc{Phon} & \backslash 2 \oplus 1 \backslash \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Categorie} & \text{P} \\ \textsc{Sous-cat} & <> \end{bmatrix} \\ \textsc{Branches} & \begin{bmatrix} \textsc{Fils-Tête} & \begin{bmatrix} \textsc{Phon} & \backslash 1 \backslash \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Catégorie} & \text{V} \\ \textsc{Sous-cat} & <[1]> \end{bmatrix} \end{bmatrix} \\ \textsc{Sujet}\ [1] & \begin{bmatrix} \textsc{Phon} & \backslash 2 \backslash \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Catégorie} & \text{N} \\ \textsc{Sous-cat} & <> \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Cette règle décrit une phrase composée d'une construction sujet-verbe. La valeur phonétique de cette phrase est la concaténation (grâce à l'opérateur $\oplus$) des deux représentations phonétiques des composants. Les deux constructions précédentes ne contenaient que de l'information locale, mais celle-ci, par l'intermédiaire du trait `Branches`, contient des informations sur les fils syntaxiques du noeud en question. Le premier, considéré comme Fils-Tête de la structure, a des traits syntaxiques lui permettant d'unifier avec un verbe intransitif. Dans cet exemple simplifié,

le rôle que joue cette règle est celui qui est habituellement tenu par les schémas et les principes généraux de HPSG.

La combinaison des deux premières structures selon le schema de la troisième, on obtient la structure suivante par unification :

$$\begin{bmatrix} \textsc{Phon} & \text{\jean-dort\} \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Categorie} & \text{P} \\ \textsc{Sous-cat} & <> \end{bmatrix} \\ \textsc{Branches} & \begin{bmatrix} \textsc{Tête} & \begin{bmatrix} \textsc{Phon} & \text{\dort\} \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Catégorie} & \text{V} \\ \textsc{Sous-cat} & <[1]> \end{bmatrix} \\ \textsc{Sémantique} & \begin{bmatrix} \textsc{Relation} & \text{'dormir'} \\ \textsc{Agent} & [2] \end{bmatrix} \end{bmatrix} \\ \textsc{Sujet}\ [1] & \begin{bmatrix} \textsc{Phon} & \text{\jean\} \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Catégorie} & \text{N} \\ \textsc{Sous-cat} & <> \end{bmatrix} \\ \textsc{Sémantique} & \text{'jean'}[2] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Cette structure de traits est équivalente à une structure arborescente dont la racine est associée à la structure de traits de la phrase, et ses fils aux structures de traits des mots. Le trait `Branches` contient les étiquettes des branches et les traits de ces noeuds.

$$\begin{bmatrix} \textsc{Phon} & \text{\jean-dort\} \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Categorie} & \text{P} \\ \textsc{Sous-cat} & <> \end{bmatrix} \end{bmatrix}$$

$$[1]\begin{bmatrix} \textsc{Phon} & \text{\jean\} \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Categorie} & \text{N} \\ \textsc{Sous-cat} & <> \end{bmatrix} \\ \textsc{Sem} & \text{'jean'} \end{bmatrix} \qquad \begin{bmatrix} \textsc{Phon} & \text{\dort\} \\ \textsc{Syntaxe} & \begin{bmatrix} \textsc{Categorie} & \text{V} \\ \textsc{Sous-cat} & <[1]> \end{bmatrix} \\ \textsc{Sem} & \begin{bmatrix} \textsc{Relation} & \text{'dormir'} \\ \textsc{Agent} & \text{'jean'} \end{bmatrix} \end{bmatrix}$$

Le LKB (Linguistic Knowledge Builder, [Copestake, 2002]) est un environnement de développement pour les formalismes grammaticaux basés sur l'unification, particulièrement adapté aux grammaires HPSG. Dans le LKB, la définition d'une grammaire HPSG repose sur les deux aspects que nous avons présenté : la définition des types (et des hiérarchies de types) et la définition des règles, lexicales et grammaticales (soit des structures de traits).

Les types se définissent dans un fichier dédié de la manière suivante :

$$type := typeHerite_1 \ \& \ \ldots \& \ typeHerite_n \ \& \\ [att_1 \ type_1 \ \ldots \ att_n \ type_n]$$

où *type* est le type à définir, $typeHerite_1 \ldots typeHerite_n$ les types dont il hérite, $att_1 \ldots att_n$ des attributs et $type_1 \ldots type_n$ les types valides pour ces attributs (éventuellement eux même des structures de traits).

Le type le plus général dans une hiérarchie est noté $*top*$.

Les règles ont une syntaxe similaire, elles consistent à donner un type à la structure de traits, puis ses traits :

$$regle := type \ \& \\ [att_1 \ type_1 \ \ldots \ att_n \ type_n]$$

La règle précédemment proposée pour les constructions sujet-verbe peut être décrite de la façon suivante :

```
sujet-verbe := phrase &
  [
    Phon = #2 + #1
    Syntaxe = [  Categorie = P ]
    Branches= [
      Tete= [
        Phon = #1
        Syntaxe = [
          Categorie & V
          Sous-cat & <SN>
          ]
      ]
      Sujet = [
        Phon = #2
        Syntaxe = [
          Categorie = N
          Sous-cat = <>
        ]
      ]
    ]
  ]
```

où #1 et #2 sont des variables d'unification.

Le formalisme HPSG permet une description plus homogène que celle proposée par LFG : l'utilisation des structures de traits pour décrire à la fois la syntaxe, la sémantique, et le lexique des langues rend l'intégration des différents niveaux de description linguistique au formalisme plus explicite.

Le LinGO Grammar Matrix ([Bender *et al.*, 2002]) permet de créer et configurer rapidement un squelette de grammaire HPSG dont on choisit les propriétés. Ce squelette est compatible avec LKB. L'utilisateur final n'a en revanche pas de contrôle sur la gestion des patrons utilisés pour produire le squelette de grammaire.

Bien entendu, la présentation des approches donnée dans cette section n'est pas exhaustive. Parmi les formalismes les plus populaires, on peut notamment citer la Grammaire Catégorielle Combinatoire (CCG, [Steedman, 2000]).

3.3 Syntaxe générative par réécriture d'arbres : la Grammaire d'Arbres Adjoints

Dans cette section, nous nous intéressons à un formalisme représentant un compromis intéressant entre l'expressivité et la complexité (puisque légèrement sensible au contexte, en étant analysable en temps polynomial) : les Grammaires d'Arbres Adjoints. Comme son nom l'indique, il ne se base pas sur la réécriture de chaînes, mais d'arbres. Pour ce formalisme, un nombre conséquent d'approches de description et de langages associés ont été proposés.

3.3.1 La Grammaire d'Arbres Adjoints

La Grammaire d'Arbres Adjoints ([Joshi et Schabes, 1997], Tree Adjoining Grammar, ou TAG) est un formalisme dont l'un des attraits principaux est de pouvoir représenter des relations entre des composants possiblement très éloignés dans la phrase. Une grammaire TAG se compose d'un ensemble d'arbres, appelés arbres élémentaires. Ces derniers se composent de la manière suivante : Chaque feuille est étiquetée soit par un terminal (mot du lexique), soit par un non terminal (catégorie syntaxique). Les noeuds non feuilles sont quant à eux étiquetés par des non terminaux (catégories syntaxiques).

Lors d'une analyse syntaxique, les arbres élémentaires sont combinés lors d'une procédure appelée dérivation. Une dérivation est l'application d'une suite de règles de réécriture d'arbre. Le formalisme TAG comprend deux opérations : la substitution et l'adjonction. C'est ici qu'interviennent les non terminaux qui étiquettent certaines feuilles, puisqu'ils portent des symboles qui les rangent soit dans la catégorie des noeuds à substituer (si l'étiquette contient une flèche ↓), soit dans celle des noeuds pieds (si l'étiquette contient une étoile $*$, voir exemple ci-dessous). Un arbre initial ne contient pas de noeud pied, alors qu'on en trouve exactement un dans un arbre auxiliaire. Un noeud à substituer doit être remplacé par un arbre non auxiliaire dont la racine correspond, comme illustré dans la figure 3.5.

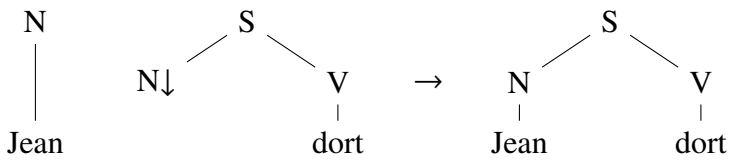


FIGURE 3.5 — La substitution en TAG

Par l'opération d'adjonction, un arbre auxilliaire peut être inséré à la place d'un noeud non feuille. Il faut pour cela que le noeud site d'adjonction (celui qui reçoit l'adjonction), la racine

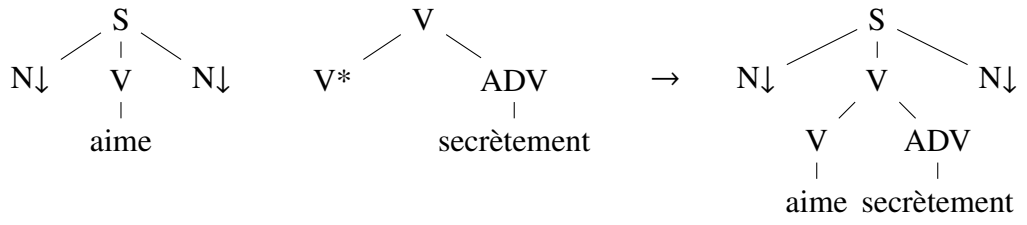


FIGURE 3.6 — L'adjonction en TAG

de l'arbre adjoint et son noeud pied portent la même étiquette. Le sous arbre ayant pour racine le site d'adjonction est rattaché au pied de l'arbre adjoint. Cette opération est illustrée dans la figure 3.6.

Une grammaire TAG à forte couverture peut être composée de dizaines de milliers d'arbres élémentaires. La création d'une telle grammaire est en conséquence une tâche très longue, et sa maintenance est laborieuse. La grammaire XTAG de l'anglais [XTAG Research Group, 2001] contient par exemple un millier de modèles d'arbres. La grammaire du français FTAG en contient plus de 5000 [Abeillé, 1991].

On remarque en outre que les grammaires TAG présentent une forte redondance structurelle. Pour illustrer ceci, considérons la figure 3.7, qui montre un ensemble de règles issus d'une même famille LTAG (TAG lexicalisé) correspondant aux verbes transitifs. Dans les grammaire LTAG, chaque arbre contient au moins un noeud feuille étiqueté par un terminal. Ici, nous utilisons le symbole ⋄ pour permettre l'insertion des éléments du lexique durant l'analyse (cette insertion étant contrainte par l'unification entre les traits du noeud et ceux de l'élément du lexique).

On peut remarquer une importante redondance à l'intérieur des règles, chacun des arbres partageant une partie de sa structure avec d'autres. Un exemple de cette redondance est illustré sur les deux règles de la figure 3.8.

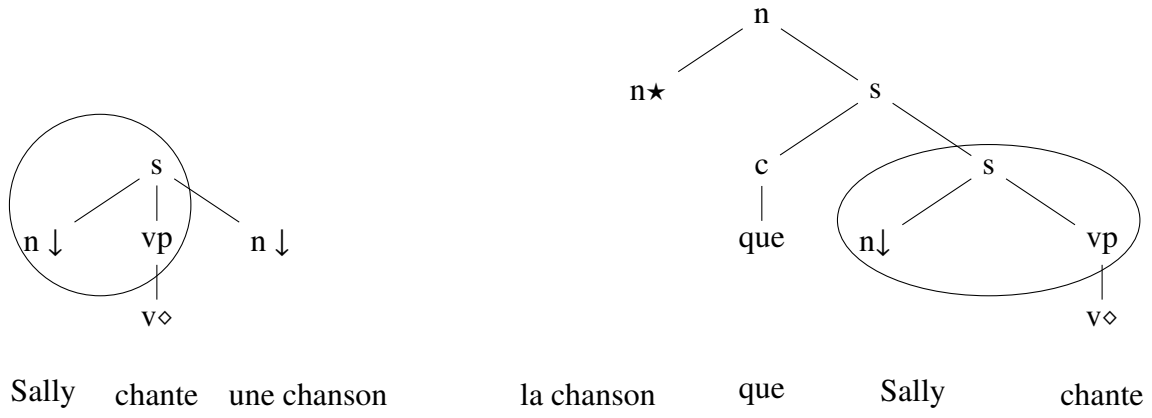


FIGURE 3.8 — Verbe avec subject canonique et objet canonique ou extrait

La structure des arbres n'est pas la seule chose partagée par les arbres. Les traits morphosyntaxiques, utilisés par exemple pour réaliser les accords en genre ou en nombre, sont également redondants.

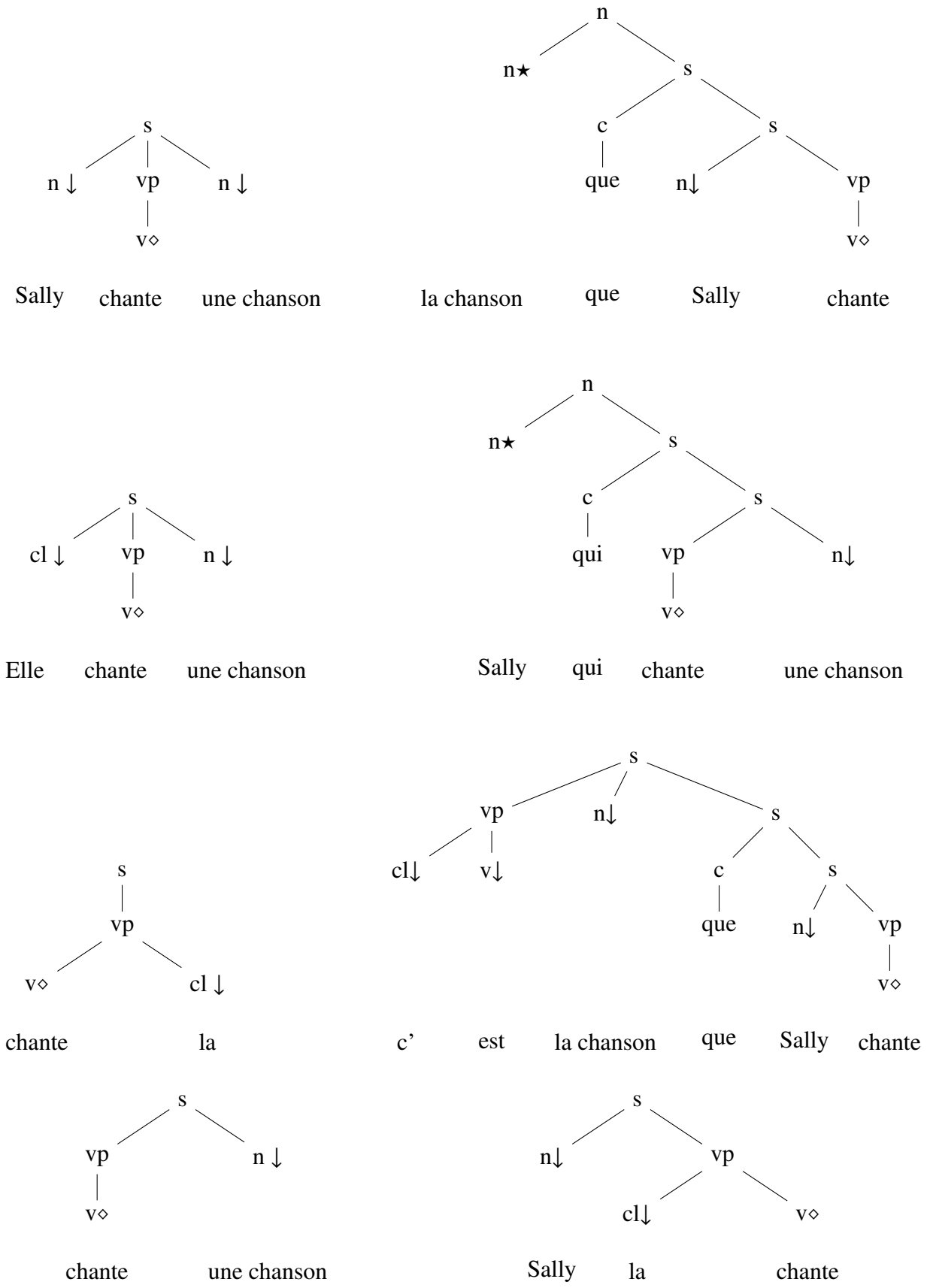


FIGURE 3.7 — Une partie de la famille des verbes transitifs en TAG

Plusieurs approches ont été proposées pour aider les linguistes dans cette tâche. On peut découper ces approches en deux familles : la génération automatique et la génération semi-automatique. Dans le premier cas, les règles de la grammaire sont apprises d'après un corpus. Les approches automatiques sont aujourd'hui des techniques très utilisées, mais nécessitent un corpus difficile à créer si l'on souhaite qu'il couvre une partie importante d'une langue. La génération semi-automatique génère l'ensemble de la grammaire à partir d'un autre type de ressource, une description de cette grammaire.

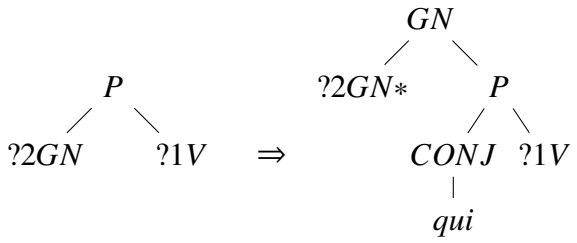
Dans la suite de cette section, nous présentons un ensemble de méthodes pour la génération semi-automatique de grammaires d'arbres adjoints.

3.3.2 Une approche basée sur les metarègles

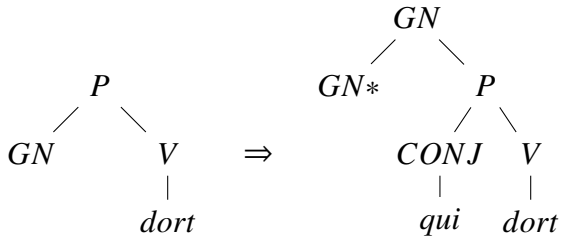
Dans le cadre du projet XTAG, Carlos Prolo [Prolo, 2002] a proposé un processus de développement de grammaires basé sur les règles de transformation. Le travail consiste à fournir un ensemble de règles initiales, appelées canoniques, puis à leur appliquer des transformations pour obtenir de nouvelles règles. Les règles qui guident ces transformations sont donc des règles qui agissent sur des règles. C'est pourquoi on les appelle des métarègles (concept introduit dans [Becker, 1993], [Becker, 2000]).

Une metarègle est de la forme *gauche* → *droite*. Le principe est le suivant : l'arbre d'entrée doit correspondre à la partie gauche de la metarègle, à partir de la racine. On génère ensuite le nouvel arbre, généralement plus complexe, en remplaçant les variables par leur valeur correspondante dans l'arbre d'entrée. La métarègle fonctionne donc comme un outil de filtrage par motif avec des arbres, la partie gauche de la règle correspondant au motif de filtrage pour sélectionner les arbres.

Par exemple, la métarègle suivante permet l'extraction du sujet.



Cette règle appliquée à l'utilisation canonique de "dort" produit l'entrée suivante :



La partie gauche de la métarègle attend un GN (la variable 2, symbolysée par ?2), et un verbe. Appliquée à notre règle, on obtient donc l'arbre de la partie droite de la métarègle, où la variable 2 a été remplacée par le groupe nominal de la règle, et le verbe par celui de la règle, en l'occurence "dort".

Cette approche a toujours un certain nombre de défauts. En effet, on se trouve face à un système de réécriture, et un ensemble de règles. Dans un tel système, il est difficile de vérifier si la tâche de génération des nouveaux arbres TAG va terminer, boucler sur une suite de règles, ou encore si l'application de certaines règles n'annule pas l'effet des précédentes. Un autre aspect négatif est l'écriture manuelle des règles initiales, et le fait de décider quelles règles feront partie de cet ensemble.

Prolo remédie à ce problème en ordonnant ses 21 métarègles et en les appliquant les unes après les autres, une seule fois. Initialement, la règle est donc appliquée à l'ensemble des règles canoniques écrites à la main, puis cet ensemble augmente avec les nouvelles règles générées.

Les résultats sont plutôt encourageants, puisqu'il a été possible de générer 783 arbres correspondant aux 53 familles de verbes (l'ensemble initial), ceci avec ces 21 métarègles [Prolo, 2002]. En revanche, un certain nombre d'arbres n'a pas pu être généré, ou a été généré alors qu'il représente des règles incorrectes, ce qui illustre le manque de contrôle sur les grammaires générées.

3.3.3 Des approches métagrammaticales

Les approches métagrammaticales privilégient une approche descriptive, basée sur des critères linguistiques, en opposition à la précédente, plus procédurale.

La métagrammaire de Candito

Candito [Candito, 1996] a été parmi les premières à s'intéresser à ce concept de métagrammaire. Elle étend les formalismes de type PATR en proposant de baser les hiérarchies d'abstraction sur un système à 3 dimensions.

Dans la dimension 1, on classe les fragments qui concernent la sous-catégorisation initiale. La dimension 2 contient les séquences possibles de redistributions. Enfin, la dimension 3 correspond aux réalisations des fonctions grammaticales.

Chaque classe de la hiérarchie est une description partielle d'arbre. On génère un arbre élémentaire par combinaison d'exactement une classe terminale de dimension 1, une classe terminale de dimension 2, et n classes terminales de dimension 3, n étant le nombre d'arguments du prédicat associé à l'arbre généré.

Ce principe de dimensions, mais surtout le manque de liberté qu'elles provoquent dans la méthodologie de développement, sont les principaux points négatif de l'approche de Candito. Il oblige la personne qui conçoit la métagrammaire à comprendre non seulement le système de dimension, mais également le fonctionnement du compilateur. De plus, l'algorithme de croisement tentant tous les croisements possibles entre les classes terminales, il est très compliqué d'assurer un contrôle permettant d'éviter les combinaisons non souhaitées.

Xia [Xia, 2001] a proposé, sensiblement à la même époque, une approche assez similaire (combinaison de classes). La différence principale réside dans le fait que toutes les variables utilisées sont locales aux descriptions partielles, alors qu'elles sont globales chez Candito.

La métagrammaire de Gaiffe

Dans son approche, Gaiffe tente de répondre à la question suivante : puisque le système de dimensions limitées semble être un aspect à abandonner, pourrait on décider d'une autre méthode

pour guider les combinaisons de fragment ?

Contrairement à Candito, Gaiffe [Gaiffe *et al.*, 2002] base son approche sur un nombre arbitraire de dimensions. On peut renseigner pour chaque classe de la hiérarchie ses besoins et ses ressources (concrètement, des structures de traits dont les traits prennent les valeurs + ou -). Les combinaisons de fragments d'arbres se font donc en prenant soin que chaque besoin soit compensé par la ressource adéquate. Les limites rencontrées ici sont sensiblement les mêmes que chez Candito (à savoir un manque de contrôle des combinaisons de fragments d'arbres), même si la notion de besoins et de resources généralise son approche. Les notions de besoins et de resources, si elle semblent aisées à définir pour des règles élémentaire, peuvent rapidement devenir très complexes (comment prévoir toutes les satisfactions possibles), et le fait de n'avoir que des noms de noeuds à portée globale dans les descriptions d'arbres manipulées rend leur gestion rapidement difficile.

Le MGCOMP

Le MGCOMP ([Thomasset et Clergerie, 2005]) est un générateur d'arbres TAG factorisés (c'est à dire parfois sous-spécifiés) qui peuvent être utilisés par le système Dyalog [de La Clergerie, 2005] pour la construction d'analyseurs syntaxiques.

Comme pour Gaiffe, une meta-grammaire est composée d'un ensemble de classes, chacune contenant un ensemble de contraintes (typiquement des descriptions d'arbres) et d'éventuelles ressources, soit fournies par la classe (`+r`) ou demandées par la classe (`-r`). L'exemple de la

```
class collect_real_subject_canonical {
  <: collect_real_subject ;
   $arg.extracted = value(~cleft) ;
   S >> VSubj; VSubj < V; V >> postsubj; VMod < postsubj;
   node postsubj : [cat:N2, id:subject,
                    type:subst, top:[wh:-,sat:+]];
   - postsubj::agreement; postsubj = postsubj::N;
   postsubj =>
     node(Infl).bot.inv = value(+),
     $arg.extracted = value(-), $arg.real = value(N2),
     desc.extraction = value(~-),
     node(V).top.mode=value(~infinitive|imperative
                            |gerundive|participle);
   ~ postsubj => node(Infl).bot.inv = value(~+)
}
```

Figure 3.9 — Exemple de classe MGCOMP (issu de [Thomasset et Clergerie, 2005])

figure 3.9 définit la classe `collect_real_subject_canonical`, héritant de `collect_real_subject` (héritage exprimé par l'opérateur `<:`), décrivant toutes les réalisations du sujet. `collect_real_subject_canonical` la complète pour le cas particulier du sujet canonique. `=`, `<`, `>>` et `>>+` sont respectivement les opérateurs posant des contraintes d'égalité, de précédence, de dominance immédiate et indirecte entre noeuds. La contrainte

```
node postsubj : [cat:N2, id:subject,
                 type:subst, top:[wh:-,sat:+]]
```

concerne quant à elle les traits attachés au noeud `postsubj`. `$arg` est une variable, `desc` fait référence à la classe courante, | et ~ sont respectivement les opérateurs de disjonction et de négation. **=>** indique une condition sur l'existence (ou la non existence, comme pour la dernière contrainte de l'exemple) d'un noeud.

La gestion de l'espace de nom ajouté à la notion de polarités rendent la description plus simple à développer et à maintenir. Le principe de la compilation est de combiner des classes, en accumulant leurs contraintes, jusqu'à arriver à neutraliser toutes les polarités.

On retrouve un grand nombre de concepts intéressants : celui des espaces de nom, l'accumulation de contraintes (ici résolues à l'analyse syntaxique) toujours au sein d'une hiérarchie d'héritage.

Dans la section qui suit, nous présenterons l'approche sur laquelle le travail de cette thèse repose : eXtensible MetaGrammar (XMG). XMG partage certaines caractéristiques avec les langage MGCOMP tout en visant une plus grande modularité.

3.3.4 eXtensible MetaGrammar

Dans l'approche XMG ([Crabbé *et al.*, 2013]), on se place dans la lignée des approches précédentes à base de combinaison de fragments d'arbres, la différence étant que les fragments sont assemblés de façon totalement déclarative.

Par exemple, la règle de la figure 3.10 peut être considérée comme l'assemblage de trois fragments de règles, l'un représentant le sujet canonique, un autre représentant l'objet relatif, et un dernier représentant la morphologie verbale. Ces fragments sont présentés dans la figure 3.11.

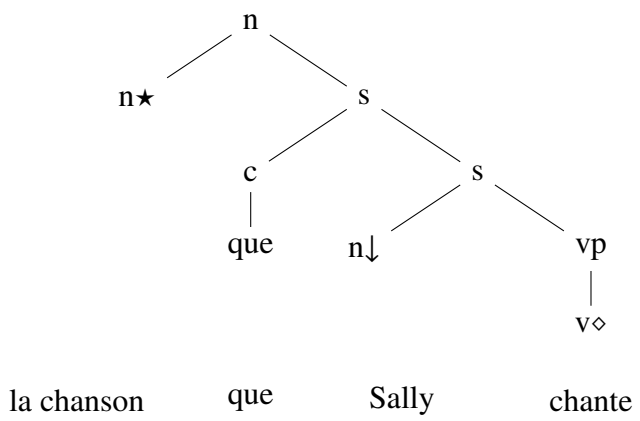


FIGURE 3.10 — Une construction active avec sujet canonique et objet relatif

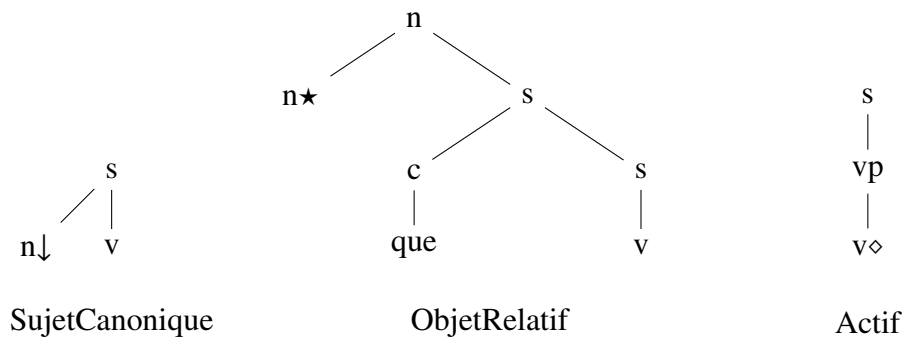


FIGURE 3.11 — Fragments d'arbres TAG pour un sujet canonique, un objet relatif et la morphologie verbale active

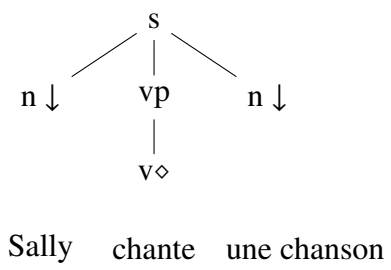
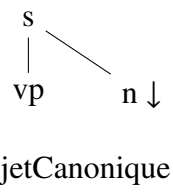


FIGURE 3.12 — Une construction active avec sujet canonique et objet canonique

La règle de la figure 3.12 peut quant à elle être vue comme la combinaison d'un fragment pour un autre type d'objet, canonique, avec les fragments du sujet canonique et de la morphologie active. Le fragment correspondant à un objet canonique est le suivant :



Pour générer les arbres de la grammaire, on donne les règles pour combiner ces fragments. Ces règles sont basées sur des opérateurs logiques. Par exemple, on peut exprimer le fait qu'un objet peut être canonique ou relatif en définissant l'abstraction suivante :

$$\text{Objet} \rightarrow \text{ObjetCanonique} \vee \text{ObjetRelatif}$$

où $\vee$ représente la disjonction. L'abstraction Transitif qui suit (dans laquelle $\wedge$ exprime la conjonction) peut ainsi décrire les deux règles présentées précédemment.

$$\text{Transitif} \rightarrow \text{SujetCanonique} \wedge \text{Actif} \wedge \text{Objet}$$

Le fractionnement des règles est non seulement un moyen de diminuer la taille de la ressource et de faciliter sa maintenance, mais elle présente également un intérêt linguistique. Elle fait apparaître à travers ses abstractions des phénomènes linguistiques plus fins et spécifiques que ceux apparaissant dans la grammaire cible.

XMG est une approche metagrammaticale fondée sur les concepts de la programmation logique. Le nom XMG désigne à la fois un langage metagrammatical, et le compilateur pour ce langage. La fonction de ce compilateur est de générer une grammaire à partir d'une description compacte de celle-ci. L'un de ses principaux atouts est son système de dimensions, permettant de manipuler indépendamment plusieurs niveaux de description linguistique.

Classes

Une métagrammaire XMG est composée d'un ensemble d'abstractions, appelées classes, représentant des fragments de la grammaire. La notion de classe permet de gérer la redondance (les classes représentant les fragments redondants étant utilisées plusieurs fois) et la portée des variables utilisées dans la description.

Une classe possède un identifiant et d'éventuels paramètres, un ensemble d'identifiants de classes importées, et deux ensembles d'identifiants correspondants aux variables déclarées et exportées. Une variable a une portée locale à la classe c lorsqu'elle est déclarée dans celle-ci. En cas d'export, cette portée peut être étendue à toute classe important la classe c.

Langage de contrôle

Comme nous l'avons vu précédemment, un concept central d'XMG est de combiner des fragments ou des abstractions. Le langage proposé pour ceci (appelé langage de contrôle) est essentiellement composé des opérateurs logiques de conjonction et de disjonction.

$$\begin{array}{rcl} \mathit{Classe} & := & \mathit{Nom}[p_1, \ldots, p_n] \rightarrow \mathit{Contenu} \\ \mathit{Contenu} & := & \langle \mathit{Dim} \rangle \{\mathit{Desc}\} \mid \mathit{Nom}[\ldots] \mid \mathit{Contenu} \vee \mathit{Contenu} \\ & & \mid \mathit{Contenu} \wedge \mathit{Contenu} \end{array}$$

L'exemple suivant illustre la combinaison conjonctive et disjonctive de fragments d'arbres.

$$\begin{array}{rl} \mathit{Objet} & \rightarrow \mathit{ObjCan} \vee \mathit{ObjRel} \\ \mathit{Transitif} & \rightarrow \mathit{SujCan} \wedge \mathit{Actif} \wedge \mathit{Objet} \end{array}$$

La classe *Objet* correspond soit à la description d'un objet Canonique (*ObjCan*) soit à celle d'un objet relatif (*ObjRel*). La classe *Transitif* est quant à elle l'association de la description d'un sujet canonique (*SujCan*), de celle de la forme active (*Actif*) et de l'une des deux formes de l'objet décrits par la classe *Objet*.

Langages de description et dimensions

Nous avons jusqu'ici vu comment combiner des abstractions, c'est à dire comment obtenir à partir de fragments grammaticaux élémentaires des règles plus complexes. Il reste à expliquer la création de ces fragments grammaticaux élémentaires.

XMG intègre une notion de dimensions, qui permettent de séparer les différents niveaux de description linguistique (par exemple, la syntaxe et la sémantique). Une dimension fournit un

accumulateur dédié aux structures d'un niveau de description, ainsi qu'un langage de description pour ces structures.

Pour les descriptions d'arbres, le langage proposé est basé sur [Rogers et Vijay-Shanker, 1994] et peut s'exprimer de la façon suivante :

$$\begin{aligned} Desc \ := x \rightarrow y \ \mid \ x \rightarrow^{+} y \ \mid \ x \rightarrow^{*} y \ \mid \ x \prec y \ \mid \ x \prec^{+} y \ \mid \ x \prec^{*} y \ \mid \ x[f{:}E] \\ \mid \ x(p{:}E) \mid \ Desc \ \wedge \ Desc \end{aligned}$$

où x et y sont des variables de noeuds, f l'attribut d'un trait dans une structure de traits, p une propriété, E une expression. L'opérateur $\rightarrow$ est la relation de dominance immédiate entre noeuds, $\rightarrow^{+}$ et $\rightarrow^{*}$ sont respectivement sa clôture transitive (dominance stricte) et réflexive transitive. L'opérateur $\prec$ est la relation de précédence immédiate, $\prec^{+}$ et $\prec^{*}$ respectivement sa clôture transitive et sa clôture réflexive transitive.

Un langage permet également la description de formules logiques pour une dimension sémantique. La dimension étant basée sur la sémantique "plate" (flat semantic, [Gardent et Kallmeyer, 2003]), les prédicats décrits sont étiquetés, et des contraintes sur la portée de ces étiquettes sont définies, dans le but de gérer les ambiguïtés. Il s'exprime de la manière suivante :

$$Desc \ := l : p(E_1, \ldots, E_n) \mid \neg l : p(E_1, \ldots, E_n) \mid E_i >> E_j \mid E$$

où $E_1, \ldots, E_n$, E_i, E_j, et E sont des identifiants sémantiques, l une étiquette, p un identifiant de prédicat, $\neg$ est la négation et $>>$ la relation de supériorité de portée (scope over).

Solveurs

Une fois la métagrammaire compilée en un programme logique, le code généré est exécuté. Cette exécution, non déterministe, produit par backtracking un ensemble de résultats. Chacun de ces résultats est une description correspondant à un ensemble (éventuellement vide) de structures à générer. Les deux dimensions proposées par XMG ont un comportement différent à cette étape. Pour les Grammaires d'Interaction ([Perrier, 2000]), un formalisme grammatical supporté par XMG et pour lequel une grammaire est constitué d'arbres sous spécifiés, l'accumulation des noeuds et des relations compose directement la grammaire. C'est le parser [3] qui construit les arbres pendant l'analyse. Pour les grammaires TAG en revanche, la ressource à produire est un ensemble d'arbres. Ces arbres doivent donc être construits à partir de l'accumulation.

Pour chaque description d'arbre, XMG génère l'ensemble des modèles minimaux (soit les modèles ou aucun noeud absent de la description n'est créé). Le solveur s'inspire des contraintes formulées dans [Duchier et Niehren, 2000].

Principes

La résolution des modèles telle que nous l'avons vue consiste en des règles de bonne formation uniquement appliquées aux structures. Il est difficile lors de l'élaboration de la grammaire d'anticiper toutes les combinaisons possibles des fragments définis. Pour éviter la sur-génération, il est nécessaire de poser un certain nombre de contraintes sur les modèles. Ces contraintes doivent être choisies de manière cohérente avec le cadre de développement.

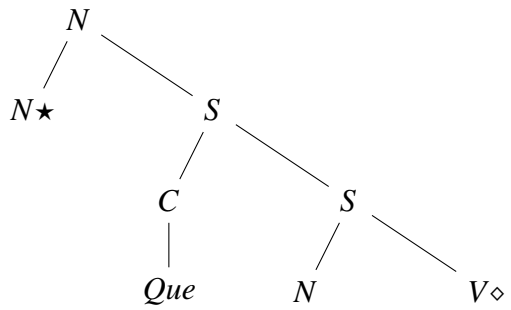
3. http://wikilligramme.loria.fr/doku.php?id=leopar:leopar

Les principes sont une manière d'incorporer de nouvelles démarches méthodologiques de développement au compilateur. Ils correspondent à un ensemble de contraintes qui s'appliquent aux modèles d'une description. Nous présenterons ici deux des principes proposés pour les grammaires arborescentes dans XMG : *colors* et *rank*.

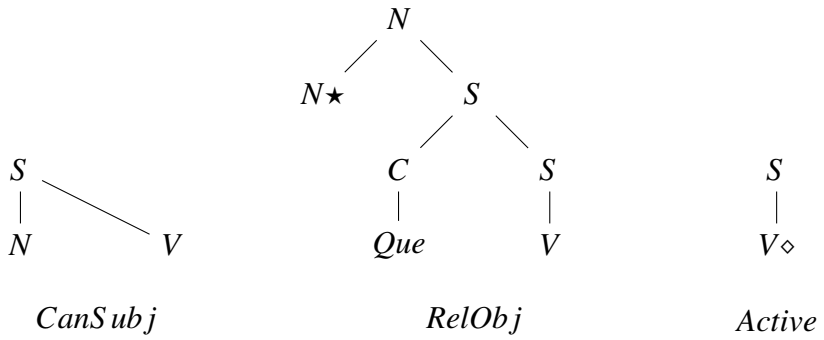
Gérer les combinaisons de fragments

Lorsque deux fragments sont combinés par conjonction, le nombre de fragments résultants dépend des unifications explicitement données dans la métagrammaire. En l'absence d'unification forcée, un noeud peut prendre n'importe quelle position dans le modèle pourvu qu'elle ne viole aucune des contraintes accumulées.

Considérons la règle suivante, applicable pour des phrases telles que "La pomme que Jean mange".



On propose le découpage suivant, isolant le sujet, l'objet, et le verbe, réutilisables dans de nombreuses autres règles.



La combinaison de ces fragments sans autre contraintes que les traits morphosyntaxiques des noeuds peut mener à différents modèles, dont ceux présentés dans la figure 3.13, qui ne sont pas linguistiquement valides. Dans le premier cas, le fragment du verbe se place sous un noeud étiqueté *s*, mais pas celui attendu. Dans le second, c'est le fragment du sujet qui est mal placé. Enfin, dans le troisième cas, les fragments sont bien combinés, mais l'unification des noeuds étiquetés *v* n'est pas effectuée.

Pourtant, l'intuition linguistique derrière les fragments est que le verbe est contribué par la morphologie active, alors que les contributions du sujet et de l'objet nécessitent la présence d'un verbe. Réaliser les unifications à la main, explicitement (avec le symbole '=') ou grâce à des variables partagées, est contraignant.

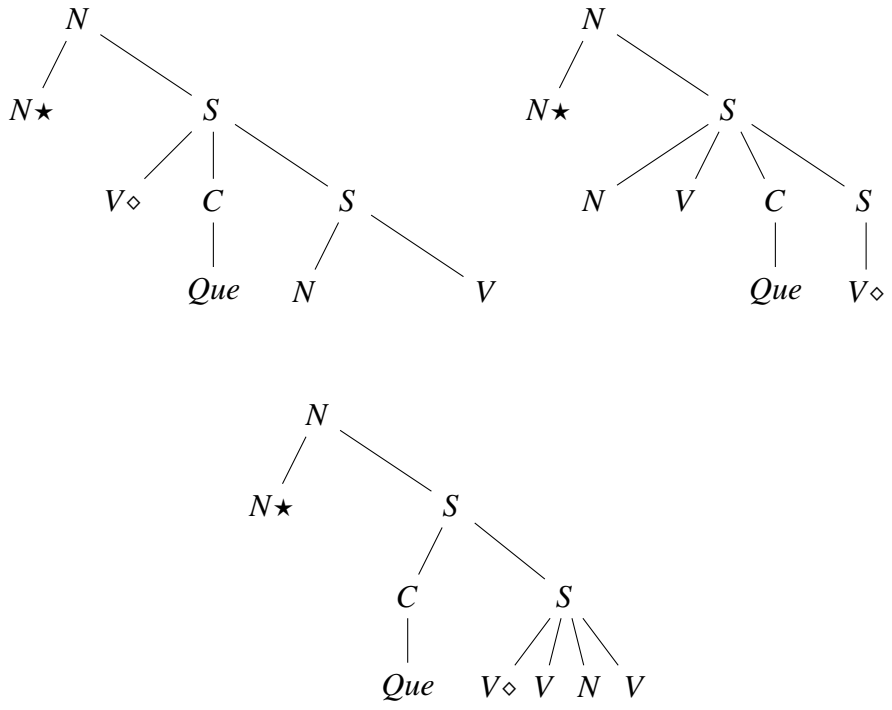


FIGURE 3.13 — Combinaisons sans contraintes

Le principe de couleurs, implémentant l'idée de [Crabbé et Duchier, 2004], permet d'associer à chaque noeud des fragments une polarité, qui ajoute des contraintes sur l'unification du noeud. Les couleurs expriment les notions de besoin et de ressource : dans un modèle, chaque noeud blanc (un besoin) doit être unifié avec un noeud noir (une ressource), les noeuds rouges correspondant quant à eux à des noeuds saturés. Les modèles valides ne contiennent que des noeuds de couleur noire et rouge. Les règles d'unification de noeuds sont résumées par le tableau suivant.

	$\bullet_B$	$\bullet_R$	$\circ_W$	$\perp$
$\bullet_B$	$\perp$	$\perp$	$\bullet_B$	$\perp$
$\bullet_R$	$\perp$	$\perp$	$\perp$	$\perp$
$\circ_W$	$\bullet_B$	$\perp$	$\circ_W$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

FIGURE 3.14 — Règles d'unification pour les couleurs.

Nous modifions donc les fragments précédents en colorant les noeuds. Le fragment *Active* contient uniquement des noeuds noirs, qui seront utilisés pour saturer les noeuds blancs des fragments du sujet et de l'objet.

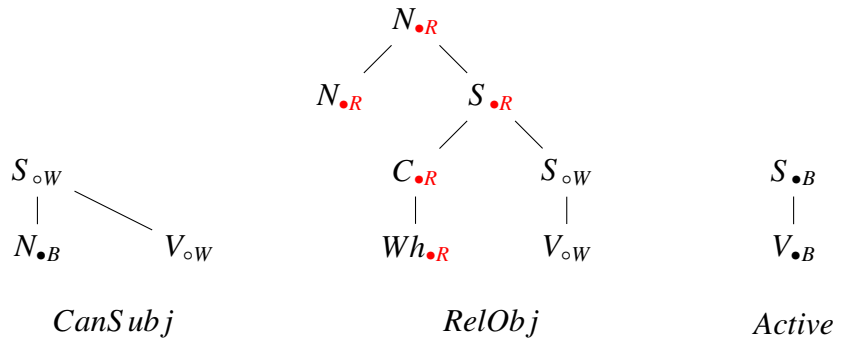


FIGURE 3.15 — Arbres élémentaires colorés

L'unique modèle valide de la combinaison contrainte par les couleurs est celui attendu [4]. Il est le seul modèle ne contenant que des noeuds rouges et noirs.

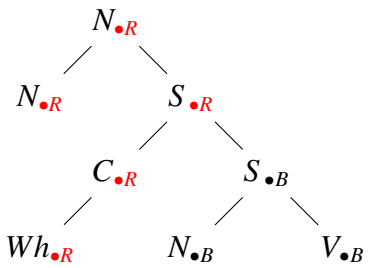
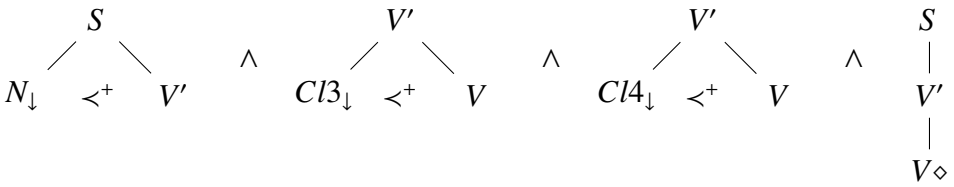


FIGURE 3.16 — Combinaisons filtrées par couleurs

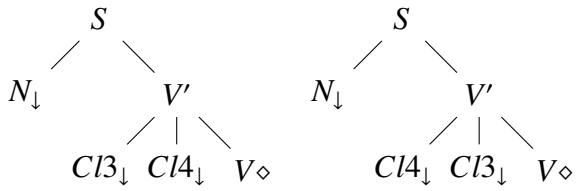
Des contraintes sur l'ordre des noeuds

En français, un certain nombre de pronoms clitiques peuvent s'insérer entre le sujet et le verbe. Par exemple, dans la phrase "Je la lui donne", "la" et "le" sont des pronoms clitiques. Leur ordre est fixe ("Je lui la donne" n'est par exemple pas une phrase valide). La construction de la règle pour cet exemple peut être faite à partir de ces quatre fragments :



Les modèles de la combinaison sont les suivants :

4. On aurait également pu obtenir cet unique modèle en ajoutant des ancres lexicales aux noeuds V des fragments $CanSubj$ et $RelObj$: ces trois noeuds seraient obligatoirement unifiés en raison de la contrainte d'unicité des ancres lexicales dans les arbres élémentaires.



Seul le premier modèle doit être un modèle valide. Pour cette raison, des contraintes supplémentaire doivent être spécifiées dans la description. Écrire de façon explicite la relation de précédence linéaire entre les noeuds des deux clitiques serait une fois encore trop contraignant.

Le principe *rank* permet de décorer certains noeuds de la description avec des rangs. Lorsque ce principe est activé, les fils de tous les noeuds doivent être ordonnés de façon à respecter les contraintes imposées par les rangs. Par exemple, un noeud étiqueté avec `rank=2` ne pourra pas être un frère droit d'un noeud étiqueté avec `rank=3`.

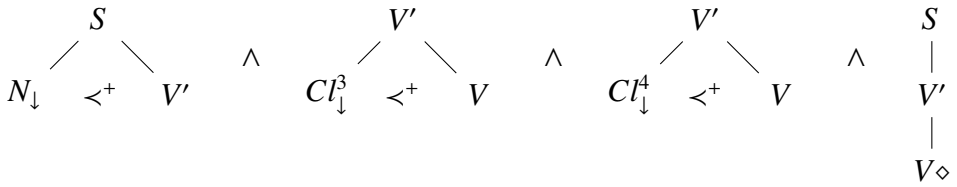


FIGURE 3.17 — Fragments décorés avec le principe *rank*

Le seul modèle de la combinaison est maintenant le suivant :

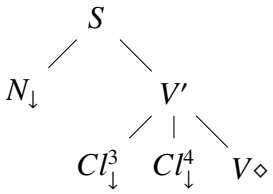


FIGURE 3.18 — Unique modèle après filtrage par le principe *rank*

Syntaxe concrète

Le premier des objectifs de validation pour l'approche que nous proposerons dans cette thèse est de pouvoir recréer la solution proposée par XMG. Pour voir plus clairement le genre de langages dédiés pour lequel nous souhaitons donner une méthode d'assemblage, nous allons passer en revue les éléments de la syntaxe concrète d'une métagrammaire XMG.

La structure d'un fichier (ou d'un ensemble de fichiers) de métagrammaire est la suivante : tout d'abord, d'éventuelles inclusions de fichiers, puis les déclarations, ensuite les classes, et pour finir les évaluations.

Entêtes. Pour des raisons évidentes de confort de développement, une description métagrammaticale peut être répartie sur plusieurs fichiers. Un mécanisme d'inclusion est donc nécessaire. Une inclusion de fichier se fait grâce au mot-clé **`include`**.

Les déclarations composent la partie constante de la métagrammaire. Elles incluent les déclarations des types manipulés dans la grammaire, mais également des exclusions mutuelles entre classes (incompatibilité entre fragments). Des déclarations de types associés à des traits peuvent aussi être effectuées.

Le mot-clé **`type`** permet de définir un type, qui pourra être associé à des données de la métagrammaire. On peut donner les contraintes sur les valeurs acceptables pour les types de différentes manières.

Il est possible de déclarer des types énumérés, en listant les valeurs possibles entre accolades.

```
type CAT = {s,n,v}
```

Des types intervalles sont aussi disponibles pour les valeurs entières, en déclarant les bornes.

```
type NUM = [1..3]
```

Enfin, il est possible de déclarer des types de structures de traits, en donnant la liste des types des valeurs pour chacun des traits.

```
type FS = [cat: CAT, agr: [num:NUM, gen:GEN]]
```

Dans la version initiale d'XMG, l'unique manière d'agir sur le typage de la métagrammaire est d'associer aux traits des types :

```
feature cat : CAT
```

Cette déclaration implique que tout trait `cat` apparaissant dans une structure de traits doit avoir une valeur de type CAT.

Classes La définition d'une classe débute par le mot-clé **`class`** suivi de son identifiant et de ses paramètres. Vient ensuite la configuration de la portée des variables : on choisit quelles variables sont déclarées dans cette classe, lesquelles de ces variables seront exportées, et les classes qui seront importées. Les classes importées rendent accessibles dans la classe courante l'ensemble des variables qu'elles exportent.

```
class MaClasse[P1,P2]
import ?MonAutreClasse
export ?S
declare ?S ?C ?V
```

Vient ensuite le corps de la classe, correspondant à la description de l'abstraction qu'elle réalise. Le langage de contrôle intervient ici. Les deux opérateurs logiques (**`|`** et **`;`**) permettent de combiner deux types de données : une contribution à une dimension, délimitée par le nom de cette dimension entre chevrons ou un appel à une classe désignée par son identifiant.

Dimensions. Les langages permettant les contributions aux dimensions sont propres aux dimensions en question. Ils sont conçus pour rendre possible une description naturelle des structures manipulées à l'intérieur de ces dernières. Ils incluent le langage de contrôle (les opérateurs logiques, l'opérateur = et les appels de classe) ainsi que des littéraux construits à partir de mots-clés et opérateurs spécifiques. Pour le langage de description d'arbres, le mot clé **node** permet de déclarer un noeud, et les opérateurs ->, ->*, ->+, >>, >>*, >>+, permettent d'exprimer les relations entre noeuds.

```
{
  <syn>{
    node ?X;
    node ?Y;
    ?X -> ?Y
   }
 |
   ?C= UneAutreClasse[]
}
```

Valuations La dernière partie du fichier de métagrammaire consiste à sélectionner les classes qui seront évaluées par le compilateur, soit les axiomes de la métagrammaire. Une évaluation est déclenchée par le mot-clé **value** suivi de l'identifiant de l'axiome.

```
value MaClasse
```

XMG, comme la plupart des approches que nous avons étudiées pour la description des grammaires d'unification, propose des moyens d'abstraction. La combinaison de ces abstraction est en revanche faite de façon totalement déclarative (grâce aux opérateurs logiques), et la séparation des différents niveaux de description linguistique est facilitée par le concept de dimensions. L'une des limites évidentes est le manque de flexibilité concernant ces dimensions : elles sont limitées au nombre de trois (syntaxe, sémantique et interface) et s'adressent donc à une communauté d'utilisateurs réduite.

3.4 Syntaxe basée sur la théorie des modèles

Les approches que nous avons étudiées jusqu'ici sont basées sur la réécriture, c'est à dire que la construction de la syntaxe est effectuée par application de règles successives. Les approches que nous présentons dans cette section ne sont pas basées sur des règles de réécriture, mais sur un ensemble de règles de bonne formation sur les énoncés. Les grammaires appartenant à ces formalismes ont donc des formes très différentes. Leur popularité naissante du fait de leur robustesse (vision non binaire de la grammaticalité) en fait des objets d'étude intéressants. De plus XDG (présenté en 3.4.1) inclut des concepts sur lesquels nous aimerions nous appuyer dans la suite de cette thèse.

3.4.1 eXtensible Dependency Grammar

Dans sa thèse, Ralph Debusmann [Debusmann, 2006] s'intéresse au développement de grammaires appartenant à un autre formalisme, les grammaires de dépendances. Ce formalisme n'appartient pas à la famille des grammaires génératives et transformationnelles, mais suit la théorie

structurale de Lucien Tesnière ([Tesnière, 1959]). Il propose pour cette tâche un formalisme, eXtensible Dependency Grammar (XDG), ainsi que son implémentation.

Une analyse par une grammaire de dépendance est un arbre de dépendance, ou un graphe de dépendance, dans lequel les noeuds sont associés aux mots de la phrase, et reliés par des arètes symbolisant les relations de dépendance, ou les fonctions grammaticales. La figure 3.19 montre une analyse par dépendance. Elle indique notamment que *eat* est une dépendance de *wants*, que *spaghetti* est son objet et *to* sa particule.

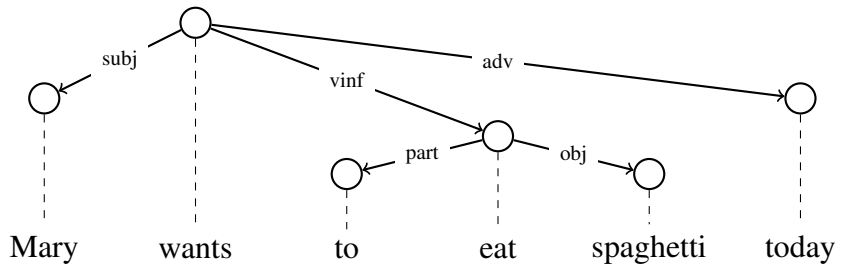


FIGURE 3.19 — Exemple d'analyse par dépendance (extrait de [Debusmann, 2006])

$$\left\{ \begin{array}{ll} word = & eat \\ in = & \{vinf?\} \\ out = & \{part!, obj!\} \end{array} \right\}$$

FIGURE 3.20 — Exemple d'entrée lexicale (extrait de [Debusmann, 2006])

Dans les formalismes grammaticaux de type Grammaire Syntagmatique (Phrase Structure Grammar), comme TAG, une grammaire est traditionellement composée d'un ensemble de règles de production (des arbres élémentaires dans le cas de TAG), comme nous l'avons vu précédemment. Une grammaire de dépendance est quant à elle exprimée en terme de valences : à chaque noeud est associé une valence entrante (l'ensemble des arètes entrantes requises) et une valence sortante (l'ensemble des arètes sortantes requises). Un exemple d'entrée lexicale est donné dans la figure 3.20, dans laquelle ! marque l'obligation et ? l'optionalité. L'entrée indique donc que le mot *eat* peut avoir une dépendance entrante étiquetée $vinf$ et doit avoir deux dépendances sortantes étiquetées *part* et obj.

L'approche est donc radicalement différente des précédentes, puisqu'une grammaire de dépendance ne présente pas le genre de redondances observées en TAG.

La contribution en terme de modularité de ce travail est de proposer un système de dimensions, permettant de séparer les différents niveaux de description linguistique, ainsi qu'un ensemble de principes. Un principe est un ensemble de contraintes de bonne formation sur les modèles produits par XDG. Par exemple, utiliser le principe *tree* pour une dimension contraint les modèles de cette dimension à être des arbres.

Pour faciliter le développement de lexiques de grande taille, la métagrammaire permet de faire des factorisations sur les entrées lexicales définies et de combiner ces factorisations.

L'exemple de la figure 3.21 présente l'entrée lexicale pour le verbe "wants", où *word*, *syn*, *sem* et *synsem* correspondent à des dimensions (*word* pour la dimension lexicale, *syn* pour la

$$\left\{\begin{array}{ll} word = & wants \\ syn = & \left\{\begin{array}{ll} in = & \{root?\} \\ out = & \{subj!, vinf!, adv*\} \\ order = & \{(subj\uparrow), (subj, vinf), (subj, adv), \\ & (\uparrow, vinf), (\uparrow, adv), (vinf, adv)\} \\ agrs = & \{(3, sg)\} \\ agree = & \{subj\} \end{array}\right\} \\ sem = & \left\{\begin{array}{ll} in = & \{root!, th*\} \\ out = & \{ag!, th!\} \end{array}\right\} \\ synsem = & \left\{\begin{array}{ll} arg = & \left\{\begin{array}{ll} ag = & \{subj\} \\ th = & \{vinf\} \end{array}\right\} \\ mod = & \{\} \end{array}\right\} \end{array}\right\}$$

FIGURE 3.21 — Entrée lexicale pour "wants" (extraite de [Debusmann, 2006])

syntaxique, *sem* pour la sémantique, et *synsem* pour l'interface entre les deux dimensions précédentes). Les traits *in* et *out* expriment les valences entrantes et sortantes. Le trait *order* contient des contraintes d'ordre partiel strict entre les dépendants et entre l'ancre (↑) et les dépendants. *agrs* spécifie les accords et *agree* l'ensemble des dépendants avec lesquels l'accord doit se faire. *arg* est un vecteur associant les étiquettes des arêtes de *sem* à des ensembles d'étiquettes d'arêtes de *syn* pour contraindre la réalisation des arguments sémantiques des verbes par des fonctions grammaticales. *mod* est un ensemble d'étiquettes d'arêtes de *sem* utilisé pour contraindre la réalisation des arguments sémantiques des adverbes par leurs parents syntaxiques.

Cette entrée est encodée pour XDG sous la forme présentée dans la figure 3.22.

```
defentry {
  dim lex {word: "wants"}
  dim syn {in: {root?}
           out: {subj! vinf! adv*}
           order: {[subj "^"] [subj vinf] [subj adv]
                   ["^" vinf] ["^" adv] [vinf adv]}
           agrs: {["3" sg]}
           agree: {subj}}
  dim sem {in: {root! th*}
           out: {ag! th!}}
  dim synsem {arg: {ag: {subj}
                    th: {vinf}}
              mod: {}}}
```

FIGURE 3.22 — Entrée lexicale pour le mot *wants* dans le formalisme XDG

Des moyens d'abstraction sont disponibles pour faciliter la création de toutes les entrées en factorisant les informations redondantes. Ces moyens d'abstractions sur les entrées lexicales sont appelés classes lexicales.

Les classes lexicales, une fois définies, peuvent être instanciées à l'intérieur de nouvelles entrées lexicales, comme dans l'exemple suivant, recréant l'entrée lexicale "wants" en utilisant des

abstractions :

```
defentry {
  "verb"
  ("intrans" | "trans")
  "vinfc"
  "fin" {Word: "wants"
    Agrs: {["3" sg]}}}
```

où | exprime la disjonction (ici, pour générer les entrées lexicales pour les alternances transitive et intransitive), et "verb", "intrans", "trans", "vinfc" et "fin" sont des abstractions. La dernière citée, correspondant aux "finite verbs", est définie de la façon suivante :

```
defclass "fin" Word Agrs {
  dim lex {word: Word}
  dim syn {in: {root?}
    out: {subj!}
    order: <subj "^" obj vinf adv>
    agrs: Agrs
    agree: {subj}}}
```

où *Word* et *Agrs* sont des arguments donnés à l'instanciation de l'abstraction (dans le cas étudié, *Word* reçoit `"wants"` et *Agrs* `{["3"sg]}`. L'entrée lexicale est donc donnée par un paramètre, et un certain nombre de traits sont ajoutés à la dimension *syn*. Une unique arête entrante optionnelle est étiquetés *root*, et une arête sortante obligatoire est étiquetée *subj*, exprimant le fait que ce type de verbes requiert un sujet dans tous les cas. Enfin, on indique que le verbe doit s'accorder avec son sujet.

La classe lexicale *verb* est exprimée de la manière suivante :

```
defclass "verb" {
  dim syn {out: {adv*}}
  dim sem {in: {root! th*}}}
```

Dans la dimension syntaxique, on précise que le verbe peut être modifié par des adverbes. Du côté sémantique, on doit avoir une arête entrante étiquetée *root* et un nombre arbitraire d'arêtes *th*, nombre correspondant au nombre d'adverbes dont le verbe est le thème.

Les verbes intransitifs appartiennent à la classe lexicale suivante :

```
defclass "intrans" {
  dim sem {out: {ag!}}
  dim synsem {arg: {ag: {subj}}}}
```

où l'unique arête sortante correspond à l'agent (*ag*).

Les verbes transitifs ont les mêmes contraintes, enrichies de nouvelles contributions aux dimensions :

```
defclass "trans" {
  "intrans"
  dim syn {out: {obj!}}
  dim sem {out: {pat!}}
  dim synsem {arg: {pat: {obj}}}}
```

En plus de l'appel à *intrans*, on doit avoir une arête sortante étiquetée *obj* pour l'objet en syntaxe, et une étiquetée *pat* pour le patient en sémantique. On indique également que le patient est réalisé par l'objet.

Enfin, les verbes nécessitant un complément infinitif :

```
defclass "vinfc" {
  dim syn {out: {vinf!}}
  dim sem {out: {th!}}
  dim synsem {arg: {th: {vinf}}}}
```

Le verbe à l'infinitif est requis dans la dimension *syn*, un thème dans la dimension *sem*. On précise dans la dimension *synsem* que le thème est réalisé par le complément infinitif.

On peut retenir de cette approche la possibilité de définir un ensemble de dimensions, chacune équipée d'un langage de contraintes propre. On notera également que XDG n'est pas l'implémentation d'un formalisme linguistique (ce qui est par exemple le cas de XLE pour LFG) mais i) un nouveau formalisme pour les grammaires de dépendances basé sur la théorie des modèles et ii) son implantation.

Nous allons maintenant nous intéresser à un autre formalisme basé sur la théorie des modèles : les grammaires de propriétés.

3.4.2 Grammaire de Propriétés

Comme dans le cas des grammaires de dépendance, les grammaires de propriétés ([Blache, 2000]) ne s'intègrent pas au courant des grammaires de réécriture, mais se basent sur un système de contraintes locales : les propriétés. Une propriété concerne un noeud et applique des contraintes à ses noeuds fils. Un des aspects intéressants des grammaires de propriétés est leur capacité à analyser des énoncés agrammaticaux. À l'analyse d'un énoncé, on calcule un score de grammaticalité, qui diminue à chaque propriété violée. Ici, nous considérerons ces six propriétés :

Obligation	A: $\triangle$B	au moins un fils B
Uniqueness	A: B!	au plus un fils B
Linearity	A: B$\prec$C	le fils B précède le fils C
Requirement	A: B$\Rightarrow$C	si un fils B, alors aussi un fils C
Exclusion	A: B$\not\Leftrightarrow$C	les fils B et C sont mutuellement exclusifs
Constituency	A: S	les fils doivent avoir des catégories dans S

Une grammaire de propriétés de taille réelle prend la forme d'une hiérarchie d'héritage de constructions linguistiques. Ces constructions se composent de structures de traits et d'un ensemble de propriétés. Les variables sont manipulées des deux côtés, et peuvent être utilisées pour partager l'information entre eux. La figure 3.23 représente une part de la hiérarchie construite pour le français dans [Guénot, 2006].

La construction V-n de la figure indique que dans les verbes avec négation en français, la négation implique la présence d'un adverbe *ne* étiqueté avec la catégorie $Adv-ng$ et/ou d'un adverbe étiqueté avec la catégorie $Adv-np$ (comme *pas*). Une contrainte d'unicité existe aussi sur ces adverbes, et un ordre linéaire doit être respecté (*ne* doit venir avant *pas*). Quand le verbe est à l'infinitif, il doit être placé après les adverbes.

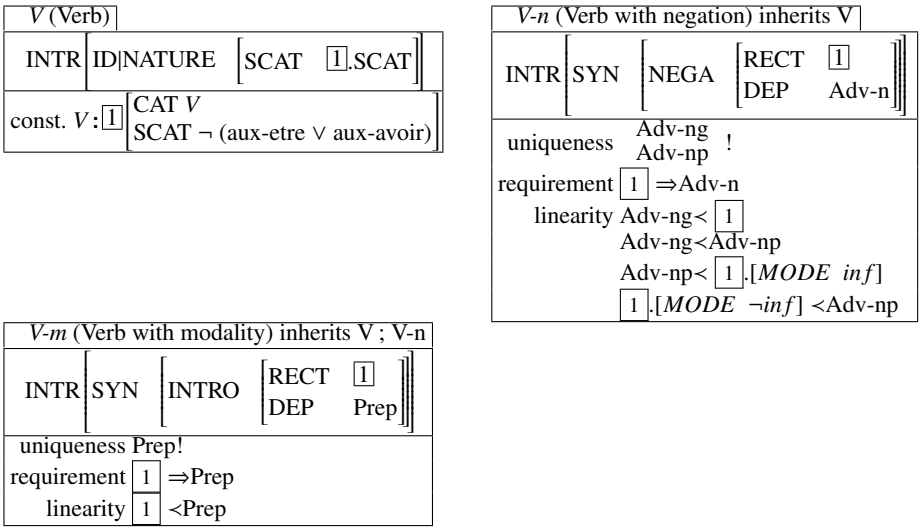


FIGURE 3.23 — Fragment d'une grammaire de propriétés pour le français (constructions verbales de base)

Pour décrire une grammaire de propriétés, nous devons pouvoir représenter des encapsulations, des variables, des structures de traits, et des propriétés. Un langage de description pour XMG a été proposé dans [Duchier *et al.*, 2011] et [Crabbé *et al.*, 2014]. Ce langage fait correspondre les encapsulations aux classes XMG, et utilise les variables et les structures de traits présentes dans d'autres langages de descriptions. Le langage proposé peut ainsi se formaliser de la manière suivante :

$$Desc_{PG} := x = y \mid x \neq y \mid [f{:}E] \mid \{P\} \mid Desc_{PG} \wedge Desc_{PG}$$
$$P := A : \triangle B \mid A : B! \mid A : B \prec C \mid A : B \Rightarrow C \mid A : B \nLeftrightarrow C \mid A : B$$

où x, y correspondent à des variables d'unification, = à l'opérateur d'unification, ≠ à l'échec d'unification, : à l'association entre l'attribut f et une expression (possiblement complexe) E, et $\{P\}$ à un ensemble de propriétés. On notera que E et P peuvent partager des variables d'unification. La traduction de la construction linguistique V-m dans ce langage serait alors :

$$V{-}m \rightarrow (Vclass \vee V{-}n) \wedge \langle PG\rangle\{[\mathsf{INTR}{:}[\mathsf{SYN}{:}[\mathsf{INTRO}{:}[\mathsf{RECT}{:}X, \mathsf{DEP}{:}\mathtt{Prep}]]]]$$
$$\wedge\ (V : \mathtt{Prep}!) \wedge (V : X \Rightarrow \mathtt{Prep}) \wedge (V : X \prec \mathtt{Prep})\}$$

L'héritage est ici rendu possible par des appels de classes. Le langage de contrôle permet même l'héritage disjonctif, nécessaire dans la classe V-m.

L'intégration de cette solution de description à XMG demande de pouvoir configurer les dimensions utilisées. C'est ce constat qui motive la nécessité d'apporter de la liberté dans la définition de nouvelles dimensions. Nous verrons dans la partie suivante comment ceci peut être rendu possible dans le cadre du développement d'une nouvelle génération de système XMG.

3.5 Conclusion

Dans ce chapitre, nous avons réalisé un inventaire d'un certain nombre d'approches proposées pour la représentation de la syntaxe. Pour chaque approche, nous avons présenté des langages

dédiés permettant la description des structures inhérentes à celle-ci, en pointant un ensemble de concepts fondamentaux (abstraction, héritage, contrôle, espace de noms, etc) auxquels nous attacherons une attention particulière.

Nous avons de plus vu qu'une approche métagrammaticale pouvait se révéler pertinente dans différentes configurations : celle des grammaires d'arbres adjoints avec XMG, mais également celle des grammaires de dépendances avec XDG, ou HPSG avec LKB/Matrix.

Adapter cette approche à de nouveaux niveaux de description linguistique, ou de nouveaux formalismes, semble donc être prometteur, mais les expérimentations sont limitées par le développement logiciel qu'elles nécessitent. En effet, tout nouveau langage de description métagrammatical implique la création d'un nouveau compilateur. C'est pourquoi dans cette thèse, nous proposons une méthode pour composer un compilateur pour la description de ressources langagières de manière à rendre le développement moins contraignant.

Dans le chapitre suivant, nous nous intéresserons à des solutions créées pour faciliter, ou automatiser, le développement de compilateurs.

Construction de compilateurs

4

Sommaire

4.1 Introduction **57**

4.2 Compilateurs de Compilateurs **58**

4.2.1 CDL3 58

4.3 Génération de Composants Spécifiques de Compilateurs **60**

4.3.1 Génération d'analyseurs syntaxiques avec Yacc 60

4.3.2 Génération d'analyseurs syntaxiques avec Parsec 61

4.4 Construction modulaire de DSL **62**

4.4.1 Neverlang 62

4.5 Conclusion **64**

4.1 Introduction

Nous nous sommes jusqu'ici concentrés sur la diversité des méthodes utilisées pour représenter la langue. Nous avons observé une grande hétérogénéité des langages utilisés pour la description de différents niveaux de la langue, mais également à l'intérieur de ces niveaux. Ceci implique que la création d'un outil adapté à un nouveau formalisme, ou rassemblant plusieurs formalismes, nécessite un travail conséquent, puisque les outils existants et leurs langages de description ont une expressivité qui se limite aux structures propres à une approche. Concevoir un outil qui soit indépendant d'une approche demande d'avantage de composabilité.

Notre but est de fournir une solution basée sur les concepts de XMG, facilement adaptable à différentes tâches de description de ressources de ce type. Cette solution est une méthode d'assemblage de compilateurs. Ce chapitre concerne les approches existantes pour la génération de compilateurs. Rappelons que dans notre cas, contrairement à certaines des approches que l'on présente ici, l'ambition n'est pas de créer des compilateurs arbitraires, mais de valider la méthodologie dans le cadre spécifique des compilateurs pour la description de ressources linguistiques: le but de notre démarche est en d'autres mots de permettre l'assemblage de DSL par des non-spécialistes, ceci en mettant de côté l'optimisation et les langages d'instructions.

L'étroite relation entre les grammaires et les programmes qui leurs sont liés, notamment les compilateurs, a généré un certain nombre de travaux. Ces travaux incluent la génération automatique de compilateurs à partir de grammaires. Ces outils sont appelés compilateurs de compilateurs, ou générateurs de compilateurs, et sont généralement orientés vers un modèle,

restreignant les compilateurs produits à ceux pour des langages proches du modèle [Aho *et al.*, 1998]. Nous nous intéresserons brièvement aux approches de ce type dans la section 4.2.

Des outils dédiés à la génération de composants spécifiques de compilateur existent également. Ils comprennent les constructeurs d'analyseurs syntaxiques et lexicaux, les moteurs de traduction dirigée par la syntaxe, les générateurs de code automatiques et les moteurs d'analyse du flot de données [Aho *et al.*, 1998]. L'objectif est de réutiliser au maximum les parties communes à des compilateurs, ou des familles de compilateurs. Par exemple, réécrire un algorithme d'allocation de registres pour chaque compilateur constitue une tâche laborieuse que l'on souhaiterait éviter. Dans la section 4.3, nous donnerons des exemples de tels outils.

4.2 Compilateurs de Compilateurs

Un compilateur permet de passer d'un programme dans un langage source à un nouveau programme dans un langage cible. Le compilateur étant lui même un programme, il est naturel qu'il puisse lui aussi être produit par compilation. Cette observation a donné naissance à un certain nombre d'outils appelés compilateurs de compilateurs.

4.2.1 CDL3

Dans [Koster et Beney, 1991], un langage de description de compilateur nommé CDL3, basé sur les Grammaires Affixes Étendues [Watt, 1974], est proposé. Ce type de programmation, basée sur les structures syntaxiques, est appelé Programmation Grammaticale, et tire parti d'un ensemble d'homomorphismes entre les différents langages manipulés par un programme (la syntaxe et l'entrée, la syntaxe abstraite et la sortie, l'entrée et la sortie, etc.).

Un programme CDL3 se compose d'un ensemble de méta règles et de règles de syntaxe. Les premières jouent le rôle de déclarations de type, et les secondes de procédures. On attribue à chaque procédure un type, selon deux critères : le fait qu'elle modifie l'état du programme et le fait qu'elle puisse échouer. Cela donne quatre types :

Une procédure de type `PREDICATE` a un effet et peut échouer, une `ACTION` a un effet et ne peut pas échouer, un `TEST` n'a pas d'effet et peut échouer. Enfin, une procédure de type `FUNCTION` n'a pas d'effet et ne peut pas échouer.

Un exemple issu de [Koster et Beney, 1991] propose la génération d'un compilateur pour la syntaxe suivante :

```
program: block.
block: "begin", units, "end".
units: unit, units; unit.
unit: application; definition; block.
application: "apply", identifier.
definition: "define", identifier.
```

CDL3 permet de générer un analyseur syntaxique construisant pour chaque bloc analysé une table de symboles locaux `RANGE`, avec les procédures suivantes :

```
R:: RANGE.
ACTION program:
  block(0)
```

```
ACTION block(>BNO):
  should be token("begin")
    units(BNO, 0|OFFSET, empty|RANGE),
      should be token("end").
ACTION units(>BNO, >OFFSET>, >R>):
  application, units tail;
  definition(BNO, OFFSET, R), units tail;
  block(BNO+1), units tail
WHERE units tail:
  is token(";"), units; +.

PRED application:
  is token "apply",
  ( is identifier(IDF);
    error("identifier expected")).
PRED definition(>BNO, >OFFSET|OFFSET+1>, >R>):
  is token("define"),
  ( is identifier(IDF), [R where IDF has OFFSET in BNO =: R];
    error("identifier expected")).
```

où `R:: RANGE` est une méta règle qui signifie que `R` est une variable de type `RANGE` (type qui doit être défini). Les symboles > à gauche et à droite d'un paramètre indiquent respectivement un paramètre entrant et sortant. `BNO` est le numéro du bloc en cours d'analyse, `OFFSET` la position de la variable en cours d'analyse. La procédure `units tail` est automatiquement paramétrée par `BNO`, `OFFSET` et `R` par héritage de `units`. + indique une alternative qui n'échoue jamais. Le prédicat `definition` enrichit `R` avec une entrée associant à l'identifiant défini `IDF` une nouvelle position (`OFFSET`) dans la table locale.

L'analyse réalisée par ce code constitue un premier passage dans le processus de compilation. Pour traiter la suite des étapes (notamment la génération de code), ces procédures doivent être enrichies avec de nouveaux éléments, séparés de ceux appartenant au premier passage par le symbole /.

La procédure `units` modifiée pour effectuer un second passage a la forme suivante :

```
ACTION units(>BNO, >OFFSET>, >R> / >ENV, >CODE>):
  application(/ ENV, CODE), units tail;
  definition(BNO, OFFSET, R), units tail;
  block(BNO+1 / ENV, CODE), units tail
WHERE units tail:
  is token(";"), units; +.
```

Deux nouveaux paramètres, `ENV` pour l'environnement et `CODE` pour le code généré, font leur apparition. Ils sont nécessaires pour le second passage de `application`, mais pas celui de `definition`, pour lequel tout le travail est réalisé au premier passage.

La procédure `application` est modifiée comme suit :

```
PRED application:
  is token "apply",
  ( is identifier(IDF),
```

```
      (found(IDF, R, OFFSET, BNO),
         [CODE apply BNO OFFSET =: CODE];
        error("missing definition for "+IDF);
      error("identifier expected")).
```

où `appli` correspond à l'application de l'identifiant à la position `OFFSET` et au bloc numéro `BNO`.

Nous venons de voir que des outils tels que CDL3 peuvent générer à partir d'une description l'intégralité d'un compilateur. Cependant, l'élaboration de cette description demande une expertise certaine dans les concepts de la compilation, notamment pour ce qui est de la construction des contextes.

4.3 Génération de Composants Spécifiques de Compilateurs

Un processus de compilation classique comprend plusieurs étapes, dont chacune consiste à transformer le programme pour qu'il adapte un nouveau langage. Le premier de ces langages est le langage source, adapté à la description, et le dernier le langage cible, dont les instructions sont exécutables. Les premières étapes de la compilation correspondent à une analyse syntaxique du programme, écrit dans un langage exprimable avec une grammaire hors contexte.

Coder un analyseur syntaxique pour un langage dédié est une tâche répétitive et qui demande d'utiliser des opérations de bas niveau. Une automatisation de cette tâche peut être obtenue par l'utilisation d'un langage de plus haut niveau, permettant de s'abstraire de la syntaxe concrète du langage de programmation. Nous verrons dans cette section deux outils de génération automatique d'analyseurs syntaxiques : Yacc et Parsec.

4.3.1 Génération d'analyseurs syntaxiques avec Yacc

Yacc (Yet another compiler compiler) est un constructeur d'analyseurs LALR (Look Ahead LR). La technique LALR est préférée à la LR en pratique, pour la taille très inférieure de ses tables, rendant l'analyse significativement plus efficace.

Le principe de Yacc est de générer, à partir d'un programme source, l'analyseur syntaxique décrit par ce programme. La description suit la structure suivante :

```
declarations
%%
regles de traduction
%%
routines C annexes
```

Les déclarations sont essentiellement celles des terminaux de la grammaire. Un terminal t est déclaré avec la syntaxe

```
%token t
```

La partie règles de traduction contient la grammaire à proprement parler, c'est à dire une suite de règles de réécriture. Une règle est composée de la façon suivante :

```
gauche : alternative 1 {action semantique 1}
       | alternative 2 {action semantique 2}
       ...
       | alternative n {action semantique n};
```

où *gauche* est un non-terminal, $alternative_1 \ldots alternative_n$ des suites de symboles terminaux (déclarés dans la partie précédente, ou placés entre apostrophes) et non terminaux (pour toute autre chaîne). Le symbole de la partie gauche de la première production est considéré comme l'axiome de la grammaire.

Les actions sémantiques sont des séquences d'instruction C, dans lesquelles on peut faire référence aux symboles des parties gauche et droite de la règle. Ces références se font au moyen du symbole $$ pour le non terminal *gauche*, et des symboles $1 ... $*n* pour les n symboles de la partie droite. Quand une réduction est opérée sur une règle, son action sémantique est effectuée, dans le but de construire la représentation sémantique de la partie gauche en fonction des valeurs sémantiques des symboles de droite.

Par exemple, si l'on souhaite manipuler des expressions arithmétiques utilisant les opérateurs + et −, et évaluer ces expressions à l'analyse syntaxique, on peut définir le programme suivant :

```
%token nombre

%%

expr : expr '+' expr {$$ = $1 + $3}
     | expr '-' expr {$$ = $1 - $3}
     | nombre {$$ = $1};

%%
```

4.3.2 Génération d'analyseurs syntaxiques avec Parsec

Parsec est une bibliothèque d'assemblage de parsers pour Haskell, qui consiste à combiner des fonctions élémentaires d'analyse pour construire des analyseurs plus complexes ([O'Sullivan *et al.*, 2008]).

Deux exemples de ces fonctions élémentaires, rendant le code d'un analyseur plus compact, sont `sepBy` et `endBy`. Ces deux fonctions prennent deux arguments, le premier étant une fonction d'analyse de contenu, et le second une fonction d'analyse de séparateur. `sepBy` renvoie la liste des contenus (initialement séparés par le séparateur donné en argument) qu'il a pu analyser et `endBy` la liste des contenus analysés (chacun d'entre eux étant initialement suivi du séparateur).

L'exemple suivant, extrait de [O'Sullivan *et al.*, 2008], montre l'utilisation de ces fonctions pour l'analyse d'un document CSV. Chaque ligne d'un tel fichier est un enregistrement, et chaque champ de cet enregistrement est séparé du suivant par une virgule.

```
-- file: csv2.hs
import Text.ParserCombinators.Parsec

csvFile = endBy line eol
line = sepBy cell (char ',')
```

```
cell = many (noneOf ",\n")
eol = char '\n'

parseCSV :: String -> Either ParseError [[String]]
parseCSV input = parse csvFile "(unknown)" input
```

`cvsFile` analyse une suite de lignes séparées par le symbole de fin de ligne (`\n`). `line` analyse une suite de cellules séparées par des virgules. `many` est une fonction permettant d'appliquer une fonction d'analyse (donnée en argument) de façon répétée, et `noneOf` une autre fonction qui permet l'analyse d'une suite de caractères jusqu'à rencontrer l'un de ceux passés en argument (ici, "," et une fin de ligne). `cell` correspond donc à une liste de chaînes de caractères. Lors de l'analyse d'une chaîne avec la fonction cvsFile, la fonction `cell` segmente cette chaîne, et selon le caractère séparant un élément du suivant (virgule ou fin de ligne), cet élément est ajouté à la liste de ceux composant la ligne ou la liste des éléments de la ligne est ajoutée à la liste des lignes.

Le retour d'une exécution du code sur la chaîne `"Hi,\n\n,Hello\n"` est le suivant :

```
parseCSV "Hi,\n\n,Hello\n"

Right [["Hi",""],[""],["","Hello"]]
```

où **`Right`** signifie que l'analyse a réussi. La liste qui le suit est le résultat de l'analyse, chacune des sous listes correspondant à une ligne, et les éléments de ces listes aux cellules.

Les techniques implémentées par Yacc et Parsec permettent donc d'obtenir assez rapidement des analyseurs syntaxiques pour des langages, en combinant des fonctions d'analyse élémentaires, ou en se basant sur la grammaire hors contexte décrivant le langage désiré. Les étapes suivantes du processus de compilation restent en revanche à coder.

4.4 Construction modulaire de DSL

Nous avons récemment découvert un ensemble d'approches visant comme la nôtre à créer des langages dédiés (ou DSL) par assemblage de modules. La découverte tardive de ces approches s'explique par le fait qu'elles appartiennent à un domaine d'ingénierie très éloigné de la linguistique. Les outils proposés pour effectuer cette tâche incluent JastAdd ([Ekman et Hedin, 2007]), LISA ([Henriques *et al.*, 2005]), et Neverlang ([Vacchi *et al.*, 2013]). Nous étudierons ce dernier dans cette section.

4.4.1 Neverlang

Composants

La construction d'un DSL avec Neverlang est effectuée par l'assemblage de *modules*, qui peuvent contenir des *définitions syntaxiques* ou des *rôles sémantiques*. On associe une syntaxe à l'action sémantique exécutée quand cette syntaxe est reconnue au moyen d'une *slice*. La figure 4.1 montre le code d'une slice (`AddExprSlice`) définissant la syntaxe et la sémantique d'une somme.

La définition de la syntaxe se compose d'un ensemble de règles hors contexte. Les nombres utilisés dans la définition du rôle sémantique correspondent aux indices des non terminaux :

```
module com.example.AddExpr {
  reference syntax {
    AddExpr <- Term;
    AddExpr <- AddExpr "+" Term;
  }
  role(evaluation) {
    0 { $0.value= $1.value; }
    2.{ $2.value= (Integer)$3.value + (Integer)$4.value; }
  }
}

slice com.example.AddExprSlice {
  concrete syntax from com.example.AddExpr
  module com.example.AddExpr with role evaluation
}
```

FIGURE 4.1 — La définition d'une *slice* Neverlang définissant la syntaxe et la sémantique d'une somme

`0` désigne le non terminal `AddExpr` de la première règle et `4` le non terminal `Term` de la seconde. La deuxième règle du rôle indique que la valeur sémantique construite lorsque la règle `AddExpr <- AddExpr "+"Term;` est réduite est la somme des valeurs sémantiques construites pour les deux non terminaux de la partie droite.

Processus de compilation

Dans un premier temps, l'arbre de syntaxe abstraite du programme en entrée est construit. Puis on applique sur cet arbre les actions sémantiques définies par les rôles.

La définition d'un rôle est propre à une phase de compilation. Le rôle défini dans la figure 4.1 est par exemple un rôle de la phase `evaluation`. Pour chaque phase, on effectue un nouveau parcours de l'arbre de syntaxe abstraite en effectuant les actions sémantiques appropriées.

Assemblage

La génération d'un DSL est effectuée à partir des slices, dans un fichier unique. La première étape est de sélectionner l'ensemble des slices à utiliser, parmi les composants de langages existants. Chaque slice utilisée peut avoir des besoins et offrir des ressources. Les ressources offertes par une slice sont les non terminaux utilisés dans les parties gauches de règles : ces non terminaux sont rendus disponibles dans les règles des autres slices. Les besoins d'une slice sont les non terminaux apparaissant dans les parties droites de ses règles, et pour lesquels il n'existe pas de règle de production localement. Dans l'exemple 4.1, le non terminal `Term` est un besoin qui doit être satisfait par une autre slice.

Lorsque les slices ont été sélectionnées, un graphe de dépendances correspondant à la satisfaction des besoins et ressources est généré. La résolution des dépendances est faite par le développeur du DSL en collaboration avec un spécialiste de l'assemblage avec Neverlang.

Un ensemble de slices, comparables à des bibliothèques, sont fournies avec Neverlang. Ces slices contiennent la définition de littéraux usuels, comme les chaînes de caractères (définies

dans le slice `neverlang.commons.SimpleTypes`). Elles contiennent également la définition de rôles sémantiques, comme `terminal-evaluation`, fourni par `SimpleTypes`, qui effectue l'évaluation des terminaux définis dans la slice.

```
language com.example.CalcLang{
  slices com.example.AddExprSlice
         com.example.MultExprSlice
         com.example.ParenExprSlice
         com.example.ExprAssocSlice
         com.example.NumberSlice

  roles syntax < evaluation
}
```

FIGURE 4.2 — La définition d'un langage Neverlang

La dernière information que contient le fichier de définition de langage est l'ordre d'exécution des phases de compilation. Dans l'exemple 4.2, `roles syntax < evaluation` indique que les rôles successifs appliqués à l'arbre de syntaxe abstraite sont l'analyse syntaxique (`syntax`), puis `evaluation` (pour lequel nous avons vu une définition de rôle dans l'exemple 4.1).

La modularité de développement de DSL offerte par Neverlang se manifeste donc à plusieurs niveaux :

- le langage de description et la sémantique qui lui est associée sont divisés en fragments et répartis dans des *modules*
- on associe syntaxe et sémantique (qui sont a priori indépendants) au moyen de *slices*, correspondant à des *features*
- la *definition du langage* combine un ensemble de slices
- elle permet également de configurer l'ordre des phases de compilation

4.5 Conclusion

Dans ce chapitre, nous avons présenté un ensemble de méthodes visant à faciliter le développement de compilateurs, notamment en générant automatiquement certains modules du processus de compilation. D'autres approches, notamment Neverlang, permettent la construction d'un DSL à partir d'une spécification et de modules.

Dans la partie qui suit, nous proposons une méthode permettant l'assemblage d'un compilateur dédié à la description linguistique, pour laquelle les langages utilisés présentent une grande hétérogénéité, comme nous l'avons noté dans les chapitres précédents.

Notre approche permet l'assemblage intégral d'une chaîne de compilation, ce qui la classe dans la catégorie des compilateurs de compilateurs. Certains composants spécifiques des compilateurs générés le sont automatiquement, en s'inspirant largement de la méthode implémentée par Yacc.

Deuxième partie

Un Compilateur Modulaire

Introduction

Dans cette partie, nous décrivons la contribution de la thèse, à savoir une solution pour la génération de ressources langagières adaptable aux besoins de l'utilisateur. Cette adaptabilité est rendue nécessaire par l'hétérogénéité des langages de description notée dans la partie précédente.

Les chapitres de cette partie montrent comment il est possible de générer automatiquement un outil adapté en se basant sur une description. Cette description doit permettre l'assemblage d'un compilateur. Dans la suite de ce manuscrit, nous nommerons l'implémentation de cette méthode d'assemblage XMG-2. [1]

Dans le chapitre 5, nous présentons les différentes manifestations de la modularité que nous souhaitons exploiter dans XMG-2. Le chapitre 6 détaille la méthodologie sur laquelle nous nous appuyons pour assembler un langage dédié. Les chapitres 7 et 8 expliquent comment cette méthodologie peut permettre non seulement d'assembler un langage de description, mais également l'intégralité des étapes de compilation et de génération incluses dans la chaîne de traitement.

1. Le choix de ce nom est motivé par le fait que les compilateurs ainsi assemblés reposent sur une architecture et des concepts (classes, langage de contrôle, dimensions) hérités de XMG.

Différents niveaux de modularité

5

Sommaire

5.1 Introduction **69**
5.2 Un besoin de modularité **69**
5.3 Différents types d'utilisateurs **70**
5.4 Une architecture modulaire **71**
5.5 Modularité linguistique **72**
5.6 Modularité d'assemblage **74**
5.7 Modularité du compilateur et de l'exécuteur **75**
5.7.1 Compilateur 75
5.7.2 Exécuteur 75
5.8 Conclusion **76**

5.1 Introduction

Ce chapitre a pour objectif d'argumenter en faveur du choix d'une approche modulaire, dont les premières idées sont présentées dans [Petitjean, 2014]. Dans les deux premières sections, nous donnons les raisons qui nous ont poussés à nous appuyer sur cette notion de modularité. Ensuite, nous détaillons comment la modularité peut jouer un rôle central à différents niveaux dans la chaîne de traitement que nous proposons.

5.2 Un besoin de modularité

Afin d'étendre l'approche métagrammaticale de XMG à d'autres formalismes ou niveaux de représentation linguistiques, la structure du compilateur doit être revue. La création d'un compilateur dédié pour chaque nouvelle configuration serait une solution trop longue et trop complexe, puisqu'elle nécessiterait des compétences très différentes du simple développement de grammaires.

La solution que nous proposons est une approche modulaire, inspirée des méthodes de développement logiciel. L'idée est de pouvoir assembler un compilateur métagrammatical à partir de parties élémentaires de compilateur. Concrètement, nous aimerions pouvoir faire l'analogie entre l'assemblage d'un compilateur et l'assemblage d'une construction avec des briques LEGO™.

La principale raison qui motive une approche modulaire pour un outil de développement est ce constat : habituellement, quand un utilisateur souhaite décrire des structures non disponibles dans un langage de programmation, la solution est d'exprimer ces structures en utilisant celles fournies par le langage. Ceci mène à des descriptions peu naturelles des structures, et à un effort particulier pour l'utilisateur. L' approche modulaire proposée ici permet d'éviter à l'utilisateur de devoir s'adapter au langage : le langage doit s'adapter à l'utilisation. Avoir un langage de description reflétant au maximum les intuitions linguistiques de la métagrammaire renforce de plus l'idée que la description métagrammaticale est une ressource linguistique en elle-même.

5.3 Différents types d'utilisateurs

XMG, de par son extensibilité, doit pouvoir s'adresser à différents profils d'utilisateurs, reflétant différents niveaux d'expertise dans le développement de métagrammaires ou de connaissance des compilateurs.

Profil utilisateur. Le premier profil correspond à l'utilisateur final, c'est à dire le développeur de la ressource linguistique. La modularité offerte à ce niveau est celle donnée par l'approche métagrammaticale, c'est à dire les abstractions par le biais de classes ou encore l'utilisation des dimensions. Pour cet utilisateur, l'idéal est une solution "clé en main", par exemple d'avoir à sa disposition un ensemble de compilateurs prêts à l'utilisation.

Profil assembleur. Le deuxième profil est celui de l'utilisateur souhaitant assembler son propre compilateur sans avoir à contribuer au code source. L'objectif de ce type d'utilisateur est de pouvoir créer un cadre métagrammatical adapté à la ressource qu'il désire produire, sans devoir connaître de façon experte le fonctionnement du compilateur. La modularité disponible à ce niveau consiste à pouvoir assembler un cadre de développement à partir de blocs. Ces blocs, réutilisables et indépendants, correspondent à l'ensemble des étapes de compilation pour une fonctionnalité donnée.

Profil programmeur. Enfin, le troisième profil correspond à l'utilisateur contributeur. Ce dernier doit pouvoir ajouter de nouvelles fonctionnalités spécifiques au compilateur, et ceci de manière indépendante des fonctionnalités déjà intégrées. La contribution peut aller d'une simple adaptation d'un langage de description à la création d'un nouveau solveur.

Pour satisfaire les besoins de ces types distincts d'utilisateurs, différents types de bibliothèques sont mis à disposition. Les bibliothèques les plus proches de l'utilisateur final sont un ensemble de compilateurs déjà assemblés, ainsi qu'un ensemble de ressources écrites pour ces compilateurs. L'utilisateur contributeur puisera quant à lui dans une bibliothèque de briques de compilateur prêtes à être assemblées. Enfin l'utilisateur désirant contribuer au compilateur pourra enrichir les librairies précédemment citées.

Ces bibliothèques sont mises à disposition des utilisateurs sous la forme de contributions. L'installation d'une contribution donne l'accès à un ensemble de fonctionnalités pouvant aller d'un ensemble de briques nécessaires à une tâche (par exemple, l'ensemble des briques permettant l'intégration d'une dimension) à un compilateur assemblé.

5.4 Une architecture modulaire

La modularité est présente à plusieurs niveaux de la chaîne de traitement que nous proposons, de la conception d'un compilateur dédié à une tâche à la description d'une ressource. Le schéma 5.1 illustre l'architecture de la chaîne de traitement modulaire.

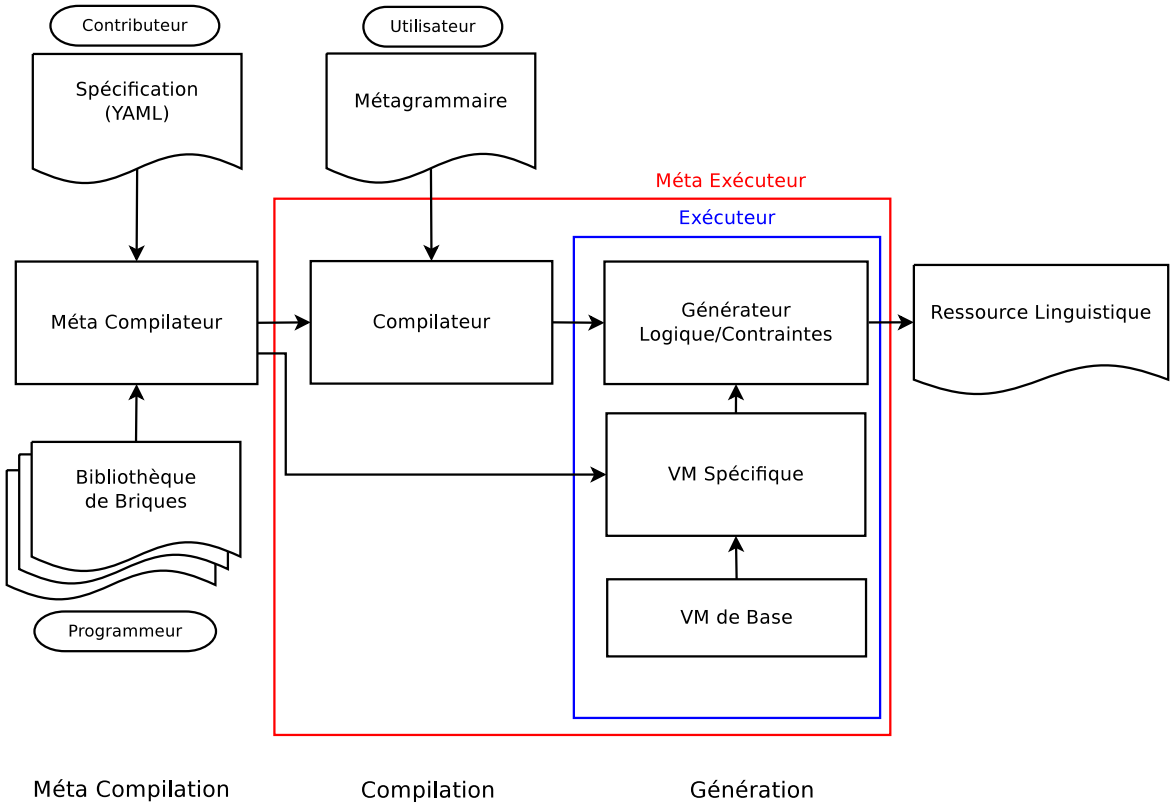


FIGURE 5.1 — L'architecture modulaire de XMG

- le `Méta Compilateur` prend en entrée une description d'un assemblage de DSL de Métagrammaire et a pour productions un `Compilateur` et un environnement d'exécution (`VM Spécifique`)
- le `Compilateur` prend en entrée une `Métagrammaire` et produit un programme logique et des plugins de solveur (en fonction des principes utilisés dans la Métagrammaire)
- on énumère par backtracking toutes les exécutions du programme logique
- chaque exécution produit un ensemble de contraintes
- chaque ensemble de contraintes est soumis à un solveur (augmenté des plugins générés) et produit des solutions
- chaque solution est traduite et produit un élément dans la `Ressource Linguistique`

Les différents profils décrits dans la section précédente nécessitent des points d'entrée différents dans la chaîne de traitement.

Profil utilisateur. L'utilisateur doit pouvoir effectuer la génération de la ressource à partir de la description de celle-ci. Ce processus est identique à celui que proposait XMG-1. Si ce dernier est qualifié de *compilateur de métagrammaire*, la compilation n'est en fait qu'une des deux phases qu'il accomplit, puisqu'elle est suivie de celle de génération. La génération correspond

à l'exécution du code généré par la compilation, et est réalisée par les boîtes contenues dans le rectangle étiqueté `Exécuteur`. Dans la suite, nous appellerons le processus composé de la compilation de la métagrammaire et de l'exécution du code produit Méta Exécution. L'ensemble de modules dédiés à cette tâche (délimités par le rectangle étiqueté `Méta Exécuteur`) forme le point d'entrée de l'utilisateur final, prenant en entrée la métagrammaire qu'il a conçue et générant la ressource linguistique.

Le compilateur de métagrammaire prend en entrée une métagrammaire et génère du code exécutable. Ce code est ensuite exécuté durant la phase de génération, utilisant une machine virtuelle spécifique. Cette dernière est la machine virtuelle Prolog (`VM de Base`, puisque les DSL que nous assemblons utilisent le mécanisme d'unification) enrichie pour les besoins spécifiques au compilateur. Enfin, la sortie est la ressource linguistique, produite par cette phase de génération.

Profil contributeur. Le contributeur (ou assembleur) vise quant à lui la création de ce Méta Exécuteur pour une adaptation précise à ses besoins. Cette tâche, correspondant à l'étape de Méta Compilation, prend en entrée une description, celle du Méta Exécuteur (dont nous donnerons le principe dans le chapitre 6). Cette description manipule un ensemble de parties élémentaires de compilateur (`Bibliothèque de Briques`). Le travail du contributeur est d'assembler les briques au travers d'un unique fichier de spécification.

Profil programmeur. La méta compilation produit deux choses : un compilateur adapté à un langage dédié et les modules supplémentaires permettant d'enrichir la machine virtuelle pour que toutes les instructions de ce compilateur soient supportées. Le programmeur intervient dans l'élaboration de nouvelles briques, permettant ainsi aux compilateurs assemblés de traiter de nouveaux langages dédiés, générant de nouvelles instructions.

Les phases de méta compilation et de méta exécution sont indépendantes : il n'est pas nécessaire de réaliser une méta compilation à chaque changement dans la métagrammaire (hormis en cas de changement de langage de description par exemple).

5.5 Modularité linguistique

Le premier niveau de modularité proposé est le niveau applicatif, c'est à dire celui proposé par le langage métagrammatical. Les concepts centraux de l'approche métagrammaticale proposée par XMG ont été pensés en cohérence avec ceci.

La notion de modularité est souvent associé à celle de réutilisation. Dans tout langage adapté à la programmation modulaire, il existe un moyen de diviser un programme en parties élémentaire, réutilisables dans d'autres travaux. Dans cette section, nous présentons les aspects modulaires inclus dans l'approche implémentée par XMG.

Abstractions. Les classes, donnant un moyen d'abstraction sur les ressources à décrire, sont la première illustration de l'idée de réutilisation donnée par XMG. Pour réaliser ces abstractions, nous utilisons le langage proposé par la première version du compilateur (voir 3.3.4). Le découpage de la description métagrammaticale en classes est en général guidée par l'intuition linguistique. Une méthodologie pour accomplir cette tâche de découpage est proposée par [Crabbé, 2005].

Le fragment d'arbre suivant, représentant la réalisation canonique du sujet, est un exemple d'abstraction réutilisable :

```
class subjCan
export S
declare ?W ?N ?M ?S ?NN ?NV
{
  <syn>{
    node ?S(color=white)[cat = s]{
      node ?NN(color=red, mark=subst)
             [cat=n, top=[idx=?W, num=?N, pers=?M]]
      node ?NV(color=white)
             [cat=v,top=[num=?N, pers=?M]]
    }
  }*=[suj = ?W]
}
```

Au même titre, les inclusions de fichiers donnent un moyen de partager des fragments plus importants de ressources, éventuellement entre plusieurs métagrammaires. Un ensemble de déclarations de type, par exemple, peut se révéler utile dans différents travaux.

Composabilité. Afin de permettre la composabilité, il est nécessaire de pouvoir définir des modules, mais également de pouvoir les assembler. Les opérateurs de conjonction et de disjonction, rendus disponibles par le langage de contrôle, remplissent ce rôle, et permettent d'exprimer, à l'intérieur des classes et des dimensions, des assemblages de blocs et des alternatives.

La classe suivante présente un assemblage de blocs élémentaires offrant une alternative dans la réalisation du sujet :

```
class transitifDirect
declare
        ?X ?Y
{
        ?X = subjCan[];
        { ?Y = objCan[] | ?Y = objRel };
        activeMorph[] ;
        ?X.S = ?Y.S
}
```

Dimensions. Lorsqu'une ressource utilise plusieurs niveaux de représentation linguistique, la gestion des interactions entre ces niveaux constitue un point critique. Les dimensions donnent un moyen de séparer les niveaux de représentation, et de partager uniquement les informations souhaitées, de manière explicite.

L'exemple suivant présente une classe contribuant à la dimension sémantique (**<sem>**), que l'on peut combiner avec le fragment décrit précédemment pour la dimension syntaxique (**<syn>**).

```
class BinaryRel[Pred]
declare
        !L !E ?X ?Y
```

```
{
  <sem>{
    !L:Pred(!E,?X,?Y)
  }*= [arg0=?X, arg1=?Y]
}
```

Solveurs. Durant la phase de génération, les dimensions accumulent des descriptions. Ces dernières constituent un ensemble de contraintes décrivant un ensemble de solutions. Les problèmes sous contraintes ainsi formés doivent être résolus par des modules assurant que les modèles sont bien formés. Le solveur de XMG-1 s'assure par exemple que toutes les solutions pour la dimension **<syn>** forment chacune un arbre. On peut adapter une ressource à une nouvelle théorie linguistique par la sélection de nouvelles contraintes de bonne formation sur les structures produites. Pour cette raison, le choix du solveur à appliquer sur chaque dimension doit pouvoir être modifiable et configurable.

Les principes, que l'utilisateur peut choisir d'activer ou non par une simple déclaration dans la métagrammaire, donnent quant à eux une flexibilité dans les méthodologies de développement. Ainsi, pour les grammaires d'arbres adjoints, l'utilisation ou non des couleurs permet de baser la description sur des intuitions linguistiques différentes (en utilisant la notion de besoins et de ressources).

5.6 Modularité d'assemblage

Notre approche propose d'assembler le méta Exécuteur à partir de fragments. En procédant de la sorte, il n'est pas seulement plus simple d'apporter de nouvelles fonctionnalités, mais également de composer les fonctionnalités déjà disponibles. Un même fragment peut intervenir dans plusieurs compilateurs distincts (par exemple, si il permet de manipuler des structures communes à plusieurs approches), mais également au sein du même Méta Exécuteur.

Par exemple, le fragment contenant les opérateurs logiques pour combiner des contributions pourrait être réutilisé partout où il est nécessaire, en particulier à l'intérieur des dimensions.

Dans les trois chapitres suivants, nous débuterons par la proposition d'une méthode pour le découpage d'un langage de description. Puis nous expliquerons comment l'intégralité des phases de compilation pour ce langage peuvent également être découpées. Nous ferons de même concernant les phases de la génération dans le chapitre suivant.

Si l'on se base sur l'architecture présentée dans le schéma 5.1, atteindre la modularité est une tâche relativement complexe, puisque le seul point qui semble commun à tout Méta Exécuteur est la machine virtuelle de base (a priori, la machine virtuelle Prolog).

Adaptation. La principale limite de l'approche métagrammaticale de XMG jusqu'ici était que même si son langage permettait une certaine flexibilité, l'adapter à de nouvelles théories linguistiques ou de nouveaux niveaux de représentation était une tâche complexe. Programmer un nouveau Méta Exécuteur, ou même adapter l'existant, n'est pas une solution acceptable, car elle demande un temps conséquent et une expertise suffisante sur les différentes étapes de la compilation.

Un nouvelle couche d'abstraction est nécessaire, cette fois ci au dessus du Méta Exécuteur. Le but est de pouvoir composer un outil dédié à une tâche précise à partir de parties élémentaires.

Créer une des ces parties élémentaires est équivalent à rendre possible la compilation d'un fragment de langage métagrammatical (et l'exécution du code généré) . Cette manière de contribuer à l'évolution du Méta Exécuteur peut être mise en parallèle avec le phénomène de communautés contribuant aux langages de programmation. La création du support pour la compilation d'un nouveau niveau de description linguistique peut ainsi être vue comme l'ajout d'une librairie, par exemple de calcul scientifique.

Cette méthode de développement a pour avantage de laisser le compilateur évoluer dans les directions souhaitées par la communauté de ses utilisateurs.

5.7 Modularité du compilateur et de l'exécuteur

5.7.1 Compilateur

Le processus de compilation métagrammaticale est composé d'un ensemble de phases, manipulant différents langages. L'architecture que nous adoptons suit le schéma suivant :

FIGURE 5.2 — L'architecture du compilateur

Le langage utilisé en entrée de cette chaîne est le langage de description utilisé dans la métagrammaire. En sortie de l'analyse lexicale et syntaxique (tokenizer puis parser), un arbre de syntaxe abstraite de la métagrammaire a été construit. L'unfolder transforme cet arbre en instructions, décrites dans un langage minimal que l'on appellera langage noyau. Enfin, le générateur réalise l'implémentation de ces instructions dans le langage cible.

Un compilateur doit fournir les règles pour passer de chacun de ces langages au suivant. Diviser un compilateur en modules signifie répartir ces règles de manière à ce que chaque ensemble créé puisse être utilisé indépendamment.

5.7.2 Exécuteur

Comme la phase de compilation, celle de génération se compose d'un ensemble d'étapes :

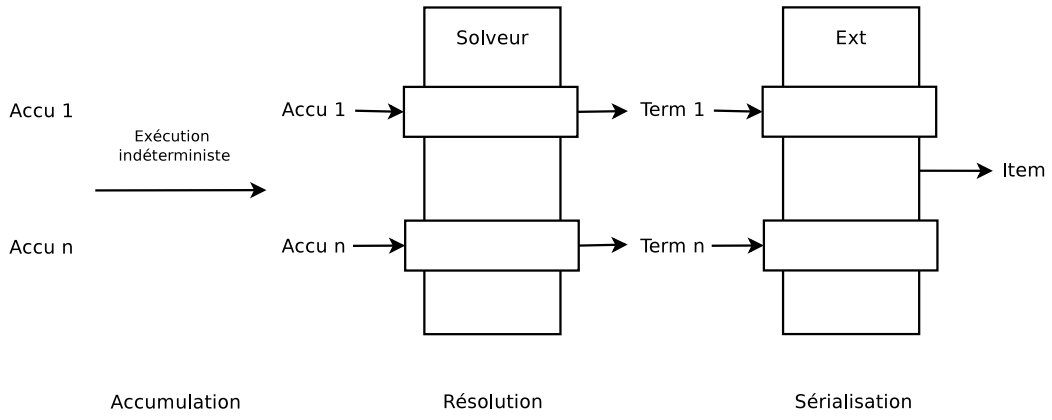


FIGURE 5.3 — L'architecture de l'exécuteur

Le codage des instructions qui sont produites par le compilateur est une tâche que l'on souhaiterait automatiser au maximum, en réutilisant celles pour lequel le support existe déjà. L'instruction d'unification est un exemple pour lequel la réutilisation est fréquente, puisque le mécanisme intervient dans de nombreuses situations.

L'utilisation de différentes dimensions, par exemple correspondant à différents niveaux de représentation, implique la manipulation d'objets de types différents. Les machines virtuelles utilisées pendant la génération des structures doivent être spécifiques aux objets manipulés. Ceci est fait au moyen de librairies spécifiques.

Chacune des dimensions possède non seulement son accumulateur dédié pendant l'exécution du code, mais également un ou plusieurs solveurs s'assurant de la bonne formation des structures produites.

Enfin, chaque solveur doit être associé à un module effectuant la conversion de ses termes produits dans un format de sortie du Méta Exécuteur.

5.8 Conclusion

Dans ce chapitre, nous avons repéré les différents niveaux où la modularité pouvait être exploitée dans l'utilisation et le développement d'un Méta Exécuteur tel que XMG. Dans le chapitre suivant, nous donnerons une méthode pour l'assemblage de langage dédiés, sur lequel nous nous appuierons dans les chapitres qui le suivent pour l'implémentation d'une solution modulaire.

Assemblage modulaire d'un langage dédié

6

Sommaire

6.1 Introduction **77**
6.2 Définir un langage dédié **77**
6.3 Enrichir un langage **78**
6.4 Des briques de langage **80**
6.4.1 Définir des briques 80
6.4.2 Connecter des briques 81
6.4.3 Assemblage d'un DSL 84
6.5 Conclusion **85**

6.1 Introduction

Dans ce chapitre, nous nous intéressons au concept de langage, au sens de langage de programmation. En particulier, les langages que nous visons ici sont des langages dédiés (DSL) adaptés à la description de données langagières. Les langages de ce type permettent notamment la description de structures données, dans notre cas des structures linguistiques, c'est sur cet aspect que nous nous focaliserons.

Pour décrire la syntaxe d'un tel DSL, on supposera qu'une grammaire hors contexte dispose d'une expressivité suffisante. Nous proposons ici de construire une grammaire hors contexte en la décomposant en fragments élémentaires réutilisables. Dans les DSL que nous visons, certaines structures sont définies inductivement, comme les structures de traits récursives. Il est important que notre approche permette de décrire ce type de structures.

Une fois cette approche modulaire définie pour les langages de description, l'objectif est de montrer que l'on peut exploiter l'approche davantage et assembler ainsi l'intégralité d'une chaîne de compilation.

6.2 Définir un langage dédié

Commençons par choisir un exemple de structure à décrire. Débutons par une structure simple (une structure arborescente que l'on pourrait trouver dans une grammaire d'arbres adjoints par

exemple), que nous enrichirons au fur et à mesure :

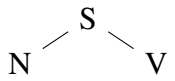


FIGURE 6.1 — Un exemple de fragment d'arbre

La description d'une telle structure peut être donnée de différentes façons. L'une d'entre elle, similaire à celle donnée dans [Rogers et Vijay-Shanker, 1994], est d'énumérer tous les composants, c'est à dire les noeuds, et de donner des contraintes sur les positions de ces composants. On peut ainsi donner cette description de la structure précédente :

$$\textbf{node}\ S\ \wedge\ \textbf{node}\ N\ \wedge\ \textbf{node}\ V\ \wedge\ (S \rightarrow N)\ \wedge\ (S \rightarrow V) \wedge (N \gg V)$$

où **node** S correspond à la déclaration d'un noeud nommé S, $\rightarrow$ représente la dominance immédiate d'un noeud sur un autre (le premier noeud est père du second), et $\gg$ la précédence immédiate (le second noeud est le frère immédiatement à droite du premier).

Pour ce DSL, on peut donner la grammaire hors contexte suivante :

$$\begin{array}{rcl} \mathit{Desc} & ::= & \mathit{Stmt} \\ & | & \mathit{Stmt}\ \wedge\ \mathit{Desc} \\ \mathit{Stmt} & ::= & \textbf{node}\ \mathit{id} \\ & | & \mathit{id} \rightarrow \mathit{id} \\ & | & \mathit{id} \gg \mathit{id} \end{array}$$

Une description (reconnue par le non terminal Desc) est une suite d'énoncés (Stmt) séparés par le symbole $\wedge$. Un énoncé est une déclaration de noeud ou une relation de précédence ($\gg$) ou de dominance ($\rightarrow$) entre deux noeuds.

6.3 Enrichir un langage

Si l'on souhaite décorer les structures d'arbres vues jusqu'ici avec de l'information linguistique, on peut vouloir associer à certains noeuds un ensemble de traits morphosyntaxiques, contenus dans une structure de traits.

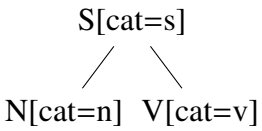


FIGURE 6.2 — Un exemple de fragment d'arbre décoré par des structures de traits

Ceci peut être exprimé de la façon suivante :

$$\begin{array}{l} \textbf{node}\ S\ \texttt{[cat=s]}\ \wedge\ \textbf{node}\ N\ \texttt{[cat=n]}\ \wedge\ \textbf{node}\ V\ \texttt{[cat=v]} \\ \wedge\quad (S \rightarrow N)\ \wedge\ (S \rightarrow V)\ \wedge\ (N \gg V) \end{array}$$

où **node** S [cat=s] signifie qu'au noeud S est associé une structure contenant l'unique trait cat=s. La grammaire hors contexte reconnaissant le langage doit donc être modifiée pour intégrer les descriptions de structures de traits :

$$
\begin{array}{rcl}
\mathit{Desc} & ::= & \mathit{Stmt} \\
 & | & \mathit{Stmt} \wedge \mathit{Desc} \\
\mathit{Stmt} & ::= & \textbf{node}\ \mathit{id} \\
 & | & \textbf{node}\ \mathit{id}\ \mathit{AVM} \\
 & | & \mathit{id} \rightarrow \mathit{id} \\
 & | & \mathit{id} \gg \mathit{id} \\
\mathit{AVM} & ::= & [\mathit{Feats}] \\
\mathit{Feats} & ::= & \mathit{Feat} \\
 & | & \mathit{Feat}, \mathit{Feats} \\
\mathit{Feat} & ::= & \mathit{id} = \mathit{id}
\end{array}
$$

FIGURE 6.3 — Une grammaire hors contexte pour des descriptions d'arbres décorées par des structures de traits

Les règles pour les non terminaux *Desc* et *Stmt* restent inchangées, à l'exception de l'ajout d'une règle. Celle-ci réécrit un littéral en un noeud associé à une structure de traits, représentée par le non terminal *AVM*. Ce non terminal se réécrit en une suite de traits (non terminal *Feat*) entre crochets.

Éviter la redondance. Les structures de traits sont des éléments intervenant dans de nombreux formalismes grammaticaux. Ces règles décrivant les structures de traits devront donc, pour chacun de ces formalismes pour lesquels on voudrait créer un langage de description, être ajoutées à la grammaire. Une forme de redondance apparaît donc dans ces langages de description, dans la mesure où un certain nombre de règles apparaissent de multiples fois. Dans notre cas, les règles en question sont celles concernant les non terminaux propres aux structures de traits, à savoir *AVM*, *Feats* et *Feat*.

La redondance s'illustre aussi de la manière suivante: les noeuds de la structure arborescente de la figure 6.1 pourraient être décorés d'une autre manière. On pourrait par exemple vouloir colorer les noeuds.

Le langage permettant la description de ce nouveau type de structures ne diffère du précédent qu'au niveau de la représentation des décoration de noeuds. Pour contourner cette redondance, nous proposons de découper les langages de descriptions en sous-langages. Ces sous-langages, que l'on appellera dans la suite *briques de langage*, sont des unités réutilisables et assemblables. Montrons maintenant comment découper un langage en parties élémentaires, puis comment assembler ces parties.

6.4 Des briques de langage

6.4.1 Définir des briques

La notion de conjonction de descriptions est indépendante du type de description, on peut donc séparer la règle concernant la conjonction de celles concernant la description d'arbres. De même, comme l'exemple précédent le montre, les structures de traits devraient être traitées indépendamment du langage de description d'arbres, même si elles peuvent intervenir à l'intérieur de ce dernier.

La grammaire proposée dans la figure 6.3 peut donc être vue comme l'assemblage du langage exprimant la conjonction, de celui permettant la description d'arbres, et du langage de structures de traits. Il reste à voir comment est composée chacune de ces briques, et surtout comment ces briques doivent interagir pour former le langage final.

Brique de conjonction:

$$\begin{array}{rcl} \mathit{Desc} & ::= & \mathit{Stmt} \\ & | & \mathit{Stmt} \ \wedge \ \mathit{Desc} \end{array}$$

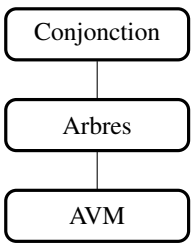
Brique de description d'arbres:

$$\begin{array}{rcl} \mathit{Stmt} & ::= & \textbf{node}\ \mathit{id} \\ & | & \textbf{node}\ \mathit{id}\ \mathit{AVM} \\ & | & \mathit{id} \rightarrow \mathit{id} \\ & | & \mathit{id} \gg \mathit{id} \end{array}$$

Brique de structures de traits:

$$\begin{array}{rcl} \mathit{AVM} & ::= & [\mathit{Feats}] \\ \mathit{Feats} & ::= & \mathit{Feat} \\ & | & \mathit{Feat}, \mathit{Feats} \\ \mathit{Feat} & ::= & \mathit{id} = \mathit{id} \end{array}$$

On souhaite recréer le langage initial en assemblant ces trois langages : en se basant sur le découpage proposé et la grammaire initiale, les liens connectant les langages peuvent être représentés comme suit.



6.4.2 Connecter des briques

Assemblage simple. Il s'agit maintenant de savoir comment spécifier les connexions. Une connexion sera vue comme le paramétrage d'une brique de langage par une autre, elle doit donc être orientée. Lorsqu'il existe une connexion allant d'une brique A à une brique B, un non-terminal de la grammaire de B doit être accessible par les règles de A.

Pour ce faire, l'un des non terminaux de la brique A doit jouer un rôle spécial : il ne peut apparaître que dans la partie droite des règles de A et se réécrit en un non terminal de la brique à laquelle il est connecté, en l'occurrence B. Ces non terminaux spéciaux sont appelés non terminaux externes. Un non terminal externe est un non terminal pour lequel la brique ne possède pas de règle de production, cette règle devant être fournie par une connexion à une autre brique. L'identifiant d'un non terminal externe débute par convention par un tiret bas et une majuscule.

Nous illustrons cette notion de connexion sur la figure 6.4, dans laquelle chaque boîte représente un ensemble de règles et chaque connexion représente une règle supplémentaire, consistant à réécrire un non terminal externe en un non terminal provenant d'une autre boîte.

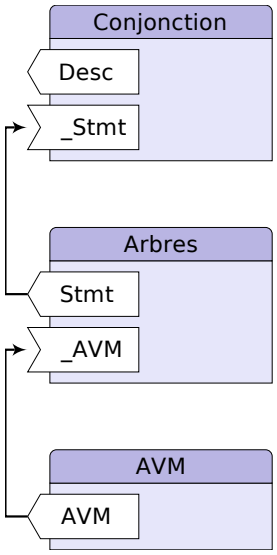


FIGURE 6.4 — Connexion de briques au moyen de non-terminaux externes

Cette manière de connecter les briques permet différentes utilisations d'une même brique, au moyen de différentes instances de cette brique, dont les paramètres, soit les connexions, peuvent varier. Par exemple, on peut considérer deux instances du langage d'arbre, l'une paramétrée par le langage d'AVM, l'autre par un langage permettant de décorer les noeuds avec des couleurs. Cet assemblage est illustré dans la figure 6.5.

La création d'une connexion doit être vue comme l'ajout d'une règle de réécriture du non-terminal externe en l'axiome de la brique connectée. Pour cette raison, le nombre de connexions d'un non-terminal externe n'est pas limité à une : si aucune connexion n'est effectuée, la règle en question n'est pas créée, et si les connexions sont multiples, la partie droite de la règle présente plusieurs alternatives. Les deux règles ajoutées par les connexions de la figure 6.4 sont :

$$\begin{array}{rcl} _AVM & ::= & AVM \\ _Stmt & ::= & Stmt \end{array}$$

Assemblage récursif. La brique de structures de traits décrite précédemment permet de manipuler des éléments syntaxiques très élémentaires, en l'occurrence uniquement des traits associés à des identifiants. On peut permettre d'associer des traits à des éléments plus complexes en connectant cette brique à différentes briques représentants les types de valeurs possibles pour les traits. Ici par exemple, on connecte la brique AVM à une brique Value. Celle-ci fournit les règles pour reconnaître les valeurs de base, ainsi qu'un non terminal externe (Else) pour des valeurs moins standard.

Brique *AVM*:

$$\begin{array}{rcl} AVM & ::= & [Feats] \\ Feats & ::= & Feat \\ & | & Feat, Feats \\ Feat & ::= & id = _Value \end{array}$$

Brique Value:

$$Value \quad ::= \quad id \quad | \quad \texttt{bool} \quad | \quad \texttt{int} \quad | \quad \texttt{string} \quad | \quad _Else$$

Si l'on choisit de connecter le non terminal externe de la brique Value (*_Else*) à la brique d'AVM déjà utilisée, on donne la possibilité de décrire des structures de traits récursives. La figure 6.6 illustre cet assemblage, créant les règles suivantes :

$$\begin{array}{rcl} _Value & ::= & Value \\ _Else & ::= & AVM \end{array}$$

Un non terminal externe peut également être laissé déconnecté. Dans le cas de la brique Value, ceci mène à un langage décrivant uniquement les valeurs des types de base.

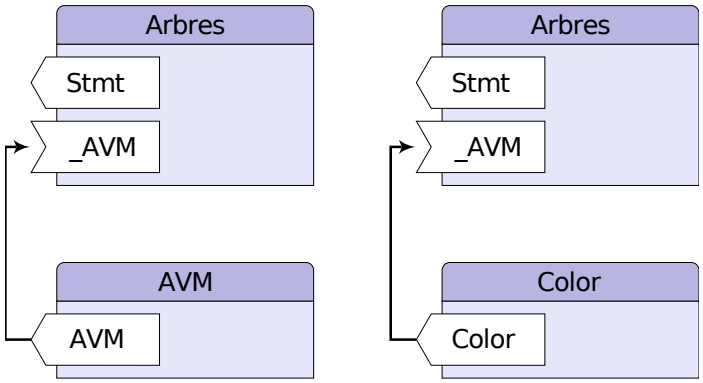


Figure 6.5 — Un assemblage utilisant plusieurs instances d'une même boîte

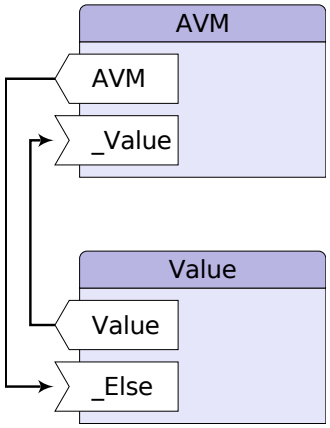


FIGURE 6.6 — Connexion de briques pour des structures récursives

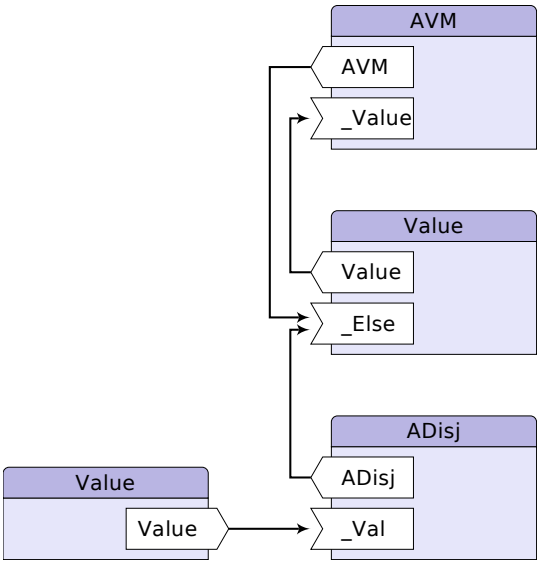


FIGURE 6.7 — Assemblage de briques pour la description de structures de traits intégrant des disjonctions atomiques

Instanciations multiples d'une même brique. Enrichissons à nouveau le langage de structures de traits, en autorisant un nouveau type de valeurs : la disjonction atomique, soit l'alternative entre plusieurs valeur atomiques. On peut par exemple écrire `cat=@{n,cl}` pour exprimer le fait que la catégorie syntaxique (`cat`) peut être `n` ou `cl`. Pour exprimer ces structures, nous proposons les briques suivante, dont les connexions sont illustrées par la figure 6.7 :

Brique *AVM*:

$$
\begin{array}{rcl}
AVM & ::= & [Feats] \\
Feats & ::= & Feat \\
 & | & Feat, Feats \\
Feat & ::= & id = _Value
\end{array}
$$

Brique *Value*:

$$Value \quad ::= \quad id \quad | \quad \texttt{bool} \quad | \quad \texttt{int} \quad | \quad \texttt{string} \quad | \quad _Else$$

Brique *ADisj*:

$$\begin{array}{rcl} ADisj & ::= & \texttt{@\{}\ Vals\ \texttt{\}} \\ Vals & ::= & _Val \quad | \quad _Val, Vals \end{array}$$

Les connexions de la figure 6.7 créent les nouvelles règles suivantes :

$$\begin{array}{rcl} _Value & ::= & Value \\ _Else & ::= & AVM \quad | \quad ADisj \\ _Else & ::= & AVM \quad | \quad ADisj \\ _Val & ::= & Value1 \end{array}$$

où *Value1* correspond à une nouvelle instance de la brique *Value*.

Ici, les disjonctions atomiques manipulées à l'intérieur des structures de traits sont simples, c'est à dire qu'elles ne permettent la disjonction qu'entre des valeurs des types fournis par la brique Value. Pour ce faire, une autre instance de la brique en question est nécessaire, le non terminal externe de cette nouvelle brique ne devant être connecté ni à la brique *AVM*, ni à la brique *ADisj*.

La modularité offerte par les briques de langage permet de composer des langages intégrant de nouvelles structures en connectant de nouvelles briques aux existantes. Au même titre que la brique de disjonction atomique, des structures telles que des formules logiques pourraient être ajoutés aux types de valeurs possibles pour les traits.

6.4.3 Assemblage d'un DSL

En utilisant cette approche modulaire, nous pouvons automatiser des tâches telles que l'analyse syntaxique, à partir de ces grammaires hors contexte. Néanmoins, dans le cadre de la création d'un compilateur, la reconnaissance du langage ne représente qu'une partie du travail.

La méthode que nous proposons pour instancier et connecter les briques en pratique repose sur un fichier utilisant la syntaxe YAML (YAML Ain't Markup Language), un format de représentation de données, qui se distingue de langages comme XML ou JSON par sa lisibilité.

Pour notre dernier exemple, le fichier d'assemblage aurait la structure suivante :

```
avm :
  _Value : value
value :
  _Else : avm adisj
adisj :
  _Val : value_1
```

où *avm*, *value*, $value_1$ et *adisj* correspondent aux boîtes présentées précédemment. Pour chacune de ces boîtes, on donne la liste des connexions. Par exemple `_Else : avm adisj` signifie que le non terminal externe *_Else* est connecté à la fois à l'axiome de la boîte *avm* et à celui de *adisj*.

La grammaire hors contexte générée par cet assemblage est la suivante :

$$
\begin{array}{rcl}
AVM & ::= & [Feats] \\
Feats & ::= & Feat \\
 & | & Feat, Feats \\
Feat & ::= & id = Value \\
Value & ::= & id \;\mid\; \mathbf{bool} \;\mid\; \mathbf{int} \;\mid\; \mathbf{string} \;\mid\; _Else \\
ADisj & ::= & \texttt{@\{}\ Vals\ \texttt{\}} \\
Vals & ::= & Value1 \;\mid\; Value1, Vals \\
Value1 & ::= & id \;\mid\; \mathbf{bool} \;\mid\; \mathbf{int} \;\mid\; \mathbf{string} \\
_Else & ::= & ADisj \;\mid\; AVM
\end{array}
$$

6.5 Conclusion

Nous avons proposé dans ce chapitre une méthode d'assemblage de langages dédiés, basée sur l'utilisation de parties élémentaires de langages, les briques. Ces unités permettent d'assembler un langage de description en donnant la liste des briques et les connexions à opérer entre elles.

Les chapitres suivants visent à montrer que l'ensemble de la chaîne de compilation peut être assemblée de la même manière, et que le découpage modulaire peut être basé sur les briques de langages que nous avons proposées dans ce chapitre. À partir d'un unique fichier de description, nous souhaitons assembler non seulement le langage dédié, mais également chacun des modules constituant la chaîne de traitement allant d'une métagrammaire décrite avec ce langage dédié à la ressource qu'elle décrit.

Assemblage modulaire d'un compilateur

7

Sommaire

7.1 Introduction **88**
7.1.1 Étapes de la compilation 88
7.1.2 Répartir les règles dans des briques 89
7.1.3 EDCG 90
7.1.4 Variables attribuées 91
7.2 Étape d'analyse **93**
7.2.1 Analyse syntaxique 93
7.2.2 Analyse lexicale 95
7.2.3 Connecter des briques 95
7.3 Étape de Typage **97**
7.3.1 Principe 97
7.3.2 Typage des données 98
7.3.3 Typage du langage de contrôle 100
7.3.4 Typage des classes 101
7.3.5 Typage des dimensions 102
7.3.6 Algorithme de typage 104
7.3.7 Implantation 106
7.4 Étape de dépliage **107**
7.4.1 Principe 107
7.4.2 Formalisation du dépliage 108
7.4.3 Exemples 111
7.4.4 Implantation 113
7.4.5 Extensibilité 113
7.5 Étape de génération **114**
7.5.1 Principe 114
7.5.2 Formalisation et implantation 115
7.5.3 Extensibilité 118
7.5.4 Code spécifique aux classes 119
7.6 Constructeurs de briques **121**

7.7 Point d'entrée de l'exécution **122**
7.7.1 Lancer l'évaluation du code généré 122
7.7.2 Définir les accumulateurs 123
7.7.3 Code généré 124
7.8 Exemple : XMG-1 . . . . . **125**
7.9 Conclusion **127**

7.1 Introduction

7.1.1 Étapes de la compilation

Comme nous l'avons vu au chapitre 5, la méta exécution se déroule en deux phases. Nous nous intéressons dans ce chapitre à la première de ces phases, appelée compilation, durant laquelle le code métagrammatical est compilé vers du code exécutable.

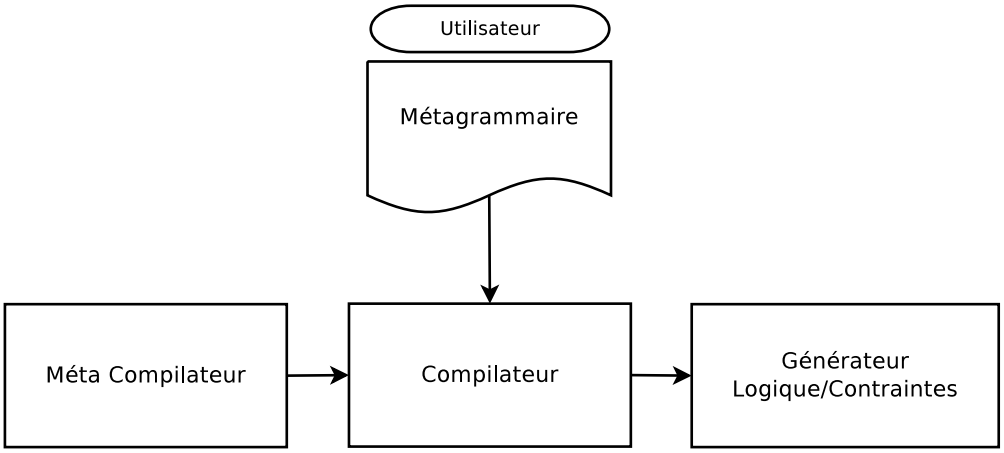


FIGURE 7.1 — Le compilateur

Le principe de la compilation est de passer d'une description, dans le langage concret assemblé de façon modulaire, à sa traduction en instructions exécutables. Cette traduction s'opère en plusieurs étapes successives. Les étapes de la compilation sont les suivantes :

Tokenizer ⟶ Parser ⟶ Type Checker ⟶ Unfolder ⟶ Code Generator

FIGURE 7.2 — L'architecture du compilateur

Tokenizer. L'analyse lexicale est une étape de segmentation du fichier d'entrée. L'idée est de reconnaître et de séparer dans le texte en entrée (la métagrammaire) tous les tokens (soit des couples catégorie-valeur sémantique).

Parser. La liste des tokens est ensuite donnée au parser pour une analyse syntaxique. Cette étape crée un arbre de syntaxe abstraite de la métagrammaire.

Type Checker. L'étape suivante est la vérification du bon typage à l'intérieur de cet arbre de syntaxe abstraite.

Unfolder. L'arbre est ensuite déplié par l'unfolder pour obtenir un jeu d'instructions, ces instructions appartenant à un langage noyau que nous présenterons plus tard.

Code Generator. La dernière étape de la compilation consiste à traduire les instructions du langage noyau vers du code exécutable.

7.1.2 Répartir les règles dans des briques

Nous avons vu précédemment que la grammaire hors contexte d'un langage dédié pouvait être assemblée à partir de briques, l'objectif de ce chapitre est de montrer que ces briques peuvent également apporter au compilateur la partie fonctionnelle associée à leur langage. En d'autres termes, nous souhaitons prouver que l'assemblage modulaire d'un compilateur peut être guidé par un assemblage de son langage.

Règles. Pour chacune des étapes de compilation, chaque brique de compilateur doit apporter sa contribution : les analyses lexicale et syntaxique dépendent naturellement du DSL, mais le reste de la chaîne de traitement également. L'unfolder prend par exemple en entrée l'arbre de syntaxe abstraite, ce qui le rend dépendant des modules qui le construisent. Le générateur de code doit également être capable de générer des instructions du langage cible adaptées aux structures manipulées.

Une brique apporte donc sa contribution à chacun des modules du compilateur. Cette contribution se compose de l'ensemble des règles de compilation pour les structures qu'elle manipule, soit par exemple les constructeurs utilisés lors de l'analyse syntaxique, ou les instructions générées. Chaque brique correspond à un module (au sens classique du mot dans les langages de programmation), et chaque structure manipulée par une brique appartient à son module. Ceci assure notamment l'évitement des conflits (par exemple entre deux constructeurs de même nom mais issus de deux briques différentes),

Langages. Pour résumer, une brique doit contenir le support pour l'analyse de son langage de description, et les langages successifs en lesquels il est transformé. À chaque étape de la compilation, ce support se constitue de l'ensemble des règles de transformation pour un langage. Le schéma suivant récapitule les étapes de compilation et les langages qui leurs sont associés.

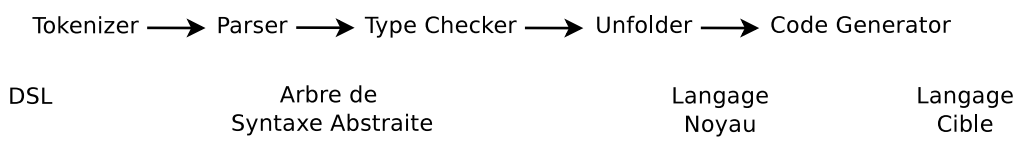


Figure 7.3 — L'architecture du compilateur et les langages manipulés

Dans ce chapitre, nous donnerons pour chacune des étapes les règles impliquées, et expliquerons comment ces règles peuvent être réparties dans les briques puis assemblées. Dans chaque

section, les règles seront dans un premier temps formalisées (en utilisant des notations fonctionnelles ou des règles d'inférence), puis nous donnerons l'implémentation réalisée pour ces règles (c'est à dire leur traduction en prédicat).

Avant de nous concentrer sur la modularisation des étapes de compilation, présentons deux technologies dont la connaissance est nécessaire pour la compréhension de l'approche proposée dans cette thèse. Ces mécanismes seront largement utilisées pour l'implémentation de la modularité.

7.1.3 EDCG

Dans un compilateur, des structures de données globales et modifiables sont nécessaires, notamment des tables modélisant le contexte, ou des listes dans lesquelles sont accumulées les instructions. Dans notre cas, ce besoin est problématique, puisque Prolog n'offre pas de structures de ce genre. La manière classique de répondre à ce besoin en programmation logique est d'utiliser des paires d'arguments supplémentaires dans les prédicats, l'un pour représenter la structure avant l'application du prédicat, et l'autre pour la représenter dans son nouvel état après son application.

Par exemple, considérons le prédicat suivant, qui permet une accumulation des éléments de la liste en premier argument dans une liste en troisième argument, sous condition qu'ils soient supérieurs à deux :

```
filtre([],Mem,Mem).
filtre([H|T],Mem,Res):- H>2, !, filtre(T,[H|Mem],Res).
filtre([_|T],Mem,Res):- filtre(T,Mem,Res).
```

Un exemple d'utilisation de ce prédicat est :

```
?- filtre([1,4,2,3],[],B).
B = [3,4].
```

Pour simuler un accumulateur dans un prédicat avec les structures de données prolog, on doit donc multiplier les arguments. Si l'on doit simuler un grand nombre d'accumulateurs, la gestion des arguments devient problématique (leur nombre complique le développement et la maintenance).

Pour rendre ces structures accessibles plus directement à l'intérieur des prédicats Prolog (et éviter une explosion du nombre de leurs paramètres), nous utilisons une librairie dédiée, inspirée des Extended DCG proposées par Andreas Kagedal, Peter Van Roy et Bruno Dumant [1] [Van Roy, 1990]. Les EDCG sont un outil des plus pertinents pour le développement d'un compilateur, puisqu'elles ont été utilisées dans le cadre du développement du compilateur Prolog Aquarius [Van Roy et Despain, 1992].

Comme leur nom l'indique, elles fournissent un sucre syntaxique semblable à celui des DCG présentées en 3.2.1. Cette librairie permet de déclarer des accumulateurs, de les associer à des prédicats, puis de déclencher à l'intérieur de ces prédicats des actions sur les accumulateurs. Là où les DCG permettaient d'alléger les notations pour deux paramètres (les listes restant à analyser avant et après l'application du prédicat), simulant ainsi un accumulateur, les EDCG font de même avec un nombre arbitraire d'accumulateurs.

1. http://www.info.ucl.ac.be/~pvr/edcg.html

La librairie *edcg*, que nous avons développée pour les besoins d'XMG, permet une utilisation transparente des accumulateurs, dispensant les prédicats des deux arguments supplémentaires pour chacun d'entre eux. On accède aux accumulateurs par leur identifiant, et on les manipule en leur appliquant les opérations définies pour eux. Par exemple, si on a déclaré l'accumulateur `acc` pour le prédicat `filtre/1`, on peut écrire à l'intérieur de ce dernier `acc::add(Add)`, signifiant l'ajout de l'élément `Add` à l'accumulation.

Si l'on met de côté la déclaration ainsi que l'initialisation de l'accumulateur, le prédicat précédent peut s'écrire :

```
filtre([]):--!.
filtre([H|T]):-- H>2, acc::add(H), !, filtre(T).
filtre([_|T]):-- filtre(T).
```

Les arguments supplémentaire apparaissent uniquement lors de leur initialisation, précédés du mot clé `with` :

```
test_filtre(Liste,ListeRes):--
    filtre(Liste) with (acc([],ListeRes)).
```

signifiant que le prédicat `test_filtre` appelle `filtre` avec un accumulateur `acc` étant initialement une liste vide, et dont la valeur finale est `ListeRes`.

Au final, l'exemple d'utilisation du prédicat utilisé comme point d'entrée (`test_filtre`) est le même que l'exemple sans EDCG, hormis le fait qu'il n'est plus nécessaire de donner l'argument correspondant à la valeur initiale :

```
?- test_filtre([1,4,2,3],B).
B = [3,4].
```

7.1.4 Variables attribuées

Prolog permet l'utilisation de structures de données dont on a une connaissance partielle, notamment les termes contenant des variables logiques. En revanche, les termes que l'on manipule ont une arité fixe, ce qui dans notre cas est une limite : nous souhaitons pouvoir manipuler des structures que l'on peut contraindre progressivement. C'est le cas des structures de traits, dont la taille augmente par unification. Il est donc nécessaire d'avoir un moyen de définir nos propres structures, et de conserver le mécanisme d'unifications pour celles ci.

L'objet Prolog représentant une structure de traits doit disposer d'un mécanisme d'unification dédié : l'unification de deux structures doit échouer si un même trait dans les deux structures a des valeurs différentes. Si l'unification réussit, le produit de l'unification est une structure de trait composée de l'union des traits des deux structures unifiées. Par exemple, l'unification des structures `[cat=s,num=sg]` et `[num=sg,gen=m]` produit `[cat=s,num=sg,gen=m]`. En revanche l'unification de `[cat=s,num=sg]` et `[num=pl,gen=m]` échoue du fait de l'incompatibilité entre les deux valeurs du trait `num`.

Pour cette raison, la brique de structures de traits inclut un module contenant une bibliothèque dédiée. Celle-ci repose sur le concept de variables attribuées [Holzbaur, 1992]. Les variables attribuées permettent d'associer à une variable Prolog un ensemble d'attributs, mais surtout de concevoir un algorithme d'unification dédié sur ces attributs lorsque deux variables attribuées de même type sont unifiées.

Une variable représentant une structure de trait se voit associée un attribut, cet attribut étant la liste des couples attribut-valeur composant la structure.

```
:- use_module(library(atts)).
:- use_module(library(rbtrees)).

:- attribute avmfeats/2.

verify_attributes(Var, Other, Goals) :-
    get_atts(Var, avmfeats(T1,U)), !,
    get_atts(Other, avmfeats(T2,U))
      ->
            rb_visit(T1,Pairs),
            unify_entries(T2,Pairs,T3),
            put_atts(Other, avmfeats(T3,U))
      ;     \+ attvar(Other),
            put_atts(Other, avmfeats(T1,U)))
    Goals = [].

verify_attributes(_, _, []).

unify_entries(T,[],T).
unify_entries(T1,[K-V0|L],T3) :-
        (rb_lookup(K,V1,T1) -> V0=V1, T1=T2;
                               rb_insert(T1,K,V0,T2)),
        unify_entries(T2,L,T3).

avm(X, L) :-
    var(L), !,
    get_atts(X, avmfeats(T,_)),
    rb_visit(T,L).

avm(X, L) :-
    list_to_rbtree(L,T),
    put_atts(Y, avmfeats(T,_)),
    X = Y.

attribute_goal(Var, avm(Var,L)) :-
    get_atts(Var, avmfeats(T,_)),
    rb_visit(T,L).
```

FIGURE 7.4 — Une bibliothèque de variables attribuées pour les structures de traits

En pratique, la création d'une bibliothèque pour un type de variables attribuées utilise la librairie `atts` de YAP. Celle-ci fournit deux prédicats pour modifier l'attribut d'une variable ou le consulter (`put_atts` et `get_atts`). Deux autres prédicats ont à être définis par l'utilisateur. Le premier est `verify_attributes`, qui est appelé pendant l'unification d'une variable, et déter-

mine le résultat de l'unification. C'est donc par ce prédicat qu'est réalisée la modification du mécanisme d'unification que nous souhaitons exploiter. Enfin `attribute_goal` convertit un attribut en but.

Une bibliothèque pour les structures de traits. L'exemple de la figure 7.4 montre la bibliothèque de variables attribuées dédiée aux structures de traits. Les attributs des variables sont des listes d'associations, converties en arbres bicolores (par le biais de la bibliothèque `rbtrees`) pour des accès plus rapides. Les prédicats `avm` permettent pour le premier d'accéder aux attributs d'une variable et pour le second de construire un attribut puis de l'associer à la variable. `verify_attributes`, déclenché lors de la tentative d'unification de **`Var`** et `Other`, consiste à récupérer les attributs de chacune des deux variables. Si `Other` est une variable attribuée du bon type (son attribut est reconnaissable par le constructeur `avmfeats`), on réalise l'unification ouverte des traits des deux structures, par l'intermédiaire de `unify_entries`. Dans le cas où `Other` est une variable libre, on lui associe directement l'attribut de **`Var`**.

Reproduire l'exemple de l'introduction de cette section en utilisant cette bibliothèque revient au code suivant :

```
?- avm(A,[cat-s,num-sg]),avm(B,[num-sg,gen-m]),A=B.

A = B,
avm(A,[cat-s,num-sg,gen-m]).
```

7.2 Étape d'analyse

7.2.1 Analyse syntaxique

Le but est de pouvoir construire un analyseur syntaxique pour un langage assemblé comme décrit dans le chapitre 6. Ce que l'on essaie de représenter est donc une grammaire hors contexte, soit un ensemble de règles de production. Une brique de langage est un fragment paramétrique de syntaxe concrète, dont la contribution au compilateur est un ensemble de règles de réécriture.

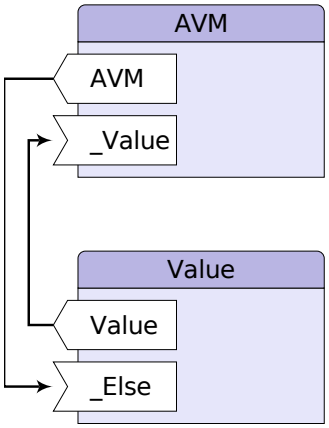
L'un des modules de compilation, le parser pour ce langage, soit les règles pour un analyseur LR, seront générées automatiquement à partir de la brique de langage. Une brique XMG contient un fichier représentant sa brique de langage, que l'on nomme `lang.def`.

Une brique de langage est décrite de la façon suivante : on définit les règles de réécriture, associant un non terminal non externe à un ensemble éventuellement vide de règles de réécriture. Une règle est composée d'une suite de symboles terminaux et non terminaux ainsi que d'une action sémantique :

```
NT : s_1 s_2 ... s_n { A1 } | ... ;
```

où `NT` est un non-terminal, les éléments `s` des symboles terminaux ou non terminaux, et `A1` une action sémantique.

Revenons à l'exemple suivant, issu de du chapitre 6.



La brique de langage AVM contient les règles suivantes :

```
%%
AVM : '[' (Feat // ',')* ']' {$$=avm:avm($2)};

Feat : id '=' _Value {$$=avm:feat($1,$3)};
%%
```

où `avm:avm` et `avm:feat` sont deux constructeurs dans l'espace de nom local à la brique `avm`. L'opérateur '*' désigne dans la brique de langage la séquence de zéro, un ou plusieurs symboles. L'application de cet opérateur à un symbole déclenche la génération de nouvelles règles de réécriture, accompagnées des actions sémantiques adéquates.

Les deux parties composant une règle de la brique ont des fonctions différentes : la première partie, soit la règle de réécriture, permet la reconnaissance du langage, alors que la deuxième, l'action sémantique, participe à la construction de la représentation sémantique. La raison pour laquelle une règle de réécriture n'est pas suffisante est que la représentation que nous souhaitons construire est une représentation sémantique et pas simplement un arbre de syntaxe abstraite.

Une action sémantique est une suite d'instructions, qui sont, dans le cas de notre compilateur, des instructions prolog. On peut à l'intérieur de ces instructions faire référence à la partie gauche de la règle par la variable $$, ainsi qu'à des symboles de la partie droite par les variables $n, où n est l'indice du symbole de la partie droite auquel on réfère.

L'évaluation des attributs utilisés dans ces règles étant faite par unification, il n'est pas nécessaire de différencier les attributs synthétisés (évalués à partir des attributs des fils du noeud syntaxique) des attributs hérités (évalués à partir des valeurs des attributs aux frères et au père du noeud).

Ici, par exemple

```
AVM : '[' (Feat // ',')* ']' {$$=avm:avm($2)};
```

signifie que la règle produit un terme construit par le constructeur `avm` (dans l'espace de nom `avm` local à la brique) ayant pour paramètres les traits analysés par les appels au non terminal `Feat`.

La brique de langage contribue au parser avec les règles pour sa grammaire. Une analyse LR est dans le cas présent suffisante à l'évaluation des attributs, puisque nous manipulons des structures contraintes, avec des variables logiques.

Dans la brique de structures de traits, les règles de syntaxe ne sont pas la seule contribution à effectuer : l'analyse lexicale doit également être adaptée pour reconnaître les tokens spécifiques à la brique.

7.2.2 Analyse lexicale

L'analyse lexicale est une tâche assez uniforme pour les langages de programmation. Ceci est dû au fait que les notions d'identifiant, d'entier, de booléen, de chaîne de caractères, sont largement partagés. En revanche, une partie des tokens à reconnaître est dépendante du langage, principalement au niveau des mots-clé et des signes de ponctuation. Par exemple, le signe '[' utilisé dans les exemples de cette section doit être reconnu comme un signe de ponctuation pour le langage défini.

Pour cette raison, l'analyseur lexical de XMG est extensible. Il fournit le support pour la reconnaissance des tokens de base (les entiers, les identifiants, etc.), et peut être enrichi pour la reconnaissance de mots-clé et signes de ponctuation propres à un nouveau langage.

Sur le même modèle que pour l'analyse syntaxique, pour lequel la contribution de chaque brique est un ensemble de règles, la contribution d'une brique à l'analyse lexicale est un ensemble de tokens. Chaque brique de langage doit donc apporter à l'analyseur lexical l'ensemble des signes de ponctuation et de mots clés qui lui sont spécifiques. On extrait alors automatiquement des règles syntaxiques de nouveaux tokens.

Observons les contributions à l'analyseur lexical pour un exemple de brique de langage:

```
%%
A : 'key' '{' B '}' {$$=key($3)};
B : int {$$=$1} | bool {$$=$1} | id {$$=$1};
%%
```

Pour que la règle `A` soit correctement traitée à l'analyse syntaxique, l'analyse lexicale doit reconnaître `key` comme un mot clé, et les accolades comme signes de ponctuation. Les autres terminaux (`int`, `bool` et `id`), présents dans la règle `B` correspondent à des types prédéfinis.

La chaîne `key { 2 }` est tokenizée en

```
[token(coord('file.mg',1,0), key),
 token(coord('file.mg',1,4), '{'),
 token(coord('file.mg',1,6), int(2)),
 token(coord('file.mg',1,8), '}')]
```

où chaque token de la liste a comme arguments une coordonnée et une valeur sémantique.

Cet analyseur lexical extensible est adaptable à la plupart des langages, dans la mesure où ils restent orienté vers le modèle initial proposé par XMG, c'est à dire manipulant les mêmes types de base, et utilisant le même type de délimiteurs de commentaires.

7.2.3 Connecter des briques

Pour créer un langage à partir de ces briques de langage, nous devons établir des connexions. Une connexion entre deux briques est la réécriture d'un non-terminal externe en un non-terminal de la brique connectée, ajouter une connexion correspond donc à considérer toutes les règles

des deux briques à connecter, et de leur ajouter une nouvelle règle de réécriture correspondant à la connexion.

Lorsque l'on instancie une brique, les non-terminaux externes sont associés à d'autres briques déjà instanciées. On paramètre donc la création d'une brique par l'ensemble de ses connexions.

La création d'un compilateur consiste en l'assemblage de briques. Le principe est de créer des instances des briques nécessaires, puis de les connecter en associant les non terminaux externes à des ensembles de briques. Le langage que nous proposons pour cet assemblage consiste simplement à instancier des briques, puis leurs connexions.

Comme annoncé précédemment, cet assemblage de langage de description se fait au moyen d'un unique fichier (`compiler.yaml`, utilisant la syntaxe YAML), rendant cette tâche simple et intuitive. L'hypothèse que nous formulons maintenant est que ce fichier de description peut non seulement permettre d'assembler l'analyseur syntaxique pour le langage assemblé, mais que l'information qu'il contient est suffisante pour assembler l'intégralité des étapes suivantes de compilation.

À chaque brique instanciée pour le compilateur, on associe l'ensemble de ses connexions. Une connexion est l'association d'un non terminal externe de la brique à une séquence de briques.

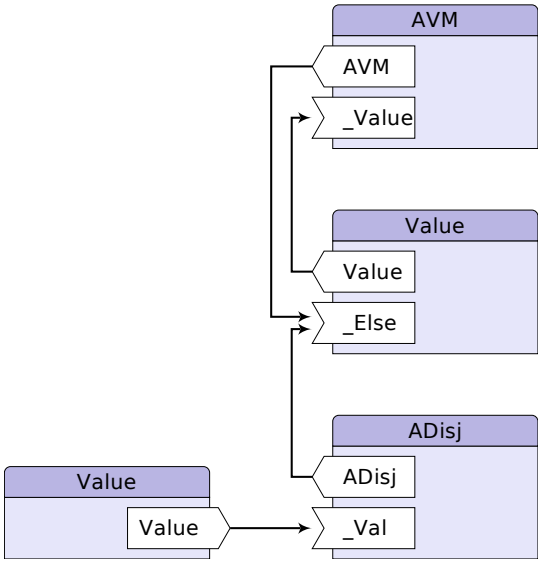
```
b :
  _C1 : b1
  ...
  _Cn : bn
```

où $b, b1 \ldots bn$ sont des briques, et $_C_1 \ldots _C_n$ les non-terminaux externes de la brique b.

Par convention, tout identifiant id débutant par une minuscule est l'instanciation d'une brique ayant pour identifiant id. Pour manipuler deux instances différentes d'une même brique, on pourra utiliser l'identifiant id suivi de _ et d'un suffixe. De même, tout identifiant débutant par une majuscule est nécessairement le non terminal externe désigné par cet identifiant dans la brique courante

Par défaut, une connexion se fait sur l'axiome d'une brique. Pour une plus grande flexibilité dans les connexions, on peut accéder à un autre non terminal de la brique avec l'opérateur '.' .

Revenons à l'exemple suivant :



La connexion des briques est réalisée de la façon suivante :

```
avm :
  _Value : value
value :
  _Else : avm adisj
adisj :
  _Val : value_adisj
```

L'indépendance des symboles entre les briques est assurée par des espaces de noms distincts : chaque instance d'une brique ne contient que des symboles préfixés par un identifiant unique correspondant à l'instance. Seules les connexions peuvent établir des liens entre deux symboles issus de deux espaces de noms différents.

Ainsi, la connexion

```
avm :
  _Value : value
```

crée une règle pour le parser, que l'on pourrait écrire

```
avm._Value : value.Value { $$ = $1 };
```

où le non terminal `Value` est l'axiome du langage de la brique `value`.

Jusqu'ici, nous ne nous sommes intéressés qu'à deux étapes de la compilation, à savoir les analyses lexicale et syntaxique. Si la création des analyseurs lexical et syntaxique est maintenant automatisée, l'ensemble des autres étapes de la compilation restent à traiter. Il est important pour la validation de la méthodologie que les autres étapes puissent suivre le découpage proposé, même si celui-ci ne se base a priori que sur la syntaxe .

7.3 Étape de Typage

7.3.1 Principe

Une fois l'arbre de syntaxe abstraite construit par l'analyseur syntaxique, le compilateur procède à une vérification des types de la métagrammaire.

L'unification étant le mécanisme central du processus d'exécution, il est important de pouvoir faire la différence entre une unification échouant à cause de valeurs incompatibles et une unification échouant pour cause de types de données incompatibles. L'utilisation des variables, leur héritage, rendent difficile la prévision de ce phénomène par le développeur de la ressource.

Ces échecs peuvent être détectés si toutes les données décrites sont typées. Par exemple, l'unification d'une structure propre à une dimension (un noeud syntaxique, un prédicat sémantique) avec une valeur doit être considérée comme une erreur dans la description.

Tous les constructeurs apparaissant dans l'arbre de syntaxe abstraite sont associés à une des briques utilisées dans l'assemblage du compilateur. Le principe est le suivant : chaque brique doit apporter les règles de typage pour l'ensemble de ses constructeurs.

Si toutes les règles de typage sont apportées par des briques, on peut néanmoins les classer en deux catégories. La première est celle des règles de typage pour les types de base, soit ceux

correspondants aux concepts de la métagrammaire, comme les types des classes, ou ceux des dimensions. La seconde catégorie est celle des règles de typages additionnelles propres à de nouveaux langages de description, et qui sont des contributions propres aux briques liées à ces langages.

Les règles de typage que nous présenterons dans cette section font intervenir des contextes et des jugements. Les contextes que nous utiliserons sont notés $\mathcal{T}$ et $\mathcal{K}$. Le contexte $\mathcal{T}$ affecte des types à des symboles, il contient les types des constantes et des identifiants du langage, des variables et des dimensions. Le contexte $\mathcal{K}$ est utilisé pour le typage des structures de traits, que nous détaillerons en 7.3.2, il affecte des espèces à des variables de type. Nous écrirons $\mathcal{K},\mathcal{T} \rhd e : \tau$ pour le jugement exprimant que l'expression e est de type τ dans le contexte composé par $\mathcal{K}$ et $\mathcal{T}$. De même, $\mathcal{K} \vdash \alpha :: k$ indique que dans le contexte $\mathcal{K}$, la variable de type α a l'espèce k.

7.3.2 Typage des données

Dans un premier temps, voyons comment sont typées les données décrites dans la métagrammaire, à savoir les constantes, les identifiants, et les structures de traits.

Typage des constantes. Les déclarations de types énumérés enrichissent le contexte avec une affectation de type pour chacune des constantes déclarées.

Par exemple :

```
type CAT = {s,n,v}
```

implique les propriétés suivantes :

$$\texttt{s:CAT} \in \mathcal{T} \quad \texttt{n:CAT} \in \mathcal{T} \quad \texttt{v:CAT} \in \mathcal{T}$$

Typage des identifiants. Les types des identifiants déclarés dans la métagrammaire sont inférés durant la étape de typage. Le contexte $\mathcal{T}$ associe à chaque variable du langage un type. La règle de typage pour les variables consiste donc à consulter le contexte.

$$\text{SYM}_\text{v} \ \frac{\texttt{v:t} \in \mathcal{T}}{\mathcal{K},\mathcal{T} \rhd \texttt{v:t}}$$

Typage des structures de traits. Concernant les structures de traits, le calcul des types s'inspire de celui présenté dans [Ohori, 1995] pour les enregistrements dans un langage fonctionnel. La difficulté réside dans la nécessité de polymorphisme dans les fonctions sur ces structures, qui s'illustre par exemple sur l'opérateur d'accès aux valeurs ".".

`?X=?A.cat` est une expression valide pour toute structure de trait `?A` contenant un trait cat. Si l'on considère les deux types de structures de traits `[cat:CAT]` et `[cat:int]`, le type à inférer pour `?X` diffère (`CAT` dans un cas, `int` dans l'autre).

Au fil des instructions, le type de la structure peut recevoir de nouvelles contraintes, comme le montre l'exemple suivant :

```
?E=[];
?X=?E.f1;
?Y=?E.f2
```

Nous n'avons initialement aucune information sur le type de `?E`, mis à part son unicité. La première instruction contraint le type de `?E` à être un type de structure de traits, soit une variable contrainte. On peut donner comme représentation de la connaissance partielle sur ce type `[...]`. Puis les deux instructions qui suivent donnent deux contraintes supplémentaires sur cette variable. dans un premier temps, son trait `f1` doit être du type de la variable `?X`. Le type de `?E` est augmenté en `[f1:t1,...]`, où `t1` est le type de `?X`. Puis le type de son trait `f2` est contraint à l'égalité avec celui de la variable `?Y`. La représentation de la connaissance sur le type de `?E` après cette suite d'instruction est donc `[f1:t1, f2:t2, ...]` où `t2` est le type de la variable `?Y`.

Pour prendre en compte le polymorphisme des structures de traits, nous utilisons la notion d'espèces (kinds en anglais), que nous empruntons à [Ohori, 1995]. Une espèce dénote un ensemble de types, de la même manière qu'un type dénote un ensemble de valeurs. Nous visons cependant quelque chose de différent : les structures que nous souhaitons typer sont des représentations contraintes, des structures de traits sous-spécifiées. Là où [Ohori, 1995] peut se permettre de typer une structure par un type AVM, nous nécessitons plus de souplesse sur leur typage.

Dans notre cas, une espèce peut être soit l'espèce universelle U, dénotant tous les types, soit une espèce $[\![l_1 : \tau_1, ..., l_n : \tau_n]\!]$ dénotant l'ensemble des types de structures de traits contenant l'ensemble des couples attribut-type spécifiés. On notera $\tau :: e$ le fait que le type τ est spécié par l'espèce e.

Les espèces représentent les contraintes sur les types. Par exemple, un type spécié par $[\![$`cat:CAT`$]\!]$ peut être `[num:int, cat:CAT]` ou `[cat:CAT, gen:GEN]`, mais pas `[cat:int]`.

Les règles de spéciation que nous utiliserons sont formulées de la manière suivante :

$$\begin{array}{l}
\mathcal{K} \vdash \tau :: U \text{ pour tout } \tau \text{ bien formé dans } \mathcal{K} \\
\mathcal{K} \vdash t :: [\![l_1 : \tau_1, \ldots, l_n : \tau_n]\!] \; si \; \mathcal{K}(t) = [\![l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots]\!] \\
\mathcal{K} \vdash [l_1 : \tau_1, \ldots, l_n : \tau_n] :: [\![l_1 : \tau_1, \ldots, l_n : \tau_n]\!] \\
\quad si \; [l_1 : \tau_1, \ldots, l_n : \tau_n] \text{ est bien formé dans } \mathcal{K}
\end{array}$$

où $\mathcal{K}$ est une affectation d'espèces à des variables de type, t est une variable de type, $\tau_1, \ldots \tau_n$ des types, et $l_1, \ldots l_n$ des étiquettes d'une structure de traits.

La première de ces règles précise que l'espèce universelle spécie tous les types. La seconde indique qu'un type spécié par une espèce l'est aussi par une espèce dénotant un sous ensemble des contraintes de la première. Enfin, la troisième règle dit que l'on peut spécier un type de structure de traits avec une espèce comprenant tous ses traits.

La règle de typage pour une structure de traits est la suivante :

$$\text{AVM} \; \frac{\mathcal{K}, \mathcal{T} \rhd M_i : \tau_i \qquad \mathcal{K} \vdash \alpha :: [\![l_i : \tau_i]\!] \qquad (1 \leq i \leq n)}{\mathcal{K}, \mathcal{T} \rhd [l_1 = M_1, ..., l_n = M_n] : \alpha}$$

où $\mathcal{K}$ est une affectation d'espèces à des variables de type, $\mathcal{T}$ une affectation de types à des variables du langage, $l_1, ..., l_n$ un ensemble d'étiquettes, $M_1, ..., M_n$ un ensemble d'expressions et $\tau_1, ..., \tau_n$ un ensemble de types. α est une variable de type, et son apparition constitue la différence principale entre notre règle de typage et celle de [Ohori, 1995]. Cette variable permet au type inféré d'évoluer, de recevoir successivement de nouvelles contraintes.

Pour un accès à une valeur d'un trait (par l'opérateur ".") :

$$\text{DOT}\ \frac{\mathcal{K}, \mathcal{T} \rhd M : \alpha \qquad \mathcal{K} \vdash \alpha :: [\![l : \tau]\!]}{\mathcal{K}, \mathcal{T} \rhd M.l : \tau}$$

Structures de traits constantes Les structures de traits typées jusqu'ici sont extensibles, ce qui signifie que le nombre de leurs traits peut augmenter, par unification ou par accès par l'opérateur `DOT`. Il est aussi possible de décrire des structures de traits constantes, c'est à dire dont le nombre de traits ne varie pas.

$$\text{CONST AVM}\ \frac{\mathcal{K}, \mathcal{T} \rhd M_i : \tau_i \qquad (1 \leq i \leq n)}{\mathcal{K}, \mathcal{T} \rhd [|l_1 = M_1, ..., l_n = M_n|] : [l_1 : \tau_1, ..., l_n : \tau_n]}$$

où $\mathcal{K}$ est une affectation d'espèces à des variables de type, $\mathcal{T}$ une affectation de types à des variables du langage, $l_1, ..., l_n$ un ensemble d'étiquettes, $M_1, ..., M_n$ un ensemble d'expressions et $\tau_1, ..., \tau_n$ un ensemble de types.

Pour un accès à une valeur d'un trait (par l'opérateur ".") :

$$\text{CONST DOT}\ \frac{\mathcal{K}, \mathcal{T} \rhd M : \tau' \qquad \mathcal{K} \vdash \tau' :: [\![l : \tau]\!]}{\mathcal{K}, \mathcal{T} \rhd M.l : \tau}$$

7.3.3 Typage du langage de contrôle

Intéressons nous maintenant aux types des instructions, en commençant par celles du langage de contrôle. Les constructeurs présents dans ses actions sémantiques sont celui de l'appel de classe, d'unification, et ceux de conjonction et de disjonction. On considère que toutes les expressions du langage de contrôle ont le type `void` si elles sont bien typées. La règle de typage pour l'unification est donc la suivante :

$$\text{EQ}\ \frac{\mathcal{K}, \mathcal{T} \rhd e_1 : \tau_1 \qquad \mathcal{K}, \mathcal{T} \rhd e_2 : \tau_1}{\mathcal{K}, \mathcal{T} \rhd e_1 = e_2 : \texttt{void}}$$

Une disjonction ou une conjonction est bien typée si ses deux opérandes ont le type `void`.

$$\text{CONJ}\ \frac{\mathcal{K}, \mathcal{T} \rhd l_1 : \texttt{void} \qquad \mathcal{K}, \mathcal{T} \rhd l_2 : \texttt{void}}{\mathcal{K}, \mathcal{T} \rhd l_1; l_2 : \texttt{void}}$$

$$\text{DISJ}\ \frac{\mathcal{K},\mathcal{T} \rhd l_1 : \texttt{void} \qquad \mathcal{K},\mathcal{T} \rhd l_2 : \texttt{void}}{\mathcal{K},\mathcal{T} \rhd l_1|l_2 : \texttt{void}}$$

Concernant le langage de contrôle permettant de combiner des littéraux à l'intérieur d'une dimension, le principe est le même, mais les opérandes ne sont pas limitées au type `void` : on considère que les littéraux d'une dimension ont leur type propre, ce qui permet notamment de différencier deux instructions provenant de dimensions différentes, et ayant une syntaxe identique (et éventuellement une sémantique différente).

Le type `void` est néanmoins présent dans certains littéraux, principalement les instructions d'égalité. On introduit la notation $\rhd_e$, signifiant qu'un contexte doit permettre de typer les littéraux avec soit le type `void`, soit le type `e`. Une contribution à une dimension est bien typée si les littéraux ont le type adéquat pour la dimension.

$$\text{DIM}\ \frac{\langle d\rangle : \tau \in \mathcal{T} \qquad \mathcal{K},\mathcal{T} \rhd_\tau E}{\mathcal{K},\mathcal{T} \rhd \langle d\rangle\{E\} : \texttt{void}}$$

où $\mathcal{T}$ représente le contexte, d une dimension, τ le type associé à cette dimension dans le contexte, et E une instruction de d.

$$\text{CONJDIM}\ \frac{\mathcal{K},\mathcal{T} \rhd_c l_1 \qquad \mathcal{K},\mathcal{T} \rhd_c l_2}{\mathcal{K},\mathcal{T} \rhd_c l_1; l_2}$$

$$\text{DISJDIM}\ \frac{\mathcal{K},\mathcal{T} \rhd_c l_1 \qquad \mathcal{K},\mathcal{T} \rhd_c l_2}{\mathcal{K},\mathcal{T} \rhd_c l_1|l_2}$$

$$\text{DIMTYPE}\ \frac{\mathcal{K},\mathcal{T} \rhd c_1 : \texttt{void}}{\mathcal{K},\mathcal{T} \rhd_c c_1} \quad \text{DIMTYPE}\ \frac{\mathcal{K},\mathcal{T} \rhd c_1 : c}{\mathcal{K},\mathcal{T} \rhd_c c_1}$$

7.3.4 Typage des classes

Abstractions. Le principal travail de l'étape de vérification de types concerne le type des classes, puisque c'est à l'intérieur de celles ci que les opérations sur les descriptions sont effectuées. Typer le contenu d'une classe est donc essentiel, mais il est également important d'associer un type à l'abstraction. Le type d'une classe est une application de la liste des types de ses paramètres dans l'ensemble des types de ses variables exportées (soit son vecteur d'export).

Dans cette section, nous choisissons de formaliser une classe en utilisant la notation des types fonctionnels :

$$A[Y_1, \ldots, Y_m] \rightarrow < X_1, \ldots, X_n > | \ B$$

où A est l'identifiant de la classe, $X_1, \ldots X_n$ les variables exportées, $Y_1, \ldots Y_m$ les paramètres de la classe, et B ses instructions. $\rightarrow$ est l'application fonctionnelle. La règle de typage utilisée est la suivante :

$$\text{CLASS}\ \frac{\mathcal{K},\mathcal{T}[Y_i : \tau_i,\ X_i : \tau'_i] \rhd B : \texttt{void}}{\mathcal{K},\mathcal{T} \rhd A[Y_1,\ldots,Y_m] \rightarrow < X_1,\ldots,X_n > \mid B :< \tau_1,\ldots,\tau_m > \rightarrow [|\tau'_1,\ldots,\tau'_n|]}$$

signifiant que la classe est bien typée si son corps est typé `void` dans le contexte augmenté avec les types de ses paramètres et de ses variables exportées (soit $\mathcal{K},\mathcal{T}[Y_i : \tau_i,\ X_i : \tau'_i]$).

En guise d'exemple, considérons la classe suivante :

```
class Classe[I]
export A
declare ?A
{
  I=4;
  ?A="chaine"
}
```

Le type de `Classe` est une application du type de son paramètre I vers le type de son vecteur d'export, soit le vecteur [|A=A|]. À l'intérieur de la classe, I est unifié avec une valeur entière et A avec une chaîne, leurs types sont donc contraints. Le type de `Classe` est donc ((`int`) → [|`A` = `string`|]).

7.3.5 Typage des dimensions

De même que pour les structures de traits, le type d'une dimension est contraint dans la méta-grammaire.

Une dimension permet l'expression d'un certain nombre de contraintes, partageant le même type, comme nous l'avons discuté en 7.3.2. En plus de ce type des littéraux, une dimension introduit un certain nombre de types pour les structures qu'elle permet de décrire. Ces types sont accessibles par une notation : `<dimension:typename>`. Accéder à ces types permet de les contraindre, ou de les utiliser dans la définition d'autres types.

Les types introduits par la dimension **syn**, présentée brièvement dans 3.3.4 et permettant de décrire des fragments d'arbres, sont au nombre de quatre. Le type des littéraux est **`<syn:tree>`**. Les propriétés et les traits associés aux noeuds ont respectivement les types **`<syn:`**`props`**`>`** et **`<syn:`**`feats`**`>`**. Un noeud de la dimension a le type **`<syn:node>`**. Une déclaration de noeud, est bien typée (par le type des littéraux de la dimension, soit **`<syn:`**`tree`**`>`**) si ses traits et propriétés sont bien typés pour la dimension.

$$\text{SYNNODE}\ \frac{\mathcal{K},\mathcal{T} \rhd [Ps...] : \texttt{<syn:props>} \quad \mathcal{K},\mathcal{T} \rhd [Fs...] : \texttt{<syn:feats>} \quad \mathcal{K},\mathcal{T} \rhd x : \texttt{<syn:node>}}{\mathcal{K},\mathcal{T} \rhd \texttt{node}\ x\ (Ps...)\ [Fs...] : \texttt{<syn:tree>}}$$

De même, les relations de dominance et de précédence sont bien typées si les deux noeuds concernés sont bien typés pour la dimension.

$$\text{SYNDOM}\ \frac{\mathcal{K},\mathcal{T} \rhd x : \texttt{<syn:node>} \quad \mathcal{K},\mathcal{T} \rhd y : \texttt{<syn:node>}}{\mathcal{K},\mathcal{T} \rhd x \rightarrow y : \texttt{<syn:tree>}}$$

$$\text{SYNPREC}\ \frac{\mathcal{K},\mathcal{T} \rhd x : \texttt{<syn:node>} \qquad \mathcal{K},\mathcal{T} \rhd y : \texttt{<syn:node>}}{\mathcal{K},\mathcal{T} \rhd x \prec y : \texttt{<syn:tree>}}$$

Les arbres syntaxiques exprimés avec la notation arborescente sont typés récursivement.

$$\text{SYNTREE}\ \frac{\mathcal{K},\mathcal{T} \rhd x : \texttt{<syn:node>} \qquad \mathcal{K},\mathcal{T} \rhd f_i : \texttt{<syn:tree>} \qquad (1 \leq i \leq n)}{\mathcal{K},\mathcal{T} \rhd \texttt{node}\ x\{f_1, ..., f_n\} : \texttt{<syn:tree>}}$$

Les variables de type fournies par les dimensions permettent, même dans le cas où peu de contraintes sont initialement données, d'assurer la cohérence des données décrites dans la dimension.

Prenons l'exemple suivant, où l'on n'exprime aucune contrainte sur les variables de type.

```
type CAT   = {v, n, s}
type GEN   = {sg, pl}
type COLOR = {black, white, red}

class UneClasse[]
declare ?X ?Y ?Z
{
  <syn>{
    node ?X [cat=v]
    node ?Y [agr=[gen=m,num=sg]]
    node ?Z (color=red) [cat=n]
  }
}
```

les types inférés pour la dimension si l'on ne considère que ce code sont :

```
type <syn:feats> = [cat:CAT,agr:[gen:GEN,num:NUM]]
type <syn:props> = [color:COLOR]
```

Si dans toute autre contribution à la dimension **<syn>**, une déclaration viole une des contraintes posées jusqu'ici sur la variable de type, une erreur de type est levée. C'est le cas dans l'exemple suivant :

```
  <syn>{
    node ?X [cat=3]
  }
```

L'accès aux types de la dimension par le biais de la notation dédiée permet plus de flexibilité sur le typage. Par exemple, dans le cas où l'on souhaite faire référence à des noeuds syntaxiques à l'intérieur de structures de traits (qui auront ici le type FS), il suffit d'écrire :

```
type FS = [noeud : <syn:node>]
```

7.3.6 Algorithme de typage

Principe. Le principe de notre algorithme de typage est, tout comme pour l'algorithme W qui implémente le système de types d'Hindley-Milner, d'inférer des types principaux, soit les types les plus généraux possibles du programme. Le but est de ne pas avoir à deviner les types, en utilisant le mécanisme d'unification pour les inférer progressivement.

Contributions à des règles de typage. Les briques participant à la construction du compilateur ont jusqu'ici apporté les règles syntaxiques permettant l'analyse du langage source. Pour typer le programme décrit dans ce langage source, on doit pouvoir typer chacun des constructeurs utilisés dans les actions sémantiques des règles de syntaxe. Ces constructeurs sont spécifiques aux briques, et pour assurer une indépendance maximale entre les briques, les règles réalisant le typage de ces constructeurs doivent également l'être. On attend par exemple de la brique `avm` qu'elle apporte les règles de typages pour tous les constructeurs du module `avm`, soit `avm:avm`, `avm:feat` et `avm.dot`. Ces règles sont contenues dans le module de typage de la brique, soit le fichier `typer.yap`.

Structures de traits. Les règles de typage pour les structures de traits sont établies d'après les déclarations faites dans la métagrammaire. Parmi les déclarations de type proposées pour les traits dans XMG-1, nous privilégions désormais celle permettant de définir un type construit, à savoir :

```
type FeatsType = [ cat : CAT,
                   agr : [num : NUM, gen : GEN] ]
```

Cette déclaration définit une variable de type, `FeatsType`, avec une contrainte d'espèce, pouvant typer une structure de traits dans les conditions suivantes : si elle possède un trait `cat`, sa valeur doit être compatible avec le type `CAT`, et si elle possède un trait `agr`, sa valeur doit être compatible avec une autre variable de type représentant l'avm `[num : NUM, gen : GEN]`. La compilation de cette déclaration enrichit le contexte en associant à FeatsType une variable sur laquelle nous poserons ces contraintes.

Le type d'une structure de traits est représenté par une structure de traits associant chaque trait à son type. Par exemple, la structure de traits `[cat=s, agr=[num=sg]]` a pour type la structure de traits

```
[cat : CAT,
 agr : [num : NUM] ]
```

Le codage des règles de typage énoncées précédemment fait intervenir les variables attribuées en tant que variables contraintes. Ces variables sont les représentations que nous utilisons pour les espèces. L'algorithme de typage est pour le reste très similaire à celui donné par Ohori.

Les types des structures de traits étant eux mêmes des structures de traits, nous utilisons la bibliothèque évoquée précédemment (figure 7.4) pour les manipuler. Dans ce cas, on peut considérer que la variable attribuée est la variable de type que l'on souhaite contraindre. Son attribut, soit la liste des contraintes sur le type, est assimilable à son espèce.

Une unification entre deux types est l'unification de leurs variables attribuées. L'opération `FS.f` consiste en l'unification de la variable attribuée représentant le type de `FS` avec une nouvelle

représentant l'espèce $[\![f : F]\!]$, où F est une variable libre. Dans l'attribut résultant, F est unifié avec un type si le trait f existe dans l'espèce de FS.

Les types des structures de traits étant représentés par des variables contraintes, ceci nous dispense de devoir utiliser le contexte $\mathcal{K}$ pour les espèces.

Polymorphisme des types des classes. Dans les langages fonctionnels, les abstractions sont liées à l'utilisation de types polymorphes. C'est le cas dans l'exemple de code Haskell suivant :

```
let hd (x:xs) = x
  in (hd[1,2,3], hd[True,False])
```

où hd désigne une fonction associant à une liste son premier élément. Ici, le type de hd ne dépend a priori pas de x. le type de x dans les deux instances de hd doit pouvoir être différent (en l'occurrence entier puis booléen). Le **let** opère une généralisation sur hd pour rendre ce polymorphisme possible. Le type de hd est après cette généralisation $\forall a : [a] \to a$.

Les abstractions que nous utilisons sont les classes, et c'est donc dans leur cas que les généralisations apparaissent. Une fois le type d'une classe calculé, un certain nombre des types des paramètres ou des variables exportées peuvent ne pas avoir été contraints. On est alors en présence d'un type de classe polymorphe. Les variables de type non contraintes lors du typage expriment ce polymorphisme. Il faut en revanche s'assurer que ces variables ne reçoivent pas de contraintes lors du typage des instanciations de la classe. Pour cela, on réalise une généralisation, comme celle proposée dans le système de types d'Hindley–Milner ([Hindley, 1969], [Milner, 1978]).

La règle en question est la suivante :

$$\text{LET}\ \frac{\mathcal{K},\mathcal{T} \rhd e_0 : \tau \qquad \mathcal{K},\mathcal{T}, x : \overline{\mathcal{T}}(\tau) \rhd e_1 : \tau'}{\mathcal{K},\mathcal{T} \rhd \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau'}$$

où $\overline{\mathcal{T}}$ correspond à la quantification de tous les types variables non liés dans $\mathcal{T}$, défini de la manière suivante :

$$\overline{\mathcal{T}}(\tau) = \forall \hat{\alpha}.\tau$$

$$\hat{\alpha} = free(\mathcal{T}) - free(\mathcal{T})$$

Pour établir un parallèle avec le système de type de Hindley-Milner, la définition d'une classe est assimilée à un

$$\texttt{let}\ A = [Y_1, \ldots, Y_n] \to < X_1, \ldots, X_n > \mid B\ \texttt{in}\ P$$

où P est le reste de la description métagrammaticale.

On peut motiver ce choix de réaliser des généralisation dans des cas comme la classe suivante :

```
class Egal[X,Y]
{
  X=Y
}
```

Ici, un appel à la classe réalise simplement une unification entre les deux valeurs passées en paramètres. Les types de X et Y doivent donc être unifiés. Le type de cette classe est donc d'après notre proposition précédente ($\forall T, (T, T) \rightarrow [||]$).

Si l'on crée une instance de la classe, par exemple `Egal[3,?A]`, le type ($(T, T) \rightarrow [||]$) reçoit de nouvelles contraintes et devient ($(int, int) \rightarrow [||]$). Ceci implique qu'un appel ultérieur à la même classe, avec des paramètres d'un autre type (par exemple `Egal["abc",?Y]`, sera interdit. Ceci est incohérent avec l'indépendance du type de la classe et du type de ses paramètres relevée précédemment. À chaque typage d'un appel, ces variables de type devraient être réinitialisée pour que le type de `Egal` soit ($(T, T) \rightarrow [||]$) indépendamment des autres instanciations de la classe.

7.3.7 Implantation

On parcourt l'arbre de syntaxe abstraite en utilisant les règles de typage qui ont été apportées par les briques. Les règles de typage sont comprises dans deux prédicats : `xmg:type_stmt/2` pour les types des instructions et `xmg:type_expr/2` pour les expressions. Ces prédicats utilisent un accumulateur fourni par les EDCG (`xmg_brick_mg_typer:types`) rendant accessibles les types des variables du contexte. Les deux arguments des prédicats de typage sont l'instruction ou expression à typer et le type inféré.

La figure 7.5 présente deux clauses du prédicat `xmg:type_stmt` effectuant le typage des constructeurs **`node`** et `dom` de la brique **`syn`**. Ces prédicats constituent l'implantation des règles `SYNNODE` et `SYNDOM` présentées précédemment.Dans la première de ces clauses, on souhaite typer le noeud en fonction du type des littéraux de la dimension, soit iprologSyn:tree(FType,PType), où `Syn` est l'identifiant de la dimension, `FType` le type des traits et `PType` le type des propriétés dans cette dimension. Le prédicat `xmg:get_var_type` récupère dans le contexte le type d'une variable.

Les correspondances entre la formalisation et l'implantation sont les suivantes :

`xmg:get_var_type` est comparable à la notation $\mathcal{K}, \mathcal{T} \ \triangleright$, l'opérateur `:--` correspond à la barre d'inférence. Les types inférés sont préfixés par une variable (`Syn`), qui est unifiée avec l'identifiant de la dimension.

Pour la première règle, `Syn:node` est l'équivalent de **`<syn:node>`**, mais dans lequel `Syn` est une variable Prolog, et permet donc de s'adapter à plusieurs dimensions (la variable est unifiée avec l'étiquette de la dimension). `ID` `Props` et `Feats` sont ceux de x, $(Ps\ldots)$ et $[Fs\ldots]$. Le type `Syn:tree`, correspond au type des littéraux, `syn:tree`, mais est dans l'implémentation paramétré par les types des structures de traits et des propriétés (`FType` et `PType`). Ces derniers sont les équivalents des types `<syn:feats>` et `<syn:props>`.

Pour la seconde règle, `N1` et `N2` correspondent à x et y, et la variable `Dom` à $\rightarrow$ (dans le cas de la dominance immédiate, `Dom` est unifié avec `'->'`).

```
xmg:type_stmt(syn:node(ID,Props,Feats),
              Syn:tree(FType,PType)):--
        xmg:type_expr(Feats,FType),
        xmg:type_expr(Props,PType),
        xmg:get_var_type(ID,Type),
        Type=Syn:node,
        !.

xmg:type_stmt(syn:dom(Dom,N1,N2),
              Syn:tree(FType,PType)):--
        xmg:get_var_type(N1,T1),
        T1=Syn:node,
        xmg:get_var_type(N2,T2),
        T2=Syn:node,
        !.
```

FIGURE 7.5 — Extrait du code d'un typer (syn)

7.4 Étape de dépliage

7.4.1 Principe

La troisième étape de la compilation du langage métagrammatical vers du code Prolog consiste à un dépliage de l'arbre de syntaxe abstraite (construit par l'analyse syntaxique).

Le but de cette étape est le suivant : les structures que l'on souhaite manipuler sont des structures contraintes, on veut donc réduire la description en un jeu de contraintes. Cette conversion est réalisée de manière récursive sur le terme en entrée.

Le langage cible vers lequel le dépliage est effectué est un langage noyau. Les premières étapes de la compilation permettent donc de passer de la syntaxe concrète, dont le but est de faciliter l'expression des contraintes, à un langage facilitant leur implantation.

Langage noyau. Le langage noyau a pour objectif d'être minimal, et donc de ne contenir qu'un nombre très restreint d'instructions. L'expressivité de ce langage minimal doit être identique à celle du langage métagrammatical, à la syntaxe riche. Par conséquent, différents littéraux du langage métagrammatical peuvent être dépliées en la même instruction. On peut prendre pour exemple le langage de description d'arbres, qui propose deux syntaxes alternatives, l'une étant une notation infixe et l'autre une notation arborescente. Ainsi, les deux littéraux `?X ->?Y` et `node ?X{node ?Y}` sont dépliées en la même instruction.

Deux types de règles de dépliage cohabitent : les règles dépliant les instructions (accumulant un ensemble d'instructions du langage noyau) et celles dépliant les expressions (auxquelles on associe une variable contrainte représentant la valeur de cette expression).

Format. Les instructions du langage noyau possèdent un format assez homogène, elles se divisent en trois types : les instructions consistant à contraindre une variable, celles consistant à accumuler du matériel dans une dimension, et les instructions d'égalité. Elles ont respectivement les syntaxes suivantes :

```
(Variable,Constrainte_sur_la_variable)
(Accumulation,Dimension_cible)
eq(Variable1,Variable2)
```

Aucune ambiguïté n'est possible entre les deux premières instructions, en raison de l'appartenance de chacun des paramètres à un module. Les dimensions appartiennent à un module spécifique les différenciant des contraintes.

Le langage noyau du DSL, élaboré à partir de briques, se compose d'instructions de ces formes combinés par les connecteurs logiques $\wedge$ et $\vee$.

Contextes. L'étape de dépliage est également celle durant laquelle nous construisons les contextes, et en particulier le contexte des classes. Ce travail consiste à créer pour chaque classe une table associant initialement chaque identifiant de variable accessible dans la classe (soit déclarée, soit importée) à un variable Prolog. Pour le dépliage de chaque classe, cette table, nommée `decls` est rendue facilement accessible pour consultation et enrichissement (avec les variables créées pendant le dépliage des expressions) par les EDCG.

7.4.2 Formalisation du dépliage

Pour formaliser les règles de dépliage, nous utiliserons $\mathcal{U}$ pour désigner la fonction de dépliage. Les règles auront la forme $\mathcal{U}(A) = B$, où A est une instruction ou une expression du langage métagrammatical, et B sa traduction dans le langage noyau.

Symboles. Le dépliage d'un symbole consiste à l'étiqueter selon le fait qu'il soit une variable ou une constante. Il s'agit donc de consulter le contexte.

$$\mathcal{U}(i) = v(i) \quad si \quad i : I \in \texttt{decls}$$
$$\mathcal{U}(i) = c(i)$$

où i est un symbole, `decls` la table associant à tous les identifiants de variables du contexte une variable Prolog.

Langage de contrôle. Commençons par des exemples d'instructions pour lesquelles le dépliage est trivial, la conjonction et la disjonction fournies par le langage de contrôle. Le travail consiste simplement à déplier chacune des deux opérandes des symboles de conjonction ou disjonction :

$$\mathcal{U}(s_1; s_2) = \mathcal{U}(s_1) \wedge \mathcal{U}(s_2)$$
$$\mathcal{U}(s_1|s_2) = \mathcal{U}(s_1) \vee \mathcal{U}(s_2)$$

Expressions. Si l'on s'intéresse maintenant à une instruction impliquant des contraintes sur les expressions, le dépliage demande une information supplémentaire : chaque expression est représentée par une variable contrainte, le dépliage d'une expression doit donc être paramétré par la variable qu'elle contraint. Nous exprimerons le paramétrage d'une règle de dépliage pour une expression e par un identifiant x de la manière suivante :

$$\mathcal{U}(e)_x$$

La règle pour l'égalité entre expressions peut par conséquent s'écrire:

$$\mathcal{U}(e_1 = e_2) = \mathcal{U}(e_1)_x \wedge \mathcal{U}(e_2)_y \wedge x = y$$

où x et y sont des variables crées dynamiquement durant le dépliage des expressions.

La règle pour le dépliage d'une expression disjonctive, du type "1 | 3" est la suivante :

$$\mathcal{U}(e_1|e_2)_x = \mathcal{U}(e_1)_x \vee \mathcal{U}(e_2)_x$$

Structures de traits. Revenons maintenant à l'exemple des structures de traits, pour lesquelles le langage noyau est composé de deux opérations. La première est l'initialisation d'une structure de traits vide :

$$x = []$$

où x est une variable du contexte de dépliage. La seconde opération correspond au `DOT`, contraignant la valeur d'un trait de la structure :

$$x[f] = v$$

où x est une variable du contexte de dépliage, f un trait de cette de cette structure, et v la valeur que l'on contraint pour ce trait.

La règle de dépliage d'une structure de traits depuis la syntaxe concrète est la suivante :

$$\mathcal{U}([f_1 = e_1, \ldots, f_n = e_n])_x = (x = []) \wedge (\bigwedge_{i=1}^{i=n} x[f_i] = v_i) \wedge (\bigwedge_{i=1}^{i=n} \mathcal{U}(e_i)_{v_i})$$

Chaque valeur de trait (e_i) est indépendamment dépliée de manière récursive, puis associée à une variable contrainte (v_i). Les instructions accumulées sont les suivantes : tout d'abord, on crée la structure de trait initiale (vide), associée à une nouvelle variable contrainte. Ensuite, on donne à cette variable l'ensemble des contraintes sur ses traits. Chaque attribut de la structure est contraint à l'égalité avec une nouvelle variable contrainte associée à sa valeur. Enfin, sont ajoutées récursivement les contraintes sur ces dernières variables.

Contributions aux dimensions. Concernant les contributions aux dimensions, il s'agit de déplier la contribution, en paramétrant ce dépliage par la dimension, soit l'accumulateur visé pour les contraintes.

$$\mathcal{U}(\langle d \rangle \{E\}) = \mathcal{U}(E)^{\langle d \rangle}$$

où $\mathcal{U}(E)^{\langle d \rangle}$ signifie que l'expression E est dépliée dans la dimension $\langle d \rangle$, c'est à dire que les accumulations que ses instructions déclencheront seront effectuées dans cette dernière.

Pour la dimension **<syn>**, le langage noyau se compose d'une instruction de création de noeud, et de deux instructions pour associer à un noeud des propriétés et de traits. Enfin, il contient

un ensemble d'instructions permettant l'accumulation de contraintes de dominance et de précédence entre noeuds.

Le dépliage d'une contrainte paramétrée par une dimension produit une instruction d'accumulation dont les paramètres sont la contrainte à accumuler et la dimension. Le dépliage d'une contrainte de précédence immédiate entre noeuds est formalisé comme suit :

$$\mathcal{U}(x >> y)^{\langle syn \rangle} = (\texttt{syn:prec(x,y)},\texttt{syn})$$

Pour ce qui est de la syntaxe arborescente des descriptions syntaxiques, le principe est identique, mais le dépliage comporte plusieurs instructions (étant donné que cette notation peut être considérée comme un sucre syntaxique) :

$$\mathcal{U}(\texttt{node}\ x\{\texttt{node}\ y\})^{\langle syn \rangle} = (\texttt{x},\texttt{syn:node}) \wedge (\texttt{y},\texttt{syn:node}) \wedge (\texttt{syn:dom(x,y)},\texttt{syn})$$

On déplie donc `node` x`{node` y`}` en la déclaration des deux noeuds x et y et la contrainte de dominance immédiate entre eux.

Abstractions. Le dépliage d'une classe consiste à créer un terme contenant les informations nécessaires pour construire une abstraction la représentant dans le langage cible de la compilation, soit un terme Prolog. Ces informations incluent son identifiant, ses paramètres, le vecteur de ses variables exportées, et le corps de la classe.

Nous utilisons ici la notation fonctionnelle de la section précédente. Le passage de la syntaxe concrète à cette notation consiste principalement à éliminer les déclaration et les imports. Les premiers sont inutiles une fois que le contexte a été construit [2]. Pour les seconds, ils sont traduits en instructions d'appels de classes ajoutées au début du corps de la classe.

Le langage noyau pour les abstractions se compose d'une seule instruction :

$$\texttt{class}(A, [P_1, \ldots, P_n], [E_1, \ldots, E_n], Instr)$$

où A est l'identifiant de la classe, $P_1, \ldots, P_n$ ses paramètre, $E_1, \ldots, E_n$ ses variables exportées, et $Instr$ ses instructions.

La règle de dépliage que nous utilisons est la suivante :

$$\begin{aligned}\mathcal{U}(A[Y_1, \ldots, Y_n] \rightarrow < X_1, \ldots, X_n > | \ B) = \\ & \mathcal{U}(Y_1)_{y_1} \wedge \ldots \wedge \mathcal{U}(Y_n)_{y_n} \wedge \mathcal{U}(X_1)_{x_1} \wedge \ldots \wedge \mathcal{U}(X_n)_{x_n} \\ & \wedge \texttt{class}(A, [y_1, \ldots, y_n], [x_1, \ldots, x_n], \mathcal{U}(B))\end{aligned}$$

où P est la liste des paramètres, E celle des variables exportées, et B les instructions du corps de la classe.

Appel de classe. Les appels de classe sont des expressions, et utilisent donc un paramètre représentant la variable contrainte représentant l'appel (soit le vecteur d'export).

$$\mathcal{U}(A[Y_1, \ldots, Y_n])_x = \mathcal{U}(Y_1)_{y_1} \wedge \ldots \wedge \mathcal{U}(Y_n)_{y_n} \wedge \texttt{call}(A, [y_1, \ldots, y_n], x)$$

2. Une étape précédant le dépliage construit pour chaque classe une table associant les identifiants des variables accessibles à des variables Prolog.

7.4.3 Exemples

La syntaxe d'une description dans la dimension **syn** est définie par le fichier `lang.def` suivant :

```
%%

SynStmt : Node {$$=$1}
        | Dom  {$$=$1}
        | Prec {$$=$1}
        | Eq   {$$=$1}
        | Tree {$$=$1};

Node : 'node' (Var)? MaybeProps (_AVM)?
       {$$=syn:node($2,$3,$4)};

MaybeProps : '(' Props ')' {get_coord($1,Coord),
                            $$=some(avm:avm(Coord,$2))}
           | {$$=none};

Props : (Feat // ',')+ {$$=$1};

Feat : id '=' id {$$=avm:feat($1,$3)}
     | id '=' int {$$=avm:feat($1,$3)}
     | id '=' bool {$$=avm:feat($1,$3)}
     | id {$$=avm:feat($1)};

Dom : IdOrNode DomOp IdOrNode {$$=syn:dom($2,$1,$3)};

IdOrNode : Node {$$=$1}
         | Var {$$=$1};

DomOp : '->' {$$=$1}
      | '->+'{$$=$1}
      | '->*'{$$=$1} ;

Prec : IdOrNode PrecOp IdOrNode {$$=syn:prec($2,$1,$3)};

PrecOp : '>>' {$$=$1}
       | '>>+'{$$=$1}
       | '>>*'{$$=$1} ;

Tree : Node '{' Children '}' {$$=syn:tree($1,$3)};

Children : Child {$$=syn:children($1,none)}
         | Child TreePrecOp Children
           {$$=syn:children($1,brothers($2,$3))};

Child : TreeDomOp Node {$$=syn:child($1,$2)}
```

```
        | TreeDomOp Tree {$$=syn:child($1,$2)};

TreePrecOp : ',,,' {$$=$1}
           | ',,,+'{$$=$1}
           | {$$=none};

TreeDomOp : '...' {$$=$1}
          | '...+'{$$=$1}
          | {$$=none};

Eq : Expr '=' Expr {$$=syn:eq($1,$3)};

Expr : id {$$=$1}
     | string {$$=$1};

Var : id {$$=$1}
    | '?' id {$$=$2};

%%
```

Les arbres d'analyse formés par ces règles hors contexte sont réécrites après dépliage à l'aide de cet ensemble minimal d'instructions, constituant le langage noyau de la brique :

```
(Var,syn:node)
(Var,syn:props(P))
(Var,syn:feats(F))
(Dim,syn:dom(N1,Op,N2))
(Dim,syn:prec(N1,Op,N2))
eq(X,Y)
```

Illustrons maintenant le passage du code métagrammatical au code déplié. La contribution suivante à la dimension syntaxique :

```
<syn>{
  node ?X [cat=s] {
    node ?Y [cat=n]
    node ?Z [cat=v]
  }
}
```

se traduit en la conjonction d'instructions suivante :

```
(X,syn:node)∧(FX,avm:avm)*
(FX,avm:dot(cat,s))*∧(X,syn:feats(FX))∧
(Y,syn:node)∧(FY,avm:avm)*∧(FY,avm:dot(cat,n))*∧
(Y,syn:feats(FY))∧(syn,dom(X,->,Y))∧
(Z,syn:node)∧(FZ,avm:avm)*∧(FZ,avm:dot(cat,v))*∧
(Z,syn:feats(FZ))∧(syn,dom(X,->,Z))∧(syn,prec(Y,>>,Z))
```

où les instructions proviennent de deux langages noyau : celui de la brique **syn** détaillé plus haut, et pour les instruction marquées par * celui de la brique avm (qui paramètre la brique **syn** dans ce cas).

7.4.4 Implantation

Comme l'illustrent les exemples précédents, l'arbre de syntaxe abstraite est composé de deux types de termes : ceux représentant les instructions du langage métagrammatical et les expressions, qui sont manipulées dans les instructions. L'unfolder de XMG s'appuie deux prédicats (unfold_stmt et unfold_expr), visant à déplier les termes de chacun de ces deux types. Les clauses de ces prédicats sont apportées modulairement par les briques utilisées, au moyen de leur module unfolder.yap.

Le prédicat de dépliage des instructions prend en entrée un terme et provoque une accumulation d'instructions. Le prédicat de dépliage des expressions prend en entrée un terme, provoque également une éventuelle accumulation d'instructions, et renvoie une variable correspondant à l'expression. Nous utilisons pour provoquer les accumulations nécessaires à cette étape les EDCG présentées en 7.1.3.

Dans une brique, l'unfolder doit proposer une règle de dépliage pour chacun des constructeurs apparaissant dans sa brique de langage. L'unfolder de la brique avm contient donc les règles présentées dans la figure 7.6, correspondant aux constructeurs avm, feat et dot.

Dans ce code, constraints::enq(Contrainte) signifie l'accumulation de la contrainte Contrainte dans l'accumulateur constraints. L'espace de noms est de nouveau important à cette étape : pour assurer l'absence de conflits entre les constructeurs, chacun de ceux manipulés par une brique doit être local à cette brique.

xmg:new_target_var(Value) signifie la création d'une nouvelle variable dans le contexte, dont l'identifiant est Value. avm:avm(Coord) et avm:feat(Target1,Value) expriment respectivement la contrainte pour une variable de représenter une structure de traits et de contenir un trait spécifique (Target1=Value).

Si l'on considère l'arbre de syntaxe abstraite suivant :

```
avm:avm(coord(file,1,1),avm:feat(f1,v1),avm(feat(f2,v2))
```

dans lequel f1, f2, v1 et v2 sont des constantes. Le dépliage réalisé par le code présenté est :

```
 (v('XMG-AVM1'),avm:avm(coord(file,1,1)))
∧(v('XMG-AVM1'),avm:feat(c(f1),c(v1)))
∧(v('XMG-AVM1'),avm(feat(c(f2),c(v2))))
```

où 'XMG-AVM1' est l'identifiant d'une variable générée par xmg:new_target_var.

7.4.5 Extensibilité

Pouvoir effectuer un dépliage de l'intégralité d'un langage implique d'avoir une méthode de dépliage pour tous les arbres d'analyse que son parser peut générer. Cela implique de disposer d'une règle de dépliage pour chacun des constructeurs apparaissant dans les actions sémantiques associées à ses règles d'analyse syntaxique.

```
xmg:unfold_expr(avm:avm(Coord, Feats), Target) :--
  constraints::enq((Target,avm:avm(Coord))),
  unfold_feats(Feats, Target).

unfold_feats([], _):-- !.
unfold_feats([Feat|Feats], Target) :--
  unfold_feat(Feat, Target),
  unfold_feats(Feats, Target).

unfold_feat(avm:feat(F,V), Target) :--
  xmg:new_target_var(Value),
  xmg:unfold_expr(F, Target1),
  xmg:unfold_expr(V, Value),
  constraints::enq((Target,avm:feat(Target1,Value))),!.

xmg:unfold_expr(avm:dot(V1,V2),Target):--
  xmg:new_target_var(Target),
  xmg:unfold_expr(V1,T1),
  xmg:unfold_expr(V2,c(T2)),
  constraints::enq((Target,avm:dot(T1,c(T2)))),!.
```

FIGURE 7.6 — extrait du code d'un unfolder (avm)

Comme nous l'avons vu dans cette section, les règles qu'une brique contribue à l'unfolder sont de deux types : `unfold_expr` pour les expressions et `unfold_stmt` pour les instructions. Étendre l'unfolder d'un compilateur consiste donc à écrire de nouvelles clauses pour ces deux prédicats, faisant apparaître les nouveaux constructeurs du langage.

Comme pour l'étape de typage, les constructeurs spécifiques permettent de séparer les clauses en modules, dans le but d'éviter qu'elles ne s'appliquent à mauvais escient.

7.5 Étape de génération

7.5.1 Principe

La génération de code est la dernière étape du processus de compilation. Elle prend en entrée les contraintes accumulées par l'unfolder, et les traduit en instructions exécutables. Le principe de modularisation pour la génération de code est le même que celui des étapes de compilation précédentes. Chaque brique doit contribuer au générateur avec les règles de génération pour les instructions qui lui sont propres, ces règles étant décrites dans le module `generator.yap`. Ces instructions sont celles produites par l'unfolder de la brique. Le langage d'instructions manipulé par le générateur de code implémente le langage noyau.

Ce module est l'unique étape de la compilation dépendante du langage cible. Créer un compilateur dont les instructions de sortie utilisent un autre langage de programmation à partir d'un compilateur existant consiste donc simplement en l'écriture de nouvelles règles pour le générateur.

L'objectif est de générer pour chaque classe un prédicat Prolog déclenchant des accumulations dans les dimensions. À cette étape sont donc également créés les accumulateurs représentant les dimensions, que l'on associe à chacun des prédicats pour lesquels on génère le code.

Les instructions prolog générées à cette étape peuvent être classées en deux catégories. La première catégorie correspond à l'utilisation de la machine virtuelle Prolog standard, en général pour l'unification ou l'accumulation de contraintes dans les dimensions. La seconde catégorie correspond à des machine virtuelles spécifiques, propres à la brique, qui seront détaillées dans la section 8.2.

7.5.2 Formalisation et implantation

Pour formaliser les règles de génération, nous utiliserons $\mathcal{G}$ pour désigner la fonction de génération. Les règles auront la forme $\mathcal{G}(A) = B$, où A est une instruction du langage noyau, et B sa traduction dans le langage cible de la compilation.

La génération de code Prolog pour les opérations de conjonction ou de disjonction peut être exprimée de la façon suivante :

$$\mathcal{G}(e_1 \wedge e_2) = \mathcal{G}(e_1)\ ,\ \mathcal{G}(e_2)$$
$$\mathcal{G}(e_1 \vee e_2) = \mathcal{G}(e_1)\ ;\ \mathcal{G}(e_2)$$

soit des appels récursifs à la fonction de génération, produisant deux suites d'instructions, séparées par les opérateurs Prolog de conjonction et de disjonction (`,` et `;`).

Deux accumulateurs sont principalement utilisés à la génération de code. L'un d'entre eux (`decls`) représente le contexte construit à l'étape précédente, et l'autre (`code`) la liste des instructions accumulées.

Structures de traits

Formalisation. Pour le langage noyau vu dans la section précédente pour les structures de traits, soit :

$$x = []$$
$$x[f] = y$$

Les règles ont la forme suivante :

$$\mathcal{G}(x = []) = \texttt{avm:avm}(x \downarrow, [])$$
$$\mathcal{G}(x[f] = y) = \texttt{avm:avm}(x \downarrow, [f = y \downarrow])$$
$$\mathcal{G}(x[f] = c) = \texttt{avm:avm}(x \downarrow, [f = c])$$

où x et y sont des symboles de variables, c une constante, et `avm` le constructeur de structure de trait, prenant en arguments la variable associée à la structure et la liste de ses traits. L'opérateur $\downarrow$ permet d'accéder à la variable Prolog associée à une variable de la métagrammaire (ou à l'une de celles créées pendant le dépliage), par l'intermédiaire de la table de déclarations créée au moment du dépliage.

Implantation. Le code correspondant à ces règles utilise le prédicat `xmg:generate_instr`, dont l'unique argument est une instruction du langage noyau. Ce prédicat correspond à la traduction de la fonction $\mathcal{G}$.

```
xmg:generate_instr((v(T),avm:avm(Coord))) :--
        decls::tget(T,Var),
        code::enq(xmg_brick_avm_avm:avm(Var,[])),
        !.
xmg:generate_instr((v(T),avm:feat(c(F),v(T2)))):--
        decls::tget(T,Var),
        decls::tget(T2,Var2),
        code::enq(xmg_brick_avm_avm:avm(Var,[F-Var2])),
        !.
xmg:generate_instr((v(T),avm:feat(c(F),c(T2)))):--
        decls::tget(T,Var),
        code::enq(xmg_brick_avm_avm:avm(Var,[F-T2])),
        !.
```

où `decls` est l'accumulateur correspondant à la table des déclarations, `tget` est la méthode permettant l'accès à une variable Prolog depuis sa clé (la variable métagrammaticale). Pour touts les identifiants étiquetés comme des variables au dépliage (`v(T)`, `v(T2)`), nous utilisons ce contexte. `decls::tget(T,Var)` est ici la traduction de $x \downarrow$ (x étant `T`).

`code` est l'accumulateur utilisé pour les instructions générées, `enq` est la méthode d'accumulation prenant en argument le terme à accumuler. `xmg_brick_avm_avm:avm/2`, l'instruction accumulée, a pour premier argument une variable attribuée, et pour second une liste d'associations contenant les traits de la structure que la variable représente.

Revenons à l'exemple déplié dans la section précédente :

```
 (v('XMG-AVM1'),avm:avm(coord(file,1,1)))
∧(v('XMG-AVM1'),avm:feat(c(f1),c(v1)))
∧(v('XMG-AVM1'),avm(feat(c(f2),c(v2))))
```

Le code généré par les prédicats présentés ici est :

```
xmg_brick_avm_avm:avm(XMG-AVM1,[]),
xmg_brick_avm_avm:avm(XMG-AVM1,[f1-v1]),
xmg_brick_avm_avm:avm(XMG-AVM1,[f2-v2])
```

où `XMG-AVM1` est la variable prolog associée à l'identifiant `'XMG-AVM1'` dans le contexte.

Contribution à une dimension

Formalisation. Pour les instructions de contributions aux dimensions, le travail consiste en la construction d'un terme, que l'unfolder a déjà façonné, puis en l'accumulation (dans le code généré) d'une instruction déclenchant l'accumulation (dans une dimension) de ce terme.

Rappelons le langage noyau dans lequel sont décrites les instructions de la dimension **<syn>** :

```
(Var,syn:node)
(Var,syn:props(P))
(Var,syn:feats(F))
```

```
(Dim,syn:dom(N1,Op,N2))
(Dim,syn:prec(N1,Op,N2))
eq(X,Y)
```

L'exemple suivant montre la règle de génération pour une relation de dominance entre noeuds syntaxiques.

$$\mathcal{G}((\texttt{syn:dom}(x, op, y), dim)) = dim\texttt{::put}(\texttt{dom}(x\downarrow, op, y\downarrow))$$

Implantation. On passe de l'instruction du langage noyau indiquant l'accumulation dans la dimension *dim* d'un terme construit avec le constructeur `syn:dom` à sa traduction prolog utilisant les EDCG et l'accumulateur identifié par *dim*.

```
xmg:generate_instr((syn:dom(v(N1),Op,v(N2),C),Acc)):--
        decls::tget(N1,V1),
        decls::tget(N2,V2),
        AccDom=..['::',(xmg_acc:Acc),put(dom(V1,Op,V2,C))],
        code::enq(AccDom),
        !.
```

L'argument de `xmg:generate_instr` est l'instruction du langage noyau indiquant l'accumulation dans la dimension `Acc` d'une relation de dominance entre les variables `N1` et `N2` du langage métagrammatical. La règle de génération consiste dans un premier temps à récupérer les variables Prolog associées à `N1` et `N2`.

Les variables `N1` et `N2` correspondent respectivement aux variables x et y de la formalisation précédente. Les instructions `decls::tget(N1,V1)` et `decls::tget(N2,V2)` sont les traductions de l'opération $\downarrow$ (`V1` correspond à $x\downarrow$ et `V2` à $y\downarrow$).

Le terme à accumuler est ensuite construit : `xmg_acc:Acc` désigne l'accumulateur identifié par `Acc`, `put` la méthode d'accumulation dans cette dimension. Le terme `dom(V1,Op,V2,C)` représente une relation de dominance entre les noeuds `V1` et `V2`, `Op` est l'opérateur de dominance (`->`, `->+` ou `->*`) et `C` la coordonnée où cette instruction apparaît dans la métagrammaire. Enfin, cette instruction est ajoutée à l'accumulateur `code`.

La raison pour laquelle nous passons par la variable `AccDom` et n'écrivons pas directement `xmg_acc:Acc::put(dom(V1,Op,V2,C))` est que nous souhaitons voir le symbole `::` interprété dans le programme généré et non au moment du chargement du fichier dans lequel il apparaît.

L'instruction correspondant à la création d'un noeud ne résulte quant à elle pas en une accumulation. Elle utilise une bibliothèque spécifique permettant de manipuler des représentations de noeuds.

```
xmg:generate_instr((v(TN),syn:node)):--
        decls::tget(TN,TV),
        code::enq(xmg_brick_syn_engine:inode(TV,TN)),
        !.
```

où `xmg_brick_syn_engine:inode` est un constructeur de noeud fourni par une bibliothèque de la brique `syn`. Il initialise un noeud avec des structures de traits vides en tant que propriétés et traits morphosyntaxiques. Il prend comme arguments la variable Prolog que l'on souhaite voir

représenter ce noeud, et un identifiant servant à repérer le noeud en question dans la ressource générée (ici, on utilise l'identifiant du noeud dans la métagrammaire).

Le langage cible de la génération pour la dimension **<syn>** est :

```
xmg_brick_syn_engine:inode(TV,TN)
xmg_brick_syn_engine:inodeprops(VN,VP)
xmg_brick_syn_engine:inodefeats(VN,VF)
xmg_acc:Acc::put(dom(V1,Op,V2,C))
xmg_acc:Acc::put(prec(V1,Op,V2,C))
X=Y
```

Pour la description métagrammaticale suivante :

```
<syn>{
  node ?X;
  node ?Y;
  node ?Z;
  ?X -> ?Y;
  ?X -> ?Z
}
```

On génère le code qui suit :

```
xmg_brick_syn_engine:inode(X,'X'),
xmg_acc:syn::put(X),
xmg_brick_syn_engine:inodeprops(X,XP),
xmg_brick_syn_engine:inodefeats(X,XF),
xmg_brick_syn_engine:inode(Y,'Y'),
xmg_acc:syn::put(Y),
xmg_brick_syn_engine:inodeprops(Y,YP),
xmg_brick_syn_engine:inodefeats(Y,YF),
xmg_brick_syn_engine:inode(Z,'Z'),
xmg_acc:syn::put(Z),
xmg_brick_syn_engine:inodeprops(Z,ZP),
xmg_brick_syn_engine:inodefeats(Z,ZF),
xmg_acc:syn::put(dom(X,->,Y)),
xmg_acc:syn::put(dom(X,->,Z))
```

FIGURE 7.7 — Code généré pour une description syntaxique

où X, Y, Z, sont des variables référant à des noeuds, XP, YP, ZP, XF, YF, ZF, des variables attribuées. Les attributs qui leurs sont associés sont des listes vides, étant donné que ces variables représentent des structures de traits vides.

7.5.3 Extensibilité

L'ajout de nouvelles instructions dans le langage noyau doit s'accompagner des règles de génération pour ces instructions. Pour cette raison, le module de génération d'une brique se compose

de l'ensemble des clauses de `generate_instr` permettant de traiter les instructions de son langage noyau.

7.5.4 Code spécifique aux classes

Formalisation. Les abstractions du langage metagrammatical sont traduites en des abstractions Prolog, des prédicats en l'occurrence. C'est l'exécution du code indéterministe de ces prédicats qui provoquera les accumulations dans les dimensions. Les prédicats ont pour arguments les paramètres de la classe et son vecteur d'export.

$$\mathcal{G}(\texttt{class}(A, P, E, B)) = \texttt{xmg:value_class}(A, \mathcal{G}(P), \mathcal{G}(E))\texttt{:-}\mathcal{G}(B)$$

où A est l'identifiant de la class, P est la liste des paramètres, E celle des variables exportées, et B les instructions du corps de la classe.

La traduction d'un appel de classe, quant à elle, est l'invocation du prédicat `xmg:value_class` que nous venons de définir. La variable x est unifiée avec le vecteur d'export généré par l'appel.

$$\mathcal{G}(\texttt{call}(A, P, x)) = \texttt{xmg:value_class}(A, P, x \downarrow)$$

Concernant les valuations (les instructions **`value`**, indiquant les classes que l'on souhaite utiliser comme axiomes de la métagrammaire), le travail est d'ajouter des clauses du prédicat `xmg:value` qui désigne les axiomes.

$$\mathcal{G}(\texttt{value}(A)) = \texttt{xmg:value}(A)$$

Exemple. Si l'on étend l'exemple précédent de description syntaxique, en l'incorporant dans une classe, comme le montre la figure 7.8, on obtient le code généré des figures 7.9 et 7.10 avant et après interprétation des opérateurs EDCG.

```
class test
declare ?X ?Y ?Z
{
        <syn>{
                node ?X;
                node ?Y;
                node ?Z;
                ?X -> ?Y;
                ?X -> ?Z
        }
}
```

Figure 7.8 — Exemple de classe contenant une description syntaxique

où `Trace0, Trace1, IFace, Syn0, Syn5, Sem` sont respectivement les valeurs initiales et finales des accumulateurs utilisés (ici, la trace d'exécution, l'interface et les dimensions **`<syn>`** et **`<sem>`**). `Syn1, Syn2, Syn3, Syn4` sont des états intermédiaires de l'accumulateur syntaxique.

```
xmg:value_class(test,params([]),exports([])):--
  xmg_acc:trace::put(test),
  xmg_brick_syn_engine:inode(X,'X'),
  xmg_acc:syn::put(X),
  xmg_brick_syn_engine:inodeprops(X,PX),
  xmg_brick_syn_engine:inodefeats(X,FX),
  xmg_brick_syn_engine:inode(Y,'Y'),
  xmg_acc:syn::put(Y),
  xmg_brick_syn_engine:inodeprops(Y,PY),
  xmg_brick_syn_engine:inodefeats(Y,FY),
  xmg_brick_syn_engine:inode(Z,'Z'),
  xmg_acc:syn::put(Z),
  xmg_brick_syn_engine:inodeprops(Z,PZ),
  xmg_brick_syn_engine:inodefeats(Z,FZ),
  xmg_acc:syn::put(dom(X,->,Y)),
  xmg_acc:syn::put(dom(X,->,Z)).
```

FIGURE 7.9 — Code généré pour une classe

```
xmg:value_class(test,params([]),exports([]),
                Trace0,Trace1,IFace,IFace,
                Syn0,Syn5,Sem,Sem):-
  (Trace1=[test|Trace0]),
  xmg_brick_syn_engine:inode(X,'X'),
  Syn1=[X|Syn0],
  xmg_brick_syn_engine:inodeprops(X,PX),
  xmg_brick_syn_engine:inodefeats(X,FX),
  xmg_brick_syn_engine:inode(Y,'Y'),
  Syn2=[Y|Syn1],
  xmg_brick_syn_engine:inodeprops(Y,PY),
  xmg_brick_syn_engine:inodefeats(Y,FY),
  xmg_brick_syn_engine:inode(Z,'Z'),
  Syn3=[Z|Syn2],
  xmg_brick_syn_engine:inodeprops(Z,PZ),
  xmg_brick_syn_engine:inodefeats(Z,FZ),
  Syn4=[dom(X,->,Y)|Syn3],
  Syn5=[dom(X,->,Z)|Syn4].
```

FIGURE 7.10 — Code généré pour une classe (après interprétation des EDCG)

Une exécution du code généré consiste à évaluer tous les buts correspondants aux classes que l'utilisateur a désigné comme axiomes.

Les prédicats utilisés pour décrire les classes doivent pouvoir accéder aux dimensions de façon transparente, et pour cette raison, nous utiliserons une nouvelle fois les EDCG pour modéliser les dimensions. Nous détaillerons leur utilisation dans la section suivante.

7.6 Constructeurs de briques

Jusqu'ici, les éléments de langage métagrammatical que nous avons évoqués pouvaient être assemblés simplement en combinant des briques. L'expression de connexions n'est pourtant pas suffisante dans certains cas, ou pas satisfaisante. Il est par exemple parfois nécessaire à l'instanciation d'une brique de définir des mots-clés spécifiques intervenant dans les règles de cette brique (sans avoir à modifier la définition `lang.def`). Nous proposons pour les cas de ce type un outil permettant de générer des briques à partir de squelettes de briques. Nous appelons ces briques paramétrées constructeurs de briques.

Dans cette section nous nous intéresserons à un exemple de brique nécessitant un paramétrage : la description du langage des dimensions, pour lequel l'assemblage est un peu plus subtil.

Partons d'un exemple simple de langage de description d'arbres (vu dans le chapitre 6) :

$$\begin{array}{rcl} \mathit{Stmt} & ::= & \textbf{node}\ \mathit{id} \\ & | & \textbf{node}\ \mathit{id}\ _\mathit{AVM} \\ & | & \mathit{id} \rightarrow \mathit{id} \\ & | & \mathit{id} \gg \mathit{id} \end{array}$$

Imaginons que nous souhaitions utiliser ce langage dans deux dimensions différentes (dont les étiquettes seraient `syn` et `syn2`), et donc étendre le langage de description pour délimiter les contributions à ces dimensions par des balises. En d'autres mots, si `Stmt` est l'axiome du langage de description de la dimension, la forme d'une contribution à une dimension est `<etiquette>{Stmt}`. Il est pour cela nécessaire de créer deux nouvelles briques de langage, ajoutant la syntaxe pour les balises (⟨`syn`⟩ et ⟨`syn2`⟩) :

$$\mathit{Dim} \quad ::= \quad \langle \texttt{syn} \rangle \{_\mathit{Stmt}\}$$

et

$$\mathit{Dim} \quad ::= \quad \langle \texttt{syn2} \rangle \{_\mathit{Stmt}\}$$

dans lesquels nous connectons les non-terminaux externes (*_Stmt*) à la brique de description d'arbres.

Créer une nouvelle brique pour chaque nouvelle instance d'une dimension n'est pas une solution très acceptable. Utiliser la même brique oblige à utiliser la même étiquette pour toutes les dimensions basées sur le même langage, puisque cette étiquette est un terminal du langage (un mot clé plus exactement).

Un nouveau type de brique, paramétrable, est donc nécessaire : un constructeur de brique. Dans le cas des dimensions, le premier de ces paramètres est l'étiquette que nous venons d'évoquer.

L'étiquette. L'étiquette est le paramètre permettant de générer le langage final de la dimension. Ce langage final est simplement le langage de description, augmenté de la balise permettant de délimiter une contribution à cette dimension. La brique de dimension doit donc contribuer à l'analyseur lexical par une règle de réécriture, qui a la particularité d'être paramétrable. Ce paramètre est donné à l'instanciation de la brique de la même manière qu'une connexion, en remplaçant le non terminal à connecter par l'identifiant du paramètre, soit `tag` dans le cas de l'étiquette.

Le langage de description. Si l'on s'en tient au langage que nous essayons de décrire jusqu'à maintenant, il suffit maintenant de connecter le non-terminal externe de la règle générée pour l'étiquette au langage de description. Néanmoins, dans les dimensions de XMG, la possibilité est traditionnellement d'offrir la possibilité de combiner les instructions avec les opérateurs logiques du langage de contrôle. Créer une nouvelle instance de la brique de contrôle dans la description yaml est une solution qui semble contraignante pour l'utilisateur.

Le second paramétrage de la brique de dimension, celui du langage de description (celui qui sera connecté à la règle de l'étiquette) permet d'éviter ce travail répétitif.

Ce paramétrage est quasiment équivalent à une connexion habituelle de non terminal externe. La seule différence est que le non terminal externe en question ne provient pas de la brique courante, soit `dim`, mais d'une nouvelle instance du langage de contrôle, créée automatiquement. Le paramètre donné, soit le langage de description, est donc donné comme langage à composer par ce langage de contrôle. En résumé, la déclaration de dimension suivante

```
control:
  _Stmt: dim_syn
dim_syn:
  tag: "syn"
  Stmt: syn
```

revient, au niveau du langage reconnu, et si l'on omet l'ajout de la syntaxe pour les balises **`<syn>`**, à la déclaration suivante

```
control:
  _Stmt: control_syn
control_syn:
  _Stmt: syn
```

Pour résumer, le constructeur de brique `dim` assemble un langage à partir de trois fragments. Le premier est un langage de dimension, composé d'une règle pour la balise. Le non terminal externe de cette règle est connecté au deuxième fragment, qui est un langage de contrôle. Ce langage de contrôle est connecté au troisième fragment, celui des instructions de la dimension. Les seules informations à donner au constructeur sont l'étiquette à placer dans la balise du premier fragment (avec le paramètre `tag`) et l'axiome du troisième fragment (avec le paramètre `Stmt`).

7.7 Point d'entrée de l'exécution

7.7.1 Lancer l'évaluation du code généré

À la fin de l'étape de génération, un ensemble de prédicats représentant les classes de la métagrammaire ont été générés. Il reste maintenant à définir un point d'entrée pour l'évaluation d'une partie de ces prédicats. Ce point d'entrée consiste à évalue le prédicat `xmg:value_all` puis à backtracker jusqu'à épuisement des solutions. Ce prédicat est le suivant :

```
xmg:value_all(Value,Class):-
        xmg:value(Class),
        xmg:start_value_class(Class,Value).
```

où `xmg:value` est un prédicat dont il existe une clause pour toutes les classes à évaluer (les clauses sont ajoutées lors de la génération de code des instructions **`value`**).
`xmg:start_value_class(Class,Value)` exécute le code de la classe `Class` et unifie la liste des accumulations dans les dimensions avec `Value`. Le principe du prédicat est simplement de faire un appel aux prédicats `xmg:value_class` du code généré.

La définition du prédicat `xmg:start_value_class` n'est pourtant pas si triviale, puisque ce dernier nécessite des informations supplémentaires : puisque le code généré (soit les clauses de `xmg:value_class`) utilise les accumulateurs fournis par les `edcg`, il est impératif à la fois de déclarer les accumulateurs, de les propager dans le prédicat `xmg:value_class`), et de les initialiser lorsqu'il est appelé pour la première fois, soit dans `xmg:start_value_class`.

Dans la suite de cette section, nous détaillerons la méthode proposée pour que les accumulateurs soient facilement configurables.

7.7.2 Définir les accumulateurs

Différents types d'accumulateurs Un compilateur comporte un certain nombre de dimensions. Chacune de ces dimensions est associée à un accumulateur, encodé grâce aux EDCG. Cette accumulateur reçoit pendant l'exécution du code généré les contraintes exprimées dans la dimension.

Le mécanisme d'accumulation pour une dimension n'est cependant pas toujours le même. Par exemple, si pour la plupart des dimensions, l'accumulation correspond à l'extension d'une liste (initialement vide) par un nouvel élément, cela n'est pas le cas pour la dimension interface. Pour cette dimension, qui permet d'accumuler une structure de traits, l'accumulation correspond à l'unification d'une nouvelle structure de traits avec celle précédemment accumulée.

Déclarer un accumulateur Une dimension doit donc venir avec un type d'accumulateur et une opération standard sur cet accumulateur. La configuration d'une brique par ces informations se fait par le biais d'un fichier : le fichier *edcg.yap*. On peut se trouver dans trois situations différentes, suivant la spécificité de l'accumulateur à définir.

La plus spécifique des situations est la déclaration d'un nouvel accumulateur, pour lequel on donne les opérateurs et la valeur initiale. Pour la dimension interface (`<iface>`), par exemple, le fichier de configuration contient la déclaration suivante :

```
:- edcg:class( accu_type,   [edcg:set,edcg:get], _ ).
```

où `accu_type` est l'identifiant de l'accumulateur, et `edcg:set` et `edcg:get` les opérations de modification et d'accès à la valeur d'un accumulateur. Pour une contribution à la dimension `<iface>`, le code généré peut donc utiliser ces deux opérations. Enfin, le dernier argument indique que la valeur initiale de l'accumulateur est une variable libre.

La deuxième option est d'utiliser un type d'accumulateur prédéfini, comme dans le cas suivant (utilisé notamment dans la dimension `<syn>`) :

```
:- edcg:class_alias(accu_type, edcg:stack).
```

où `edcg:stack` désigne le type des piles (pour lesquelles l'opération par défaut est d'empiler).

Enfin, la troisième option, qui dans la pratique s'adapte à la plupart des cas, est de ne pas donner de fichier de configuration. Ceci implique l'utilisation de l'accumulateur par défaut, c'est à dire

la pile. Ceci convient donc a priori à toutes les dimensions pour lesquelles on construit un ensemble de contraintes.

7.7.3 Code généré

Lors de l'assemblage du compilateur (par le méta compilateur), les accumulateurs correspondants aux dimensions sont créés, et rendus disponibles dans `xmg:value_class`. Les dimensions, et donc les accumulateurs à déclarer, sont connus dès l'analyse de la description yaml (ce sont les étiquettes données en paramètre des constructeurs de brique `dim`). Le code de la figure 7.11 est généré automatiquement dans le cas d'un compilateur comprenant deux dimensions (une dimension syntaxique et une interface).

`edcg:thread` constitue la déclaration d'un accumulateur, avec son identifiant et son type. Les types `xmg_brick_syn_edcg:accu_type` et `xmg_brick_syn_edcg:accu_type` sont ceux que nous avons définis précédemment. `edcg:weave` permet de rendre disponible une liste d'accumulateurs (le premier paramètre) dans un ensemble de prédicats.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Threads initialization
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-edcg:thread(xmg_acc:trace, edcg:stack).

:-edcg:thread(xmg_acc:syn, xmg_brick_syn_edcg:accu_type).

:-edcg:thread(xmg_acc:iface, xmg_brick_iface_edcg:accu_type).

:-edcg:weave([xmg_acc:trace, xmg_acc:syn, xmg_acc:iface],
             [xmg:value_class/3]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Starting valuation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

xmg:start_value_class(Class, [trace-Trace,
                              syn-Syn,
                              iface-Iface]):--
  xmg:value_class(Class,_,_) with (xmg_acc:trace(Trace),
                                   xmg_acc:syn(Syn),
                                   xmg_acc:iface(Iface)
                                   ).
```

FIGURE 7.11 — Exemple de fichier `edcg.yap` généré

Cette étape (même si elle est effectuée à l'assemblage du compilateur, et non à la étape de génération de code) constitue la génération de code pour l'instruction **`value`** du langage méta-grammatical.

7.8 Exemple : XMG-1

Pour illustrer les concepts de l'approche sur un compilateur concret, cette section décrit l'assemblage du compilateur XMG pour grammaires d'arbres adjoints et sémantique avec des briques. La figure 7.12 présente le fichier `compiler.yaml` utilisé pour générer automatiquement toute la chaîne de traitement.

```
mg:
  _Stmt: control
  _EDecls: decls
decls:
  _ODecl: feats
control:
  _Stmt: dim_sem dim_syn dim_iface stardim
  _Expr: value
stardim:
  _Stmt: control
  proxy: dim_iface
avm:
  _Value: value
  _Expr: value
value:
  _Else: avm adisj avm.Dot control.Call
syn:
  _AVM: avm
iface:
  _AVM: avm
dim_syn:
  tag: "syn"
  Stmt: syn
  Expr: value
dim_sem:
  tag: "sem"
  Stmt: sem
  Expr: value
dim_iface:
  tag: "iface"
  Stmt: iface
  Expr: value
```

Figure 7.12 — L'assemblage d'un compilateur recréant le XMG existant

Deux briques centrales. Une partie du compilateur, celle permettant de faire des abstractions et de les combiner, reste assez fixe, puisque c'est autour d'elle que s'articule la modularité. Un compilateur métagrammatical basé sur un autre langage de base est tout de même envisageable (notamment dans le cas où les changement apportés sont purement syntaxiques). Nous division ce langage, constituant le coeur du compilateur, en deux briques.

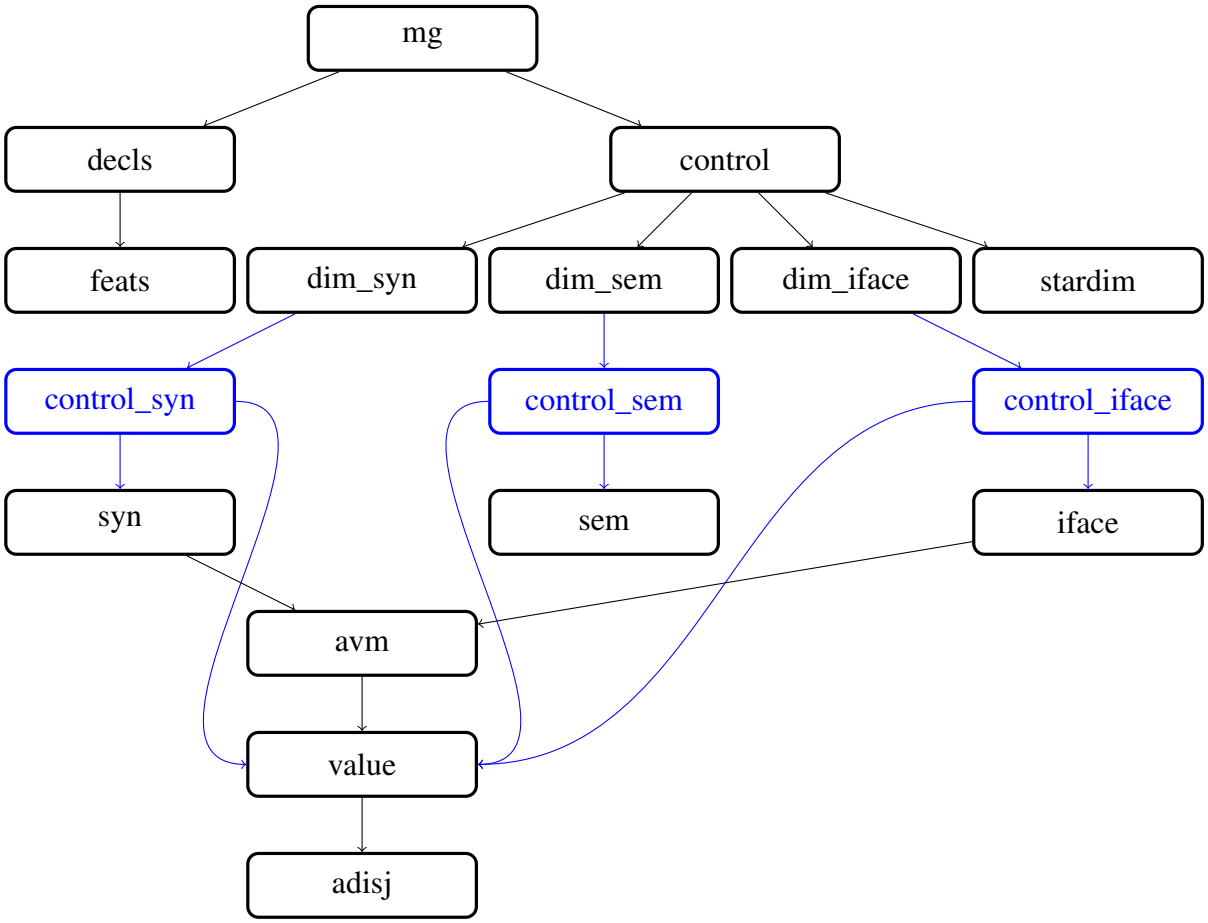


FIGURE 7.13 — Connexions entre briques

La première de ces briques, appelée brique métagrammaticale (`mg`), gère principalement la notion d'abstraction et la portée des variables. Elle contient le support pour les classes, les variables, les valuations, ainsi que des librairies permettant de représenter les accumulateurs ou de gérer les erreurs.

L'autre brique est la brique de contrôle (`control`), incluant essentiellement les opérateurs de conjonction et disjonction ainsi que les appels de classes. Ces briques sont séparées pour la raison suivante : si la brique métagrammaticale est unique, plusieurs instances du langage de contrôle dans un même langage peuvent se révéler utiles (le langage de contrôle est traditionnellement disponible à l'intérieur des balises de dimension, pour des raisons pratiques).

Connexions sur mg. La brique `decls`, branchée sur `mg`, permet les déclarations, soit la partie constante de la métagrammaire. Les déclarations de type sont fournies par la brique. Les autres déclarations, spécifiques au compilateur assemblé, doivent être ajoutées. C'est ce qui est fait avec la brique `feats`, qui contient le support pour la déclaration de traits typés.

La brique de langage de Contrôle est également branchée sur `mg`. Comme son nom l'indique, elle permet l'utilisation du langage de contrôle à l'intérieur d'une classe, c'est à dire la conjonction, la disjonction, et l'appel de classes. Les contributions aux dimensions sont rendues possibles par les briques que l'on branche sur le non terminal `_Stmt`.

Dimensions. Les deux briques de dimensions utilisées ici sont celles du langage d'arbre et du langage sémantique de XMG-1, apportées respectivement par les briques `syn` et `sem`.

La brique `avm` est paramétrable par le type de valeurs qu'elle peut recevoir. La brique value utilisée ici est également paramétrée par des valeurs additionnelles. En l'occurrence, ces valeurs additionnelles sont une avm (pour les structures de traits récursives) et la brique contenant le support pour les disjonctions atomiques.

Les briques de principes (`tree`, `colors` ...) n'apparaissent pas dans cet assemblage, puisqu'elles ne contribuent aucune partie du langage.

La figure 7.13 montre l'assemblage créé par le générateur de compilateur. Les briques et les connexions colorées en bleu sont celles n'apparaissant pas dans la description, mais dont les instances sont créées automatiquement (par les constructeurs de briques).

7.9 Conclusion

Dans ce chapitre, nous avons donné une vue concrète de l'intégration des briques proposées au chapitre 6 dans un processus de compilation. Pour chacune des étapes de ce processus, nous avons proposé une méthode pour décomposer le compilateur en parties élémentaires, tout en permettant son extensibilité. Pour synthétiser, les paragraphes qui suivent montrent concrètement comment les briques sont constituées.

Briques de compilateur. Une brique de compilateur (voir figures 7.14 et 7.15) se compose de trois parties : le fichier `lang.def` contenant la grammaire hors contexte du fragment de DSL défini dans la brique, un dossier `compiler` incluant un fichier contenant les règles propres à la brique pour chaque étape de la compilation, et un dossier `yaplib` réservé aux bibliothèques spécifiques. La figure 7.14 montre l'architecture de la brique de structures de traits. La figure 7.15 donne celle de la brique `syn`, qui fournit un fichier `edcg.yap`, spécifiant l'accumulateur à utiliser lorsqu'on utilise cette brique comme dimension.

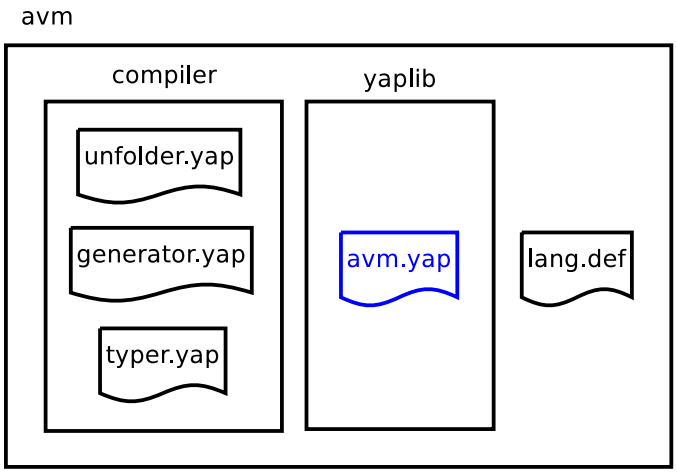


FIGURE 7.14 — L'architecture d'une brique (avm)

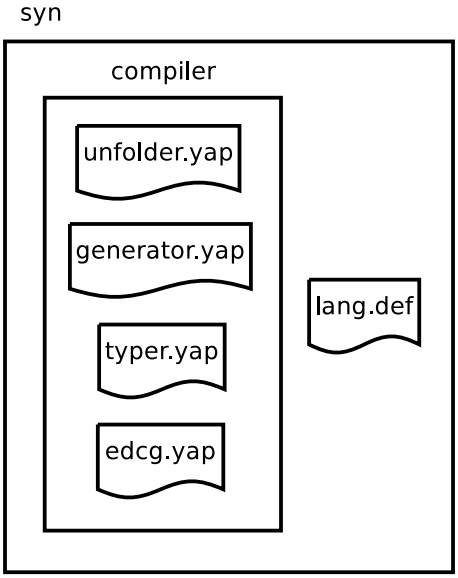


FIGURE 7.15 — L'architecture d'une brique (syn)

Briques d'assemblage. Une brique d'assemblage (voir figure 7.16) contient initialement un unique fichier de description : le fichier `compiler.yaml`. La construction du compilateur à partir de cette description repose sur l'utilisation du code contenu dans les briques de compilateur et la génération automatique de code. Les fichiers générés sont les suivants :

— Pour l'étape d'analyse, on génère les fichiers contenant les fichiers `tokenizer_keywords.yap` et `tokenizer_punct.yap`, fournissant respectivement les règles pour l'analyse lexicale des mots-clés du langage et de ses signes de ponctuation.
— Pour l'analyse syntaxique, le fichier `parser.yap` contenant les règles de l'analyseur, est créé.
— Toutes les briques figurant dans l'assemblage contribuent au compilateur avec l'ensemble de leurs modules de compilation, un fichier `loader.yap` visant à charger ces modules est généré.
— Le fichier `edcg.yap` évoqué en 7.7.3 et contenant les informations relatives aux accumulateurs à créer pour les dimensions, est également ajouté pendant la génération du compilateur.
— Enfin, le méta compilateur écrit dans le fichier `dimensions.yap` des prédicats associant à chaque identifiant de dimension (l'étiquette présentée en 7.6) la brique qui lui correspond.

La figure 7.16 représente l'ensemble des fichiers contenus dans la brique correspondant à l'assemblage du compilateur XMG-1.

Dans le chapitre suivant, l'objectif est de fournir une modularité similaire à celle présentée ici pour la seconde phase opérée par XMG, celle de génération des structures à partir du code généré par la compilation.

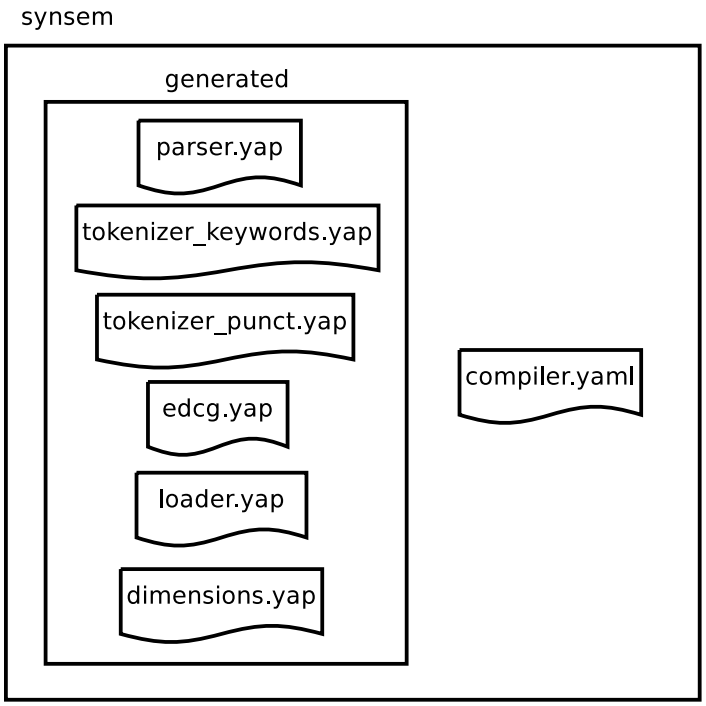


FIGURE 7.16 — Structure d'une brique d'assemblage (synsem)

Assemblage modulaire d'un générateur

8

Sommaire

8.1 Introduction **131**
8.1.1 Architecture du générateur 131
8.1.2 Exemple 132
8.2 Étape d'exécution **133**
8.3 Étape de résolution **134**
8.3.1 Solveurs et principes 134
8.3.2 Solveurs 135
8.3.3 Principes et plugins 135
8.3.4 Brique de solveur 135
8.3.5 Associer les solveurs aux dimensions 136
8.4 Étape de sérialisation **137**
8.4.1 Fonctionnement 137
8.4.2 Extensibilité 138
8.5 Conclusion **138**

8.1 Introduction

8.1.1 Architecture du générateur

Dans la deuxième phase de travail du méta exécuteur, le code généré, qui est non déterministe, est exécuté, à l'aide de la machine virtuelle Prolog à laquelle s'ajoutent des bibliothèques spécifiques au compilateur [1], pour produire des accumulations dans les dimensions.

1. Ces bibliothèques font parfois intervenir d'autres langages de programmation, comme c'est le cas pour C++ et Gecode pour la résolution de contraintes, par le biais de bindings.

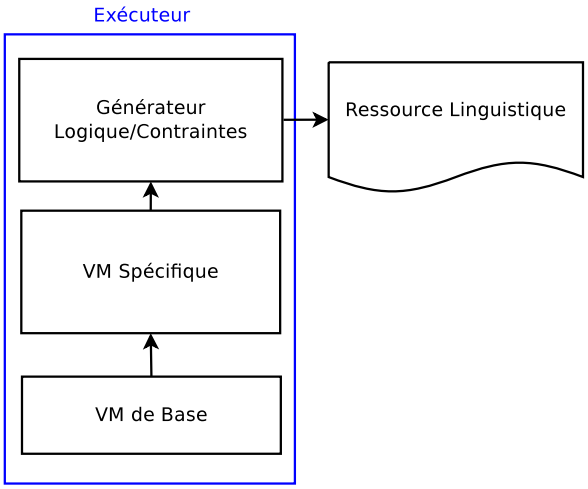


FIGURE 8.1 — L'exécuteur

Chaque accumulation est ensuite résolue de façon indépendante, pour produire un terme, correspondant à une solution du problème de satisfaction de contraintes exprimé, soit une nouvelle contribution pour la ressource.

Ces termes sont ensuite convertis dans un format de sortie par un troisième module. Le fichier résultant de ces conversions est le produit de la génération, soit la ressource langagière décrite par la métagrammaire.

On peut résumer cette phase avec le schéma suivant, laissant apparaître les trois étapes évoquées (accumulation, résolution puis sérialisation) :

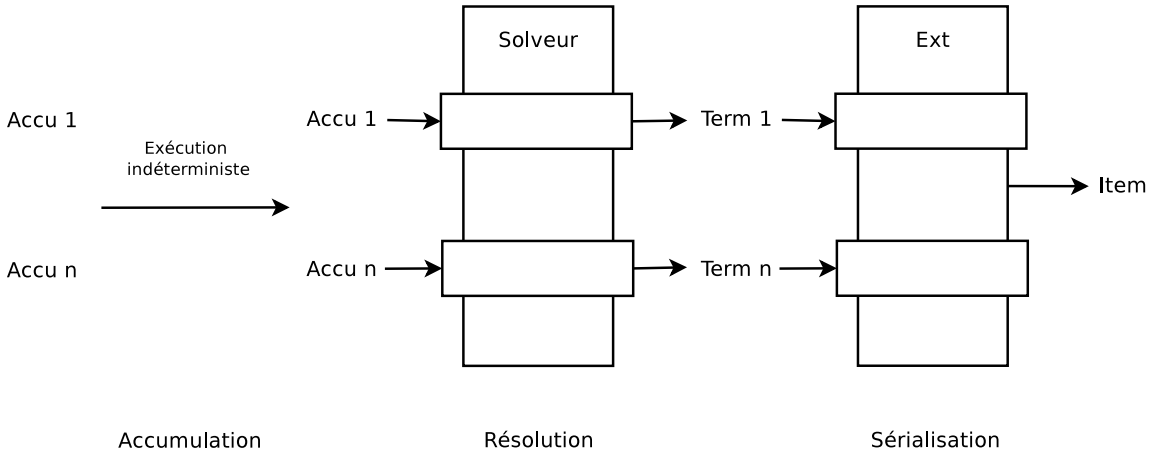


FIGURE 8.2 — L'architecture du générateur

8.1.2 Exemple

En guise d'exemple (illustré par la figure 8.3), l'exécution du code généré par XMG-1 pour les TAG produit toujours trois accumulations. La première correspond à une sous-spécification d'arbre (la dimension **<syn>**), la deuxième à une formule logique (**<sem>**), et la troisième à une structure de trait permettant l'interface entre les deux autres dimensions (`<iface>`).

Ces trois accumulations, en raison de leurs natures hétérogènes, doivent subir des traitements différents à l'étape de résolution. Pour la description d'arbre, on souhaite calculer tous les modèles minimaux, c'est à dire les modèles respectant toutes les contraintes de dominance et de précédence exprimés dans l'accumulation, et n'utilisant que les noeuds déclarés. Les deux autres résultats d'accumulation n'ont pas à être résolus, ils constituent une solution.

Une solution est donc un ensemble de trois termes, qu'il faut traduire dans le format de sortie. Chaque dimension doit donc fournir les règles de traduction pour les termes qu'elle produit.

Dans ce chapitre, nous nous inspirerons de la même idée de modularité que précédemment, celle donnée par les briques, pour modulariser les étapes de cette seconde phase.

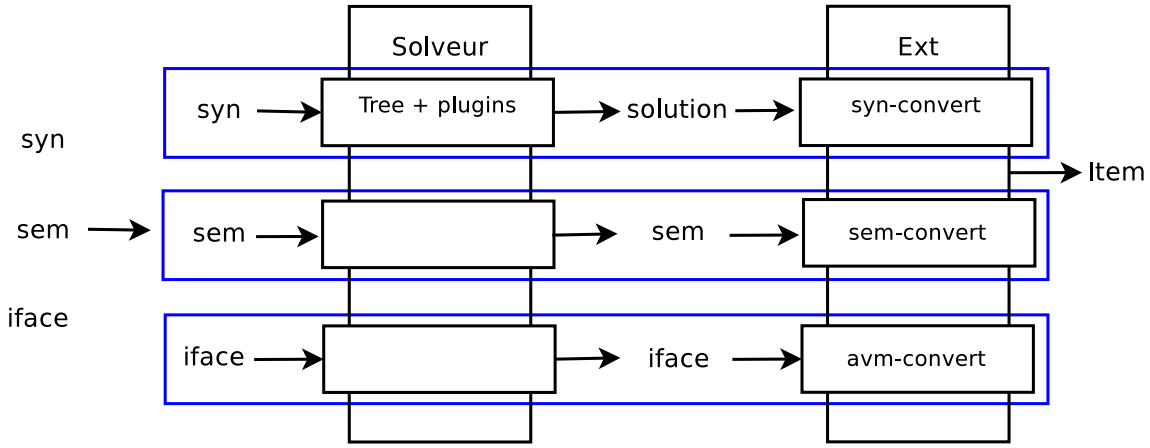


FIGURE 8.3 — L'architecture du générateur de XMG-1

8.2 Étape d'exécution

Si l'exécution du code généré consiste principalement à créer des accumulations, certaines structures décrites dans le langage métagrammatical doivent tout de même subir des opérations avant d'être accumulées. Les instructions d'accumulation, ainsi qu'une partie des instructions qui construisent les structures, sont supportées par la machine virtuelle Prolog (notamment lorsqu'elles sont basées sur l'unification classique, ou la construction de termes) et les EDCG. Certaines instructions du langage noyau font en revanche intervenir des mécanismes demandant d'enrichir le langage cible. Leur traduction par le générateur produit donc des instructions pour lesquelles le support doit être intégré d'autre part.

Prenons l'exemple des instructions générées dans la section 7.5. L'exécution de l'instruction `xmg_brick_avm_avm:avm(Var,[F-Var2])` doit modifier l'information associée à la variable **`Var`**, en lui ajoutant une contrainte (sur la valeur d'un trait).

C'est ici que la bibliothèque de variables attribuées pour les structures de traits (définie en 7.1.4) intervient. Cette bibliothèque est incorporée à la machine virtuelle par la brique des structures de traits.

Chaque brique vient avec un ensemble de bibliothèques, rassemblées dans le dossier *yaplib* de la brique. Quand un compilateur est construit à partir de cette brique, ses bibliothèques font partie des modules chargés, au même titre que les modules de compilation de cette brique. Les instructions pour lesquelles elle donne le support sont donc intégrées au langage noyau.

Concernant l'exemple présenté au chapitre précédent dans la figure 7.7, l'accumulation produite est la suivante :

```
[trace-[test],
 iface-IFace,
 syn-[dom(node(PX,FX,NX),->,node(PY,FY,NY)),
      dom(node(PX,FY,NX),->,node(PZ,FZ,NZ)),
      node(PX,FX,NX),
      node(PY,FY,NY),
      node(PZ,FZ,NZ)],
 sem-[]]
```

dans lequel toutes les variables apparaissant sont des variables attribuées : `IFace` est la structure de traits associée à la dimension `<iface>`, `PX`, `PY`, `PZ`, `FX`, `FY`, `FZ` sont respectivement les propriétés et les traits des noeuds `X`, `Y` et `Z`. `NX`, `NY`, `NZ` sont quant à eux les identifiants des noeuds (dont le mécanisme d'unification est la concaténation).

8.3 Étape de résolution

Une fois l'exécution terminée, les accumulateurs contiennent une description. Cette description peut être totalement spécifiée, et être équivalent à une solution. La description accumulée peut également être sous spécifiée, comme c'est le cas pour le langage de description d'arbres. Dans ce cas, elle est composée de relations de dominance et de précédence entre noeuds.

On ne peut pas considérer une accumulation de ce type comme un modèle, et une étape de résolution est nécessaire. Cette étape consiste à considérer l'accumulation comme les paramètres d'un problème sous contraintes. Les contraintes de ce problème se divisent en deux ensembles.

8.3.1 Solveurs et principes

Le premier ensemble de contraintes est propre à la description métagrammaticale, puisqu'il est constitué par l'accumulation. Chaque élément accumulé est une contrainte sur les modèles de la description. Le problème défini par ces contraintes forme une sous spécification de modèles.

Un autre ensemble de contraintes s'applique aux modèles qui seront ajoutés à la ressource : des contraintes de bonne formation assurant que les solutions produites appartiennent à la classe voulue (des arbres, des graphes, ...). À ces contraintes peuvent s'ajouter de nouvelles contraintes dépendantes du type de structures produites par la dimension. Elles correspondent à la notion de principes de XMG.

Le problème constitué de ces deux ensembles de contraintes est résolu par un solveur, le deuxième module de la phase de génération. Le solveur étant non-déterministe, on obtient pour chaque accumulation zéro ou plusieurs modèles sous la forme de termes.

La technologie principalement visée par la nature des accumulations étant celle des contraintes, il est possible, par le biais de bindings vers la bibliothèque C++ Gecode (http://www.gecode.org/), que nous avons développés spécifiquement pour XMG, de formuler des problèmes sous contraintes (notamment ensemblistes) de façon homogène au reste de la chaîne de compilation.

8.3.2 Solveurs

Les principes tels qu'ils étaient conçus dans XMG ne sont pas en totale adéquation avec la notion de modularité. Ces principes étaient très spécifiques à un formalisme ou à une approche donnée. Par exemple, le principe de couleurs permettant de filtrer les combinaisons de fragments d'arbres n'a de sens que dans un contexte où l'on manipule des arbres. Le solveur pour ce principe ne peut donc exister qu'en présence du solveur d'arbres.

Des solveurs plus généraux, paramétrables et combinables, sont nécessaires pour s'adapter à de nouveaux compilateurs. Ces solveurs se divisent en deux catégories. Le solveur de contraintes utilisé dans XMG-1, construisant tous les arbres modèles de la description accumulée, peut être considéré comme un solveur indépendant. Il est rendu disponible par la brique `tree`. De nouvelles dimensions modélisant des grammaires arborescentes (par exemple minimalistes) pourront ainsi disposer de ce solveur.

8.3.3 Principes et plugins

Les principes présent dans XMG-1, en revanche, dépendent de la notion d'arbre. Les briques `rank`, `colors`, et `unicity` fournissent les solveurs pour les principes de même noms, mais on une dépendance envers la brique `tree`. On appelle ces solveurs des plugins de solveur, puisqu'ils constituent des ensembles de contraintes supplémentaires à appliquer sur les descriptions.

Le découpage des solveurs offert par les plugins permet de faciliter l'extensibilité. Le développement d'un plugin consiste uniquement à poser de nouvelles contraintes sur le problème, sans nécessairement connaître précisément les contraintes déjà posées par le solveur sur lequel le plugin se branche.

8.3.4 Brique de solveur

Une brique de solveur ne repose pas sur le langage métagrammatical, puisque son langage d'entrée est le langage de contraintes accumulé dans les dimensions durant l'exécution du code. Une définition de langage par le fichier `lang.def` n'est donc pas nécessaire ici. La connexion d'une brique de solveur, et même son intégration au compilateur, doit être gérée différemment, puisque jusqu'ici, une connexion entre langages de descriptions était la seule façon de rendre une brique disponible.

Les briques correspondantes aux principes indépendants contiennent au moins trois modules de compilation, correspondant à trois étapes successives illustrées dans la figure 8.4.

Préparateur. Le premier de ces modules est le préparateur (`preparer.yap`), qui permet de passer de l'accumulation provoquée par l'exécution de la métagrammaire à une forme exploitable par un solveur.

Résolution. Le deuxième module est celui de résolution, qui déclare les variables du problème et pose les contraintes sur ces variables (`solver.yap`).

Extracteur. Enfin, un extracteur construit le modèle à partir d'une solution de la résolution (`extractor.yap`).

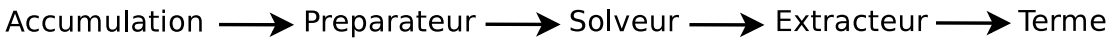


FIGURE 8.4 — Les étapes de la résolution

Architecture des briques de plugins. Les briques de plugins comprennent également un préparateur et un solveur. Leur préparateur doit comporter un prédicat `prepare/2`. Le premier argument correspond à l'accumulation avant préparation par le plugin. Le second est un terme `prepared(Plug,Acc)` contenant les contraintes extraites par le plugin (`Plug`) et l'accumulation initiale éventuellement modifiée (`Acc`). Le solveur des briques de plugins doivent contenir un prédicat `post`, appelé automatiquement pendant la pose des contraintes du solveur duquel le plugin dépend (si le principe est activé en tous cas). Les étapes de la résolution comprenant les plugins se schématise de la façon suivante :

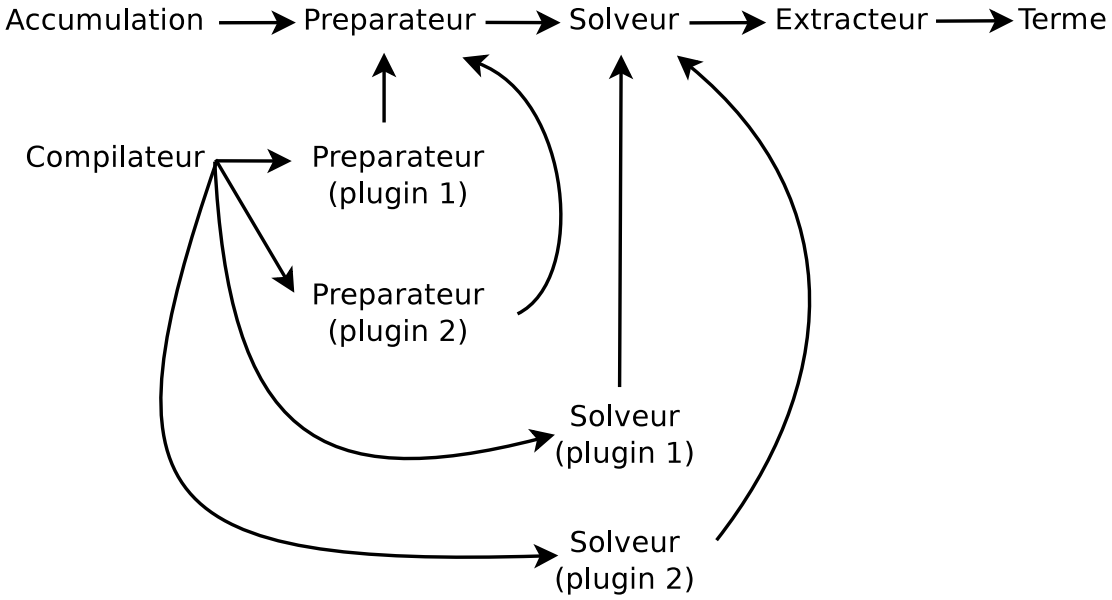


FIGURE 8.5 — Les étapes de la résolution avec plugins

Les préparateurs et solveurs de plugins sont activés par des déclaration dans la métagrammaire (les principes), et contribuent leurs règles au préparateur et au solveur principal.

8.3.5 Associer les solveurs aux dimensions

Comme nous l'avons vu en 7.6, la description des dimensions dans le fichier `lang.def` passe par l'utilisation d'un constructeur de briques, en passant en paramètres une étiquette correspondant à l'identifiant de la dimension, ainsi que le langage utilisé dans cette dimensions. C'est par ce même moyen que nous associons aux dimensions les solveurs souhaités.

```
dim_syn:
  tag: "syn"
  solver: "tree"
  Stmt: syn
  Expr: value
```

Ici, la dimension `syn` se voit associé le solveur `tree`. Tous les plugins de ce solveur sont rendus disponibles pour une activation dans la métagrammaire.

8.4 Étape de sérialisation

8.4.1 Fonctionnement

La génération d'une ressource linguistique n'est en général qu'une étape d'une chaîne de traitement plus complète. L'objectif final peut être d'utiliser la ressource produite pour de l'analyse du langage (en la passant en paramètre d'un parser par exemple), ou encore pour de la génération. Ces opérations sont réalisées par des outils externes au compilateur, l'objectif est d'offrir suffisamment de modularité dans le format de la ressource pour qu'elle soit adaptable à ces outils, éventuellement déjà existants.

Chaque production du solveur est un terme, représentant un modèle d'une accumulation de contraintes. Ce terme doit être converti dans un format exploitable par un outil. L'utilisation de langages tels que XML et JSon est une solution traditionnelle pour faire communiquer des applications dans un environnement hétérogène tel que le nôtre.

L'écriture d'un convertisseur en format de sortie suit le même modèle que l'écriture d'un unfolder ou d'un générateur de code. Le principe est de donner des règles pour tous les types de termes produits par le solveur de la brique, par l'intermédiaire du prédicat `xmg:xml_convert_term`.

La brique correspondant au solveur d'arbres fournit les règles de traduction du terme représentant le modèle vers XML :

```
xmg:xml_convert_term(tree:tree(Node,Trees), Root):--
  Node=node(PropAVM,FeatAVM,N),
  xmg_brick_avm_avm:avm(PropAVM, Props),
  xmg_brick_avm_avm:avm(FeatAVM, Feats),
  xmg_brick_avm_avm:const_avm(FeatAVM,CAVM),
  xmg:xml_convert(syn:props(Props),props(Name,N,XMLMark)),
  (
    var(CAVM)->
    (
      xmg:convert_new_name('@AVM',CAVM),
      xmg:xml_convert(avm:avm(Feats),XMLFeats),!,
      Root=
        elem(node, features([type-XMLMark, name-Name]),
             children([elem(narg,
                            children(Children))|Trees1])),
      Children=[elem(fs,features([coref-CAVM]),
                children(XMLFeats))]
    )
    ;
    (
      !,
      Root=
        elem(node, features([type-XMLMark, name-Name]),
```

```
                children([elem(narg,
                                children(Children))|Trees1])),
        Children=[elem(fs,features([coref-CAVM]))]
      )
    ),
    xmg:xml_convert(Trees,Trees1),!.
```

où `Node` est la racine de l'arbre passé en paramètre, `PropAVM` `FeatAVM` et `N` des variables attribuées correspondantes aux propriétés, aux traits et à l'identifiant de cette racine. Les attributs de ces variables sont respectivement `Props`, `Feats` et **`Name`**. `CAVM` est une variable associée à une structure de trait, et instanciée seulement si la structure apparaît déjà dans la sortie (On a alors à faire à une co-référence). `Root` est le terme construit par le traducteur.

8.4.2 Extensibilité

Pour chaque type de terme, la conversion est réalisée par le biais du même prédicat, `xmg:xml_convert_term`. Pour étendre un compilateur avec un nouveau solveur, il est nécessaire d'écrire une clause de ce prédicat pour chaque terme qu'il peut produire.

8.5 Conclusion

Dans ce chapitre, nous avons proposé une approche modulaire pour le développement des modules de la seconde phase opérée par XMG, celle de génération des ressources à partir du code compilé. Si l'extensibilité est moins immédiate que pour les modules de compilation, puisqu'elle consiste à écrire des solveurs ou des librairies pour les variables attribuées, l'aspect composable est toujours facilité par l'utilisation de briques. La notion de plugins permet de composer de nouveaux jeux de contraintes et de les intégrer à un solveur. Ces contraintes s'activent d'après les déclarations de principes dans la métagrammaire.

Même si l'extensibilité est plus compliquée à atteindre pour la phase de génération en comparaison avec la phase de compilation, notre solution est adaptable (sans extension) à de nombreux travaux de génération de ressources. Le solveur d'arbres peut être utilisé dans tous les formalismes manipulant sur ces structures (LFG par exemple). Nombreuses sont également les structures qui ne nécessitent pas d'être résolues : les structures de traits (omniprésentes dans les formalismes grammaticaux), les sous-spécifications manipulées dans les grammaires basées sur la théorie des modèles (soit des ensembles de contraintes) ou encore les prédicats utilisés dans la dimension sémantique de XMG-1.

Nous terminons ce chapitre par une vue concrète de l'architecture des briques de solveur proposées, ainsi que de celle des briques de compilateur définies au chapitre précédent (et enrichies avec les modules propres à la phase de génération).

Briques de solveur. Comme pour la phase de compilation, les briques de solveurs apparaissant dans le fichier de définition du compilateur `compiler.yaml` apportent l'ensemble de leurs modules de génération : un préparateur, un solveur, un extracteur, et un sérialiseur. Le chargement de ces modules est ajouté au fichier `loader.yap`. Un exemple d'architecture de brique de solveur est proposé dans la figure 8.6. Les briques de plugins contiennent quant à elles un sous ensemble de ces fichiers (un préparateur et un solveur).

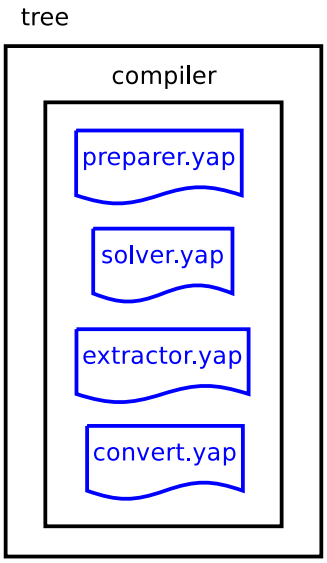


FIGURE 8.6 — Structure d'une brique de solveur (tree)

Briques de compilateur. Les autres briques (celles contenant un fichier `lang.def`) apportent pour cette phase leurs bibliothèques spécifiques, rassemblées dans le dossier `yaplib`. Elles peuvent également contenir un module de sérialisation pour ses structures (`convert.yap`). La figure 8.7 montre l'architecture de la brique `syn`, qui contribue au générateur avec des bibliothèques spécifiques : `engine.yap` contient le support pour les représentations de noeuds syntaxiques, et `nodename.yap` et `most.yap` sont des bibliothèques de variables attribuées gérant l'unification entre les identifiants des noeuds et celle entre les relations.

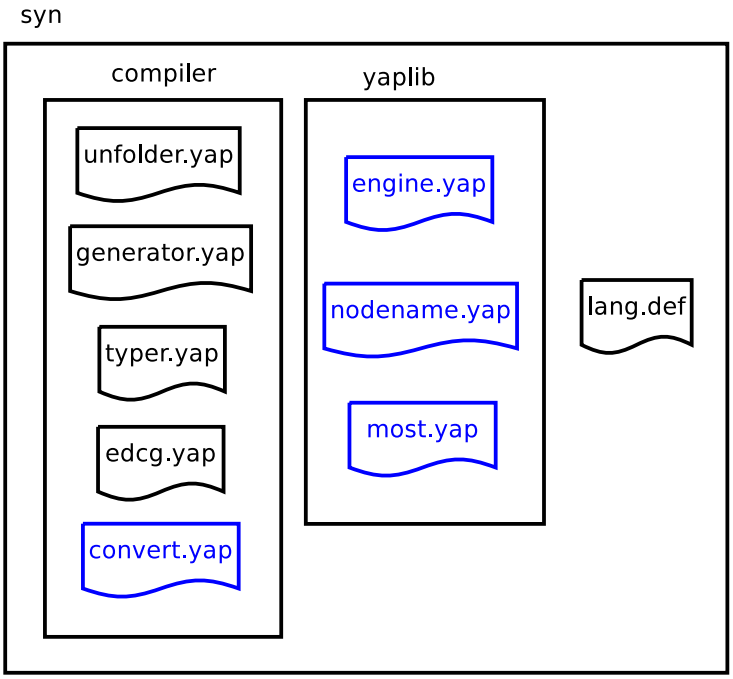


FIGURE 8.7 — L'architecture d'une brique (syn)

Briques d'assemblage. Le méta compilateur crée également un fichier `solvers.yap` où figurent les associations entre les dimensions et les solveurs, et qui s'ajoute aux fichiers générés automatiquement pour la phase de compilation. L'exemple de la brique d'assemblage `synsem` est donné dans la figure 8.8.

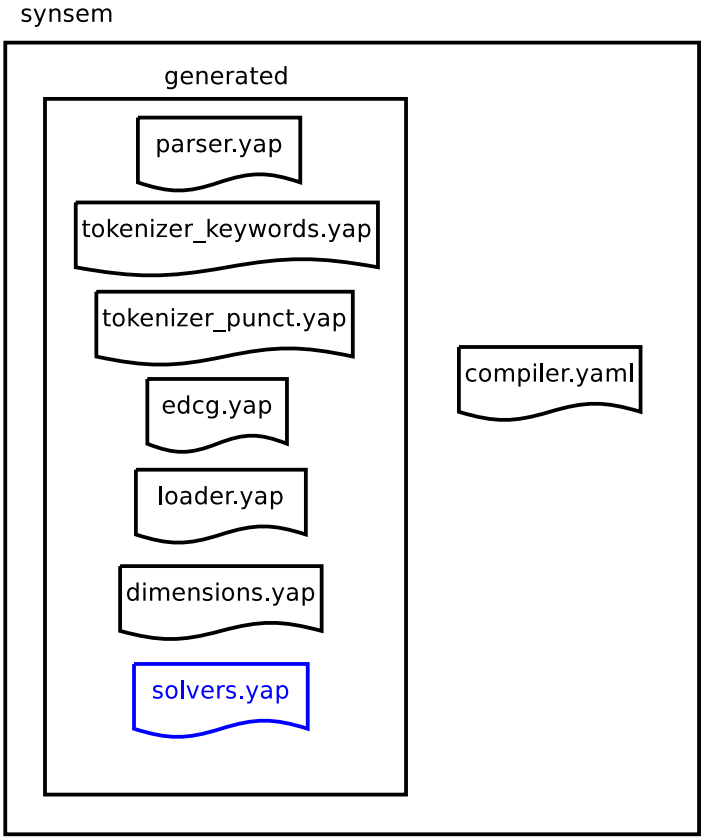


FIGURE 8.8 — Structure d'une brique d'assemblage (synsem)

Dans le chapitre suivant, nous détaillerons les outils rendant l'extensibilité du méta exécuteur possible et facilement accessible, quel que soit le profil de l'utilisateur.

Comparaison et conclusion 9

Sommaire

9.1 Introduction **141**

9.2 Comparaison **141**

9.2.1 Similitudes architecturales 141

9.2.2 Comparaison contrastive 143

9.3 Conclusion **145**

9.1 Introduction

La section 9.2 vise à placer notre approche relativement aux approches existantes et comparables pour la tâche de génération modulaire de DSL.

Nous choisissons pour effectuer cette comparaison l'outil Neverlang ([Vacchi *et al.*, 2013], présenté dans la section 4.4), qui est à notre connaissance le plus proche sous de nombreux aspects. La solution apportée par JastAdd par exemple ne semble pas adaptée pour effectuer de grands changements dans le compilateur (l'idée est de recréer Java et de faciliter son extensibilité).

Enfin, la section 9.3 contient la conclusion de la partie.

9.2 Comparaison

9.2.1 Similitudes architecturales

Débutons par une comparaison des approches au niveau des architectures des chaînes de traitements qu'elles proposent. Rappelons tout d'abord le fonctionnement de XMG (figure 9.1).

XMG. La métacompilation permet de générer, à partir d'un fichier de spécification et de briques choisies dans une bibliothèque, un compilateur et une machine virtuelle spécifique. Ces deux éléments forment, lorsqu'on leur ajoute la machine virtuelle Prolog, un méta exécuteur. Cet outil prend en entrée une description métagrammaticale, et génère la ressource linguistique qu'elle décrit. La méta exécution se divise en deux phases. La première est la compilation (dont les étapes ont été assemblées par le métacompilateur). La seconde est la génération, consistant à exécuter le code compilé à l'aide de la machine virtuelle spécifique assemblée.

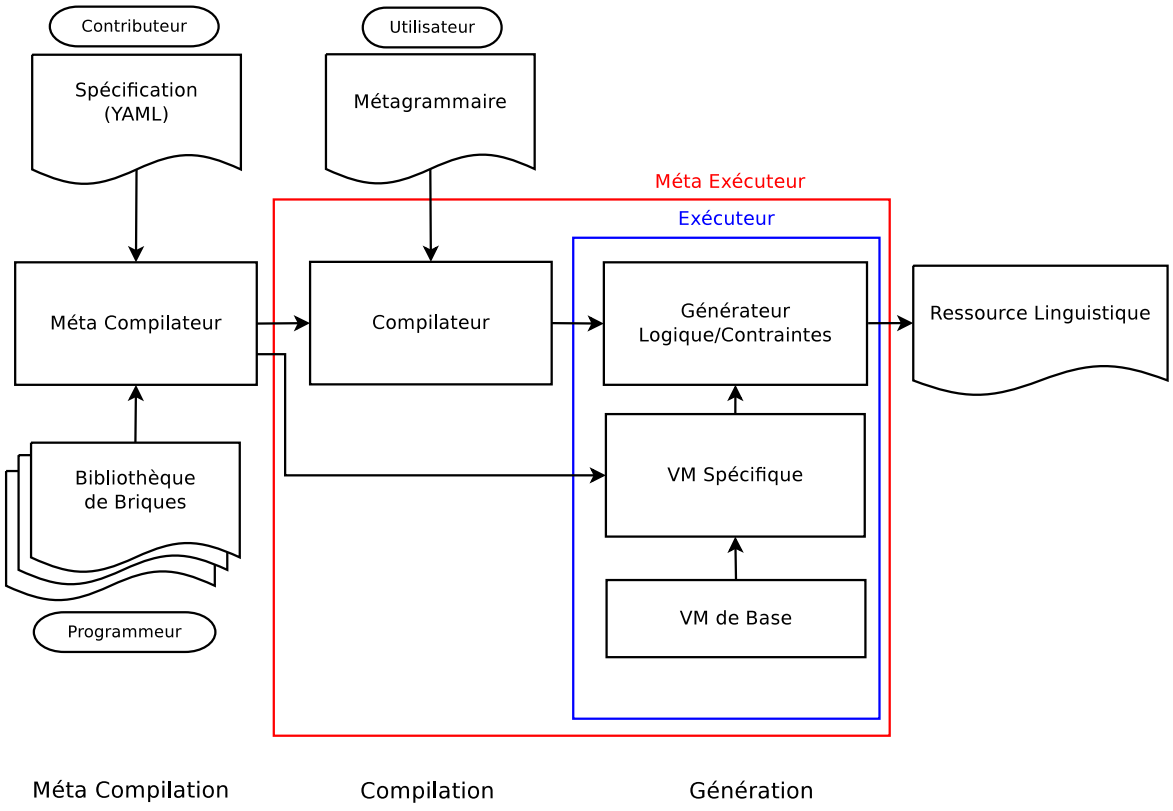


FIGURE 9.1 — L'architecture modulaire de XMG

Neverlang. L'architecture modulaire de Neverlang est présentée dans la figure 9.2.

La première des similitudes avec la chaîne présentée pour XMG est la présence d'une phase de métacompilation, ici réalisée par la boîte `Neverlang`. La métacompilation prend en entrée une description du langage (comparable à la spécification yaml de MXG), cette description piochant dans une bibliothèque de slices (comparables aux briques de XMG).

La première production de la métacompilation est un parser, que l'on peut assimiler au compilateur généré par la métacompilation de XMG. La phase de compilation dans laquelle intervient le parser construit l'arbre de syntaxe abstraite d'un programme passé en entrée. L'arbre de syntaxe abstrait correspond dans la chaîne de traitement que nous proposons au programme logique généré.

La seconde production de Neverlang est une suite de rôles sémantiques. Ces rôles accomplissent la phase exécution, en parcourant les uns après les autres l'arbre de syntaxe abstraite. L'exécuteur composé par ces rôles est comparable à la machine virtuelle spécifique assemblée par XMG.

Unités élémentaires. Les unités élémentaires apparaissant dans les assemblages de compilateurs des deux outils (les slices et les briques) sont comparables sous certains aspects :

— ils contiennent une description du langage (la *concrete syntax* de Neverlang et le fichier `lang.def` de XMG)
— cette description est donnée par une grammaire hors contexte, sous forme BNF
— on associe aux règles syntaxiques des règles sémantiques

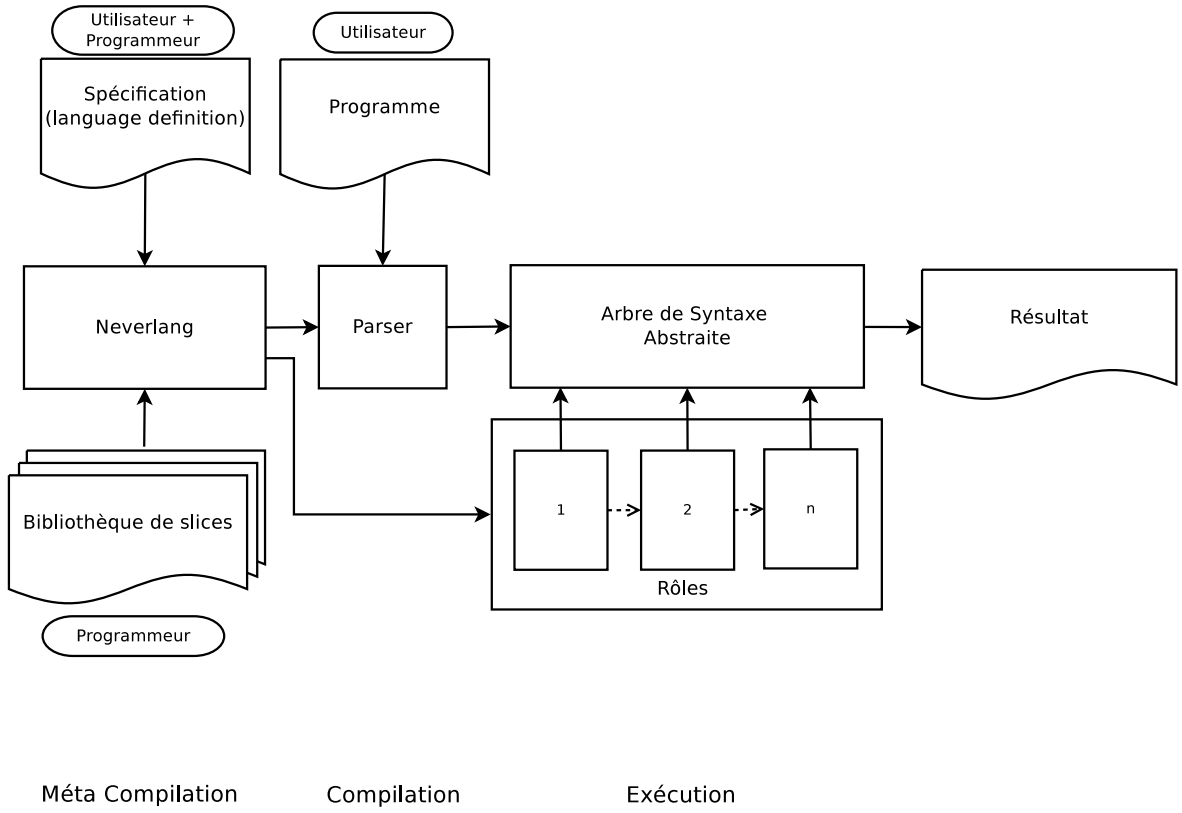


FIGURE 9.2 — L'architecture modulaire de Neverlang

9.2.2 Comparaison contrastive

Nous avons noté un nombre important de convergence entre les chaînes de traitement de XMG et de Neverlang. Malgré ceci, les deux approches divergent de façon significative sous plusieurs aspects.

DSL visés. Les langages que visent les deux outils sont fondamentalement différents, et cette différence explique la plupart des différences méthodologiques.

Neverlang permet d'assembler des langages impératifs, dans la tradition de Java. L'outil étant basé sur la JVM, sa dépendance au langage (ou à la famille de langages) est forte. C'est donc un langage d'instructions qui est construit.

Le code généré par XMG s'inscrit quant à lui dans le domaine de la programmation logique et par contraintes. L'objectif est de décrire des structures contraintes. Le code généré est du code Prolog, et changer ce langage cible semble difficilement réalisable à moins d'importants efforts.

Chaîne de traitement. La chaîne de traitement de XMG, de par les tâches de génération de ressources qu'elle vise, s'articule en deux phases. Là où Neverlang propose une chaîne de traitement unique dont le nombre et l'ordre des étapes sont configurables, XMG compile dans un premier temps la description métagrammaticale en programme, puis exécute ce programme en suivant de nouvelles étapes (elles aussi assemblées modulairement). Il est également possible d'agir les étapes de cette deuxième phase directement au travers de la métagrammaire, par le

biais de l'activation de principes. Cette modularité, offerte par les plugins de solveurs, constitue une contribution originale de notre approche.

Cette différence dans la technologie est due aux types de DSL visés. La deuxième phase de traitement de XMG ne fait sens que dans le cas où l'on souhaite manipuler des contraintes. De même, la configuration et le nombre des étapes importent peu dans le cas de la génération des programmes logiques et par contraintes de XMG : l'arbre de syntaxe abstraite est modifié, produisant une nouvelle construction à chaque étape. Augmenter le nombre de ces étapes ne ferait qu'augmenter la complexité du développement.

Unités élémentaires. Si nous avons noté les similarités entre les unités élémentaires proposées dans les deux approches, la manière d'associer syntaxe et sémantique diffère. Dans Neverlang, syntaxe et sémantique sont définis indépendamment, puis associés dans la slice. Dans XMG, les actions sémantiques sont attachées aux règles.

Neverlang offre donc la possibilité d'offrir plusieurs syntaxes pour une même sémantique, en définissant simplement une nouvelle syntaxe, avec de nouveaux mots clés par exemple, et l'associant au même module sémantique. Faire quelque chose de comparable avec XMG revient à utiliser un constructeur de briques paramétré par un ensemble de mots clés.

La seconde différence impliquée par l'indépendance de la syntaxe et de la sémantique est la possibilité avec Neverlang de définir plusieurs phases de traitement pour la syntaxe. Ceci est rendu possible par le fait que toutes les phases de la compilation travaillent sur l'arbre de syntaxe abstraite sans le modifier. XMG opère quant à lui sur des arbres successifs, utilisant différents langages.

Assemblage. L'assemblage d'un DSL avec les deux outils consiste à rassembler des unités élémentaires. Dans le cas d'XMG, le fichier `compiler.yaml` décrit un assemblage de brique en spécifiant des connexions entre ces dernières. L'exécution du métacompilateur sur cette description assemble l'intégralité des phases de compilation pour le DSL. Le fait de spécifier les connexions, de pouvoir instancier plusieurs fois une même brique en la paramétrant de différentes façons, donne une flexibilité importante

Du côte de Neverlang, la définition de langage consiste à lister l'ensemble des slices à utiliser. Les connexions entre les slices se font en revanche implicitement : les non-terminaux utilisés dans les règles doivent au final tous posséder une règle de production, éventuellement fournie par une autre slice. Ceci donne lieu à la construction d'un graphe de dépendances, qui doivent être résolues, tâche nécessitant une certaine expertise (la résolution consiste à créer un modèle de variation, ou VAM).

Intégration de différents profils d'utilisateurs. L'un des objectifs de l'approche modulaire de XMG est la volonté d'intégrer un type d'utilisateur intermédiaire entre le programmeur expert et l'utilisateur linguiste. Pour cette raison, le point d'entrée du métacompilateur traite une description (le fichier `compiler.yaml`) réalisable sans connaître en détail la structure interne des briques. Dans un assemblage de compilateur Neverlang, l'utilisateur doit collaborer avec un programmeur expert pour résoudre les dépendances créées par l'assemblage de slices.

Pour résumer, les principales différences que nous avons notées entre les approches sont dues au domaine d'application visé par les outils. L'assemblage modulaire des unités élémentaires proposé par XMG est uniquement basé sur la description donnée par le fichier `compiler.yaml`,

ce qui rend la tâche plus intuitive pour un utilisateur ayant le profil assembleur (ou contributeur). Neverlang offre quand à lui à cadre plus unifié : toute la chaîne de traitement (mise à part l'analyse syntaxique) est contenue dans les rôles sémantiques, articulés en phases, et associés à la syntaxe par le biais des slices.

9.3 Conclusion

Dans cette partie, nous avons présenté une méthode pour le développement de nouveaux compilateurs pour la génération de ressources langagières. Cette méthode est basée sur la modularité, et rend possible l'assemblage de nouveaux cadres métagrammaticaux à partir de briques existantes, tout en facilitant l'extensibilité à de nouvelles applications.

Nous avons analysé les différents niveaux où l'on pouvait tirer partie de la modularité, dans les phases de compilation et de génération, et donné les méthodes utilisées pour son implantation.

Dans la partie qui suit, nous proposerons différentes extensions du compilateur, permettant de traiter des niveaux de représentation tels que la morphologie ou un langage sémantique basé sur les structures de traits.

Troisième partie

Application de l'assemblage modulaire de DSL à la description de ressources linguistiques

Introduction

Dans cette partie, nous montrons l'apport du système aux trois profils d'utilisateurs évoqués au chapitre 5. Nous présentons donc pour chacun de ces profils des exemples concrets d'utilisation du système. Chaque chapitre se consacre à un niveau de représentation linguistique et détaille successivement :

— i) la problématique linguistique et sa formalisation en un langage dédié, soit l'apport au profil utilisateur
— ii) la conception des briques nécessaires à cet assemblage, soit le travail du profil programmeur
— iii) l'assemblage d'un compilateur pour ce langage, soit la tâche attribuée au profil contributeur

Dans le chapitre 10, nous nous intéresserons aux deux formalismes pour lesquels XMG-1 offrait un cadre de développement : les grammaires d'arbres adjoints, et les grammaires d'interaction. Dans le chapitre 11, nous présenterons une approche métagrammaticale à une théorie morphologique, celle des champs topologiques, et son application pour la description d'une langue Bantoue : l'Ikota. Enfin, dans le chapitre 12, nous présentons l'assemblage d'un DSL permettant la manipulation de structures de traits typées pour la description de la sémantique.

Application à la syntaxe 10

Sommaire

10.1 Grammaires d'Arbres Adjoints **151**
10.1.1 Problématique linguistique 151
10.1.2 Création de briques 152
10.1.3 Assemblage du compilateur 154
10.2 Grammaires d'Interaction **154**
10.2.1 Problématique linguistique 154
10.2.2 Création de briques 156
10.2.3 Assemblage du compilateur 157
10.3 Conclusion **158**

Dans le but de valider l'approche proposée, la première application réalisée concerne la compilation de grammaires d'arbres. Concrètement nous souhaitons assembler modulairement la chaîne de traitement correspondant à XMG-1, ce qui permettra en outre d'assurer la pérennité des ressources qui ont été développées avec XMG-1.

En section 10.1, après avoir brièvement présenté la problématique linguistique (définition modulaire de grammaires d'arbres adjoints), nous présentons le DSL que nous cherchons à produire (le langage XMG-1). Finalement, nous donnons la méthode pour assembler le compilateur correspondant à ce DSL.

Dans un deuxième temps (section 10.2), nous appliquons la même approche à une variante de ce DSL, utilisé pour décrire des grammaires d'interaction [Guillaume et Perrier, 2009].

10.1 Grammaires d'Arbres Adjoints

10.1.1 Problématique linguistique

Comme nous l'avons vu en section 3.3 (page 35), les grammaires d'arbres adjoints ont inspiré de nombreuses approches métagrammaticales, et ont été le premier formalisme syntaxique géré par XMG. En effet, Le gain offert par les abstractions de XMG pour décrire des formalismes grammaticaux présentant un grand niveau de redondance (cas de TAG) est important. XMG a ainsi permis de développer des grammaires de grande taille ([Crabbé, 2005], [Kallmeyer *et al.*, 2008] , [Gardent, 2008], par exemple).

Au delà de ce gain au niveau du coût de développement, l'approche métagrammaticale permet d'appréhender la ressource linguistique différemment. Il s'agit non seulement de factoriser des structures, mais de bâtir ces factorisations sur des critères linguistiques. Vue ainsi, la métagrammaire ne constitue pas seulement une formalisation mais une ressource linguistique en elle même.

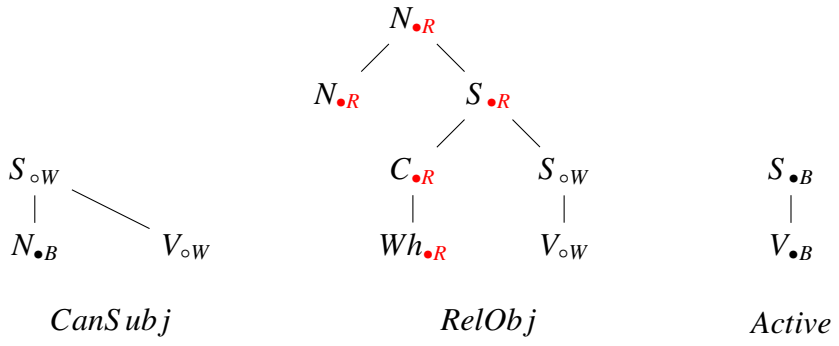


FIGURE 10.1 — Arbres élémentaires colorés

En suivant la méthodologie proposée par [Crabbé, 2005], le linguiste ne décrit pas des arbres TAG, mais des fragments d'arbres. Les descriptions de fragments d'arbres sont ensuite combinées à l'aide des opérateurs logiques puis résolues pour produire la grammaire ressource.

La figure 10.1 illustre cette combinaison de fragments. Rappelons que les couleurs sont des contraintes servant à guider les identifications entre noeuds, concrètement, elle agissent comme filtre sur les modèles possibles.

Le DSL à assembler pour cette tâche de description est celui que nous avons détaillé en sous-section 3.3.4 (page 41). Il est composé du langage de contrôle (offrant les opérateurs de conjonction, de disjonction, ainsi que les appels de classes), ainsi que d'un langage de description d'arbres (déclaration de noeuds et de contraintes entre les noeuds). Ces deux langages se formalisent comme suit :

$$\begin{array}{rcl} Classe & := & Nom[p_1, \ldots, p_n] \rightarrow Contenu \\ Contenu & := & \langle Dim \rangle \{Desc\} \mid Nom[\ldots] \mid Contenu \vee Contenu \\ & & \mid Contenu \wedge Contenu \end{array}$$

$$\begin{array}{l} Desc := x \rightarrow y \mid x \rightarrow^{+} y \mid x \rightarrow^{*} y \mid x \prec y \mid x \prec^{+} y \mid x \prec^{*} y \mid x[f{:}E] \\ \qquad \mid x(p{:}E) \mid Desc \wedge Desc \end{array}$$

La syntaxe concrète du DSL a été donné figure 10.2 propose un exemple d'utilisation de la syntaxe concrète de ce langage pour traduire les fragments de la figure 10.1.

Dans la section qui suit, nous allons voir comment intégrer les phases de compilation liées à ce DSL dans des briques.

10.1.2 Création de briques

En section 7.8, nous avons vu comment assembler modulairement le DSL XMG-1. Il nous reste cependant à décrire la conception de la brique `stardim`. Cette brique a été ajoutée pour assurer

```
class CanSubj
{
  <syn>{
    node (color=white)[cat=s]{
      node (color=black)[cat=n]
      node (color=white)[cat=v]
    }
  }
}

class RelObj
{
  <syn>{
    node (color=red)[cat=n]{
      node (color=red)[cat=n]
      node (color=red)[cat=s]{
        node (color=red)[cat=c]{
          node (color=red)[cat=wh]
        }
        node (color=white)[cat=s]{
          node (color=white)[cat=v]
        }
      }
    }
  }
}

class Active
{
  <syn>{
    node (color=black)[cat=s]{
      node (color=black)[cat=v]
    }
  }
}
```

Figure 10.2 — Code XMG pour les arbres élémentaires colorés

la compatibilité totale avec XMG-1. Son but est de rendre disponible l'opérateur ***=**, qui se place entre deux contributions à des dimensions. La première de ces contributions concerne une dimension parmi **<syn>** et **<sem>** dans XMG-1, et la seconde concerne une structure de traits appelée interface. Rappelons que l'interface est traditionnellement utilisée pour partager des informations entre les dimensions (concept d'extension d'espaces de noms). Un exemple d'utilisation de l'opérateur ***=** est le suivant, dans lequel l'interface permet de rendre accessible la valeur associée à la variable locale `?Y` via l'identifiant global `suj`.

```
<syn>{
```

```
 node ?X [label=?Y]
}*=[suj=?Y]
```

stardim est, comme dim, un constructeur de brique. Ses paramètres sont la dimension rendue accessible par l'opérateur (proxi), et le type de littéraux auxquels l'opérateur peut s'attacher (Stmt). Plus concrètement, le langage que permet de reconnaître une brique construite par stardim peut se décrire de la façon suivante :

```
Star : _Stmt '*=' _proxi
       {$$=control:and($1,dim:dim(id_proxi,$3))} ;
```

dans laquelle id_proxi est l'identifiant de la dimension passée en paramètre (proxi). Deux règles de production sont ajoutées : les deux non-terminaux externes _proxi et _stmt sont respectivement réécrits en les axiomes des briques proxi et Stmt passées en paramètres.

Dans l'assemblage du DSL XMG-1, nous paramétrerons stardim par la dimension interface, qui est par ailleurs accessible comme les autres dimensions via la balise <iface>, et lui permettons de s'attacher à tout élément du langage de contrôle.

10.1.3 Assemblage du compilateur

L'assemblage que nous utilisons pour recréer le compilateur TAG de XMG-1 est présenté dans la figure 10.3.

Cet assemblage a été validé sur les grammaires de grande taille existantes (la ressource produite est identique à celle qui l'était par XMG-1). De nouvelles ressources ont également commencé à être développées avec le compilateur assemblé modulairement (notamment pour le São Tomense [Schang *et al.*, 2012], le Gwada, l'Ikota, le serbe). [1]

10.2 Grammaires d'Interaction

10.2.1 Problématique linguistique

Les Grammaires d'Interaction ([Perrier, 2000]) sont le second formalisme syntaxique pour lequel XMG-1 permet le développement de grammaires. La grammaire d'interaction du français ([Perrier, 2007]) est une grammaire couvrante qui est élaborée en utilisant XMG.

Une grammaire de ce type n'est pas constituée d'arbres, mais de *descriptions d'arbres*. Ces dernières correspondent à des arbres sous-spécifiés (des contraintes de dominance et de précédence entre noeuds). Ceci place les grammaires d'interaction dans la catégorie des approches basées sur la théorie des modèles (cf. section 3.4).

Une caractéristique des grammaires d'interaction est qu'elles intègrent aux descriptions syntaxiques des étiquettes de noeuds sous forme de structures de traits polarisées. On peut noter que ces traits polarisés forment des contraintes supplémentaires sur les modèles (elles contraignent l'unification entre structures de traits lors du calcul des modèles). Ces polarités sont très similaires aux notions de besoins et de ressources de la métagrammaire de Gaiffe (cf. sous-section 3.3.3), ou au langage de couleurs de XMG (cf. sous-section 3.3.4). La différence majeure est que ces polarités guident l'analyse syntaxique et non la construction de la grammaire. En d'autres

1. collaborations en cours

```
mg:
  _Stmt: control
  _EDecls: decls
decls:
  _ODecl: feats
control:
  _Stmt: dim_sem dim_syn dim_iface stardim
  _Expr: value
stardim:
  Stmt: control
  proxy: dim_iface
avm:
  _Value: value
  _Expr: value
value:
  _Else: avm adisj avm.Dot control.Call
syn:
  _AVM: avm
iface:
  _AVM: avm
dim_syn:
  tag: "syn"
  solver: "tree"
  Stmt: syn
  Expr: value
dim_sem:
  tag: "sem"
  Stmt: sem
  Expr: value
dim_iface:
  tag: "iface"
  Stmt: iface
  Expr: value
```

FIGURE 10.3 — Fichier `compiler.yaml` pour l'assemblage du compilateur `synsem`

termes, la compilation de la métagrammaire ne produit plus des arbres (ils sont construits par l'analyseur syntaxique), mais se contente d'accumuler des descriptions.

Les figures 10.4 10.5 présentent un ensemble de descriptions d'arbres pour une grammaire d'interaction jouet [2], contenant les entrées correspondants aux éléments du lexique *Jean*, *voit*, *la* et le signe de ponctuation ".". Les traits apparaissant en rouge (polarité négative) indiquent un besoin, qui doit être compensé par une ressource. Les ressources sont fournies par unification avec un trait apparaissant en bleu (polarité positive).

Lors de l'analyse syntaxique, les descriptions sont accumulées, puis combinées jusqu'à com-

2. extraite de la documentation http://wikilligramme.loria.fr/doku.php?id=ig:formalism

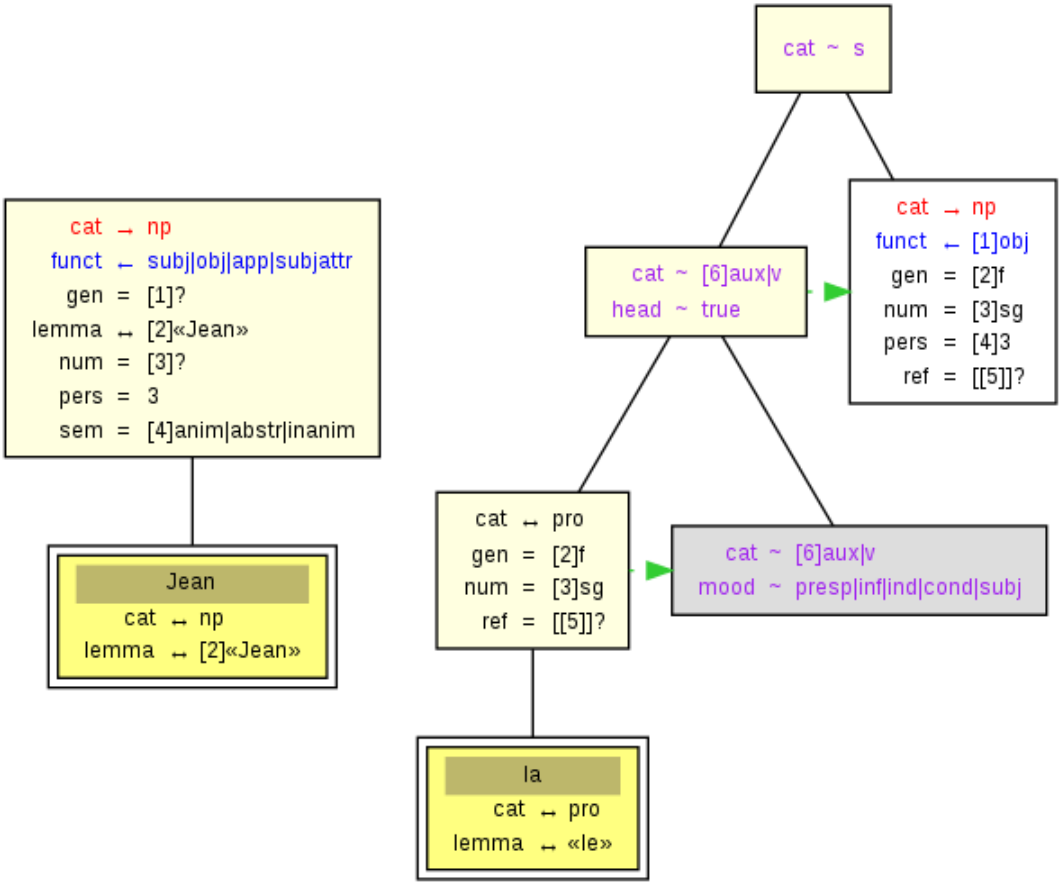


FIGURE 10.4 — Un extrait de grammaire d'interaction

pensation de l'ensemble des polarités : les traits positifs doivent avoir été unifiés avec des traits négatifs, et les traits apparaissant en violets (dits virtuels) doivent avoir été unifiés avec un trait quelconque. Une analyse de la phrase "Jean la voit." à partir des descriptions précédentes est donnée par la figure 10.6.

Le DSL proposé par XMG-1 pour les grammaires d'interaction est le même langage de description d'arbres que celui utilisé pour les arbres TAG, à ceci près qu'il autorise les traits polarisés. Il faut noter de plus que les descriptions accumulées ne sont pas résolues : les entrées de la grammaire générée sont des arbres sous-spécifiés, soit un ensemble de noeuds et de relations entre eux.

La figure 10.7 donne le code correspondant à une entrée de la grammaire d'interaction précédente : *le*. Les structures de traits polarisées sont rendues disponibles par trois opérateurs s'ajoutant au = à l'intérieur des traits : =+ et =- indiquent respectivement des traits à polarité positive à négative, et =~ est utilisé dans les traits virtuels.

10.2.2 Création de briques

À la différence de celles manipulées dans les grammaires TAG, les structures de traits utilisées dans les grammaires d'interaction sont polarisées. Pour rendre ces structures disponibles dans le DSL, nous avons créé la brique `polAvm`.

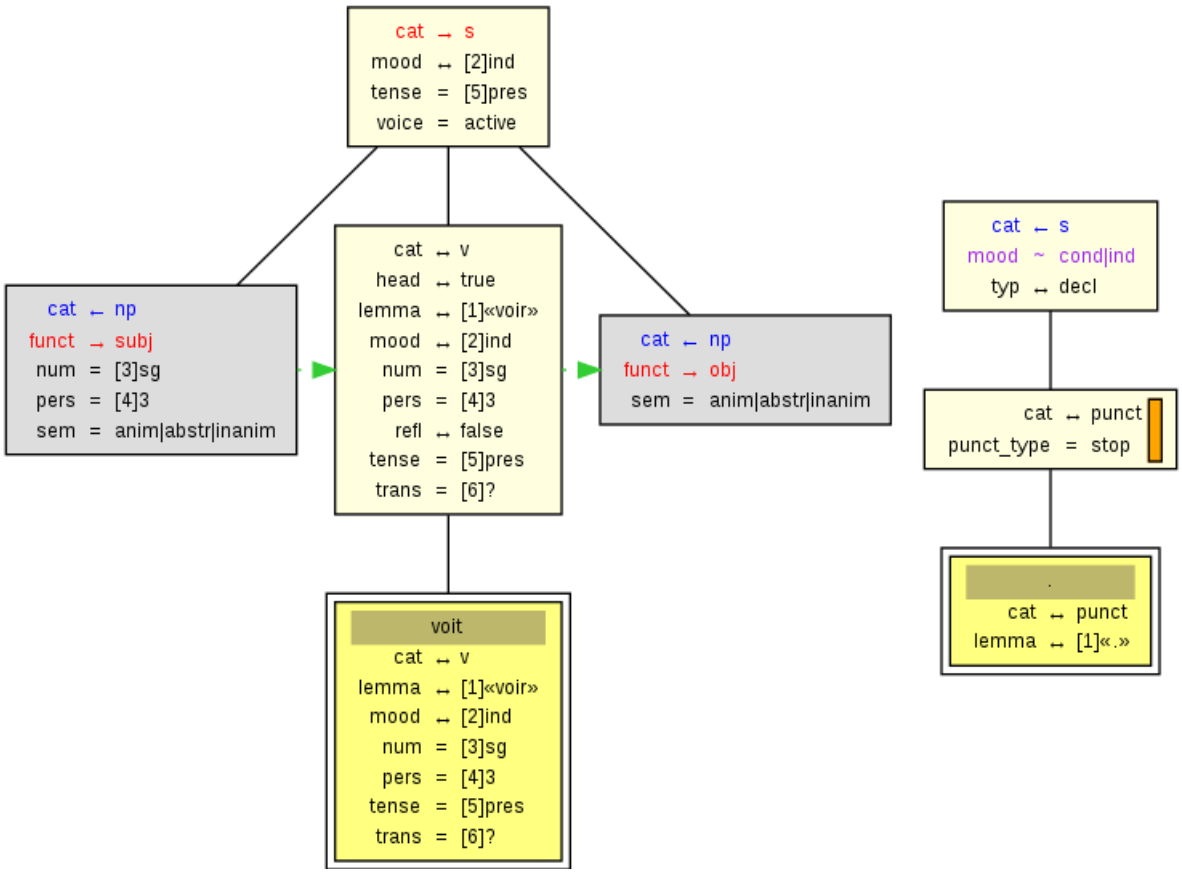


FIGURE 10.5 — Un extrait de grammaire d'interaction (suite)

Cette dernière permet de manipuler des structures de traits polarisées, c'est à dire pour lesquelles on associe à chaque trait une polarité. Le mécanisme d'unification des structures de traits est modifié en fonction de ces polarités.

Le fichier `lang.def` de cette brique est donné dans la figure 10.8.

10.2.3 Assemblage du compilateur

L'assemblage que nous utilisons pour recréer le compilateur de grammaires d'interaction de XMG-1 est présenté dans la figure 10.9.

L'assemblage est le même que pour le compilateur TAG, mis à part trois choses.

La première différence est qu'aucun solveur n'est spécifié pour la dimension `syn`. Les solutions sont donc les accumulations, converties sans être résolues. Le module de conversion utilisé est donc celui fourni par la brique `syn` et non celui de la brique `tree`.

La deuxième différence est l'absence de dimension sémantique, qui n'est pas utilisée dans les grammaires conçues jusqu'ici.

Enfin, la dernière différence est l'utilisation d'une brique `avm` alternative : `polAvm`, que nous venons de présenter.

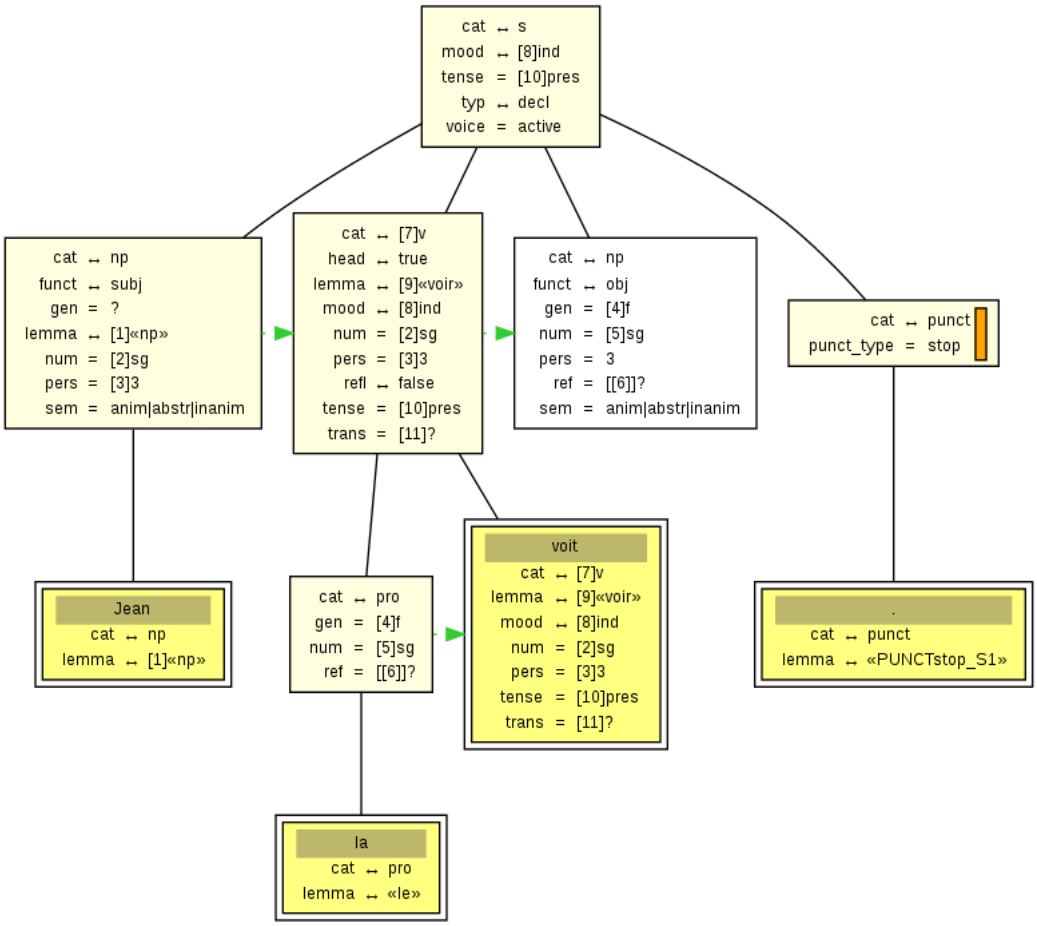


FIGURE 10.6 — Une analyse par une grammaire d'interaction

10.3 Conclusion

Dans ce chapitre, nous avons vu comment assembler modulairement le compilateur XMG-1. Nous avons proposé deux assemblages, générant respectivement des compilateurs dédiés aux grammaires d'arbres adjoints et aux grammaires d'interaction. Chacun de ces compilateurs a été utilisé pour compiler des métagrammaires existantes. Les ressources ainsi générées sont identiques à celles que produit XMG-1.

Nous avons pu donner une idée concrète de la contribution de chacun des trois profils d'utilisateurs à la tâche de production de ressource linguistique. Pour rappel, ces contributions sont pour le linguiste de définir la métagrammaire à partir d'un DSL disponible, pour le contributeur d'assembler ce DSL à partir de briques existantes, et enfin pour le programmeur de développer de nouvelles briques.

Dans le chapitre suivant, nous nous intéresserons à un autre type de ressource, les lexiques morphologiques.

```
class Pronoun
declare ?X
{
  <syn>{
    node [cat=~ s]{
      node [cat=~ @{aux,v}, head=~ true]{
        node [cat= pro, gen=f, num= sg, ref= ?X]{
          node [cat= pro, lemma= "le"]
        }
        node [cat=~ @{aux,v},
              mood=~ @{presp,inf,ind,cond,subj}]
        }
      node [cat=- np, funct=+ obj, gen= f,
            num= sg, pers= 3, ref= ?X]
      }
    }
  }
}
```

FIGURE 10.7 — Code XMG pour une description d'arbre en grammaire d'interaction

```
%%

AVM : '[' (Feat // ',')* ']'
      {get_coord($1,Coord), $$=avm:avm(Coord,$2)};

Feat : id Pol _Value {$$=polAvm:feat($1,$3,$2)};

Pol : '=' {$$=$1}
    | '=+' {$$=$1}
    | '=-' {$$=$1}
    | '=~' {$$=$1};

Dot : _Expr '.' id {$$=avm:dot($1,$3)};

%%
```

FIGURE 10.8 — Fichier `lang.def` de la brique `polAvm`

```
mg:
  _Stmt: control
  _EDecls: decls
decls:
  _ODecl: feats
control:
  _Stmt: dim_syn dim_iface stardim
  _Expr: value
stardim:
  _Stmt: control
  proxy: dim_iface
polAvm:
  _Value: value
  _Expr: value
value:
  _Else: polAvm adisj polAvm.Dot control.Call
syn:
  _AVM: polAvm
iface:
  _AVM: polAvm
dim_syn:
  tag: "syn"
  Stmt: syn
  Expr: value
dim_iface:
  tag: "iface"
  Stmt: iface
  Expr: value
```

FIGURE 10.9 — Fichier `compiler.yaml` pour l'assemblage du compilateur `ig`

Application à la morphologie 11

Sommaire

11.1 Problématique linguistique : morphologie verbale en Ikota **161**
11.1.1 Blocs élémentaires 163
11.1.2 Exemple de dérivation 164
11.1.3 Phonologie de surface 165
11.2 Création des briques **165**
11.2.1 Une brique pour décrire des champs ordonnés 165
11.2.2 Une brique pour la description morphologique 166
11.3 Assemblage du compilateur **168**
11.4 Conclusion **169**

L'approche métagrammaticale proposée par le formalisme XMG a été appliquée, comme nous l'avons vu précédemment, à différentes langues au niveau de la syntaxe (formalisée en grammaires d'arbres adjoints par exemple) et de la sémantique. Dans ce chapitre, nous nous intéressons à un autre niveau de description linguistique : le niveau morphologique. Nous proposons un nouveau langage de description, et nous l' utilisons pour décrire certains phénomènes apparaissant dans une langue à la morphologie riche, l'Ikota. Ces travaux ont été effectués en collaboration avec des linguistes du Laboratoire Ligérien de Linguistique à Orléans.

Dans un premier temps (section 11.1), nous présenterons la morphologie verbale de cette langue, puis donnerons une formalisation utilisant un DSL. Dans un deuxième temps (section 11.2), nous montrerons comment créer de nouvelles briques XMG intégrant le support pour les parties manquantes de ce DSL. Enfin, dans la section 11.3, nous réaliserons l'assemblage du DSL en utilisant ces nouvelles briques.

11.1 Problématique linguistique : morphologie verbale en Ikota

L'Ikota est une langue assez peu étudiée du Gabon et de la République Démocratique du Congo. Elle appartient à la famille des langues bantoues, avec lesquelles elle partage un certain nombre de traits (notion de classe de mots). Nous nous intéressons ici à la morphologie verbale de la langue.

L'approche métagrammaticale proposée dans [Duchier *et al.*, 2012b] a ici deux objectifs : permettre de formaliser les théories concernant la formation des verbes en Ikota, et générer des

ressources pour une langue qui en manque. Pour cette application à la morphologie, la ressource produite est un lexique de formes fléchies des verbes.

L'Ikota est une langue agglutinante, un verbe est donc un assemblage de morphèmes. Des affixes viennent se positionner de chaque coté de la racine verbale.

Les exemples suivants illustrent la conjugaison de bòʤákà "manger", un exemple typique de la classe verbale *aka* :

(1) m-à-ʤ-á ǹlɛ́sì
1sg-Présent-manger-Actif riz
"Je mange du riz" (Présent)

(2) a. m-à-ʤ-á-ná yàná
1sg-Passé-manger-Actif-Jour hier
"J'ai mangé hier" (Passé (hier))

b. m-à-ʤ-á-sá kúlá mwáyèkànàmwɛ
1sg-Passé-manger-Actif-Distant année dernière
"J'ai mangé l'année dernière" (Passé Distant)

c. m-é-ʤ-á ǹlɛ́sì
1sg-Passé-manger-Actif riz
"J'ai mangé du riz" (Passé Récent)

(3) a. m-é-ʤ-àk-à ǹlɛ́sì
1sg-Futur-manger-Inaccompli-Actif riz
"Je mangerai du riz" (Futur Moyen)

b. m-é-ʤ-àk-à-ná yàná
1sg-Futur-manger-Inaccompli-Actif-Jour demain
"Je mangerai demain" (Futur (demain))

c. m-é-ʤ-àk-à-sá kúlá mwáyàkàmwɛ
1sg-Futur-manger-Inaccompli-Actif-Distant année prochaine
"Je mangerai l'année prochaine" (Futur Distant)

d. m-ábí-ʤ-àk-à òsátè
1sg-Futur-manger-Inaccompli-Actif bientôt
"Je mangerai bientôt" (Futur Imminent)

L'ordre de ces affixes ne varie pas et peut être exprimé en terme de positions. le préfixe d'accord du Sujet occupe la position la plus à gauche, au début du mot. Le préfixe de Temps apparaît à la gauche de la racine verbale (RV). Le marqueur du progressif est immédiatement à droite de la RV. Le suffixe de l'Actif occupe le champ à la gauche du Proximal. Le suffixe du Proximal/-Distal occupe la position la plus à droite.

Pour la formalisation de ce système, nous nous inspirons du modèle Paradigm-Function Morphology [Stump, 2001b], reposant sur le concept de classes de positions. Nous nous basons sur

Sujet-	Temps-	RV	-(Aspect)	-Actif	-(Proximal)

Tableau 11.1 — Formes verbales de bòʤákà "manger"

Sujet	Temps	RV	Aspect	Actif	Prox.	Valeur
m-	à-	ʤ		-á		présent
m-	à-	ʤ		-á	-ná	passé, hier
m-	à-	ʤ		-á	-sá	passé distant
m-	é-	ʤ		-á		passé récent
m-	é-	ʤ	-àk	-à		futur moyen
m-	é-	ʤ	-àk	-à	-ná	futur, demain
m-	é-	ʤ	-àk	-à	-sá	futur distant
m-	ábí-	ʤ	-àk	-à		futur imminent

Tableau 11.2 — Formation du verbe en Ikota

la notion de domaine topologique utilisée pour la description de la syntaxe de l'allemand [Bech, 1955] pour instancier ces classes. Un domaine topologique est une séquence linéaire de champs. Chaque champ peut accueillir des contributions, et il peut y avoir des restrictions sur le nombre d'éléments qu'un champ peut ou doit recevoir. Dans notre cas, le domaine topologique d'un verbe sera tel que décrit dans le Tableau 3, et chaque champ accueillera au plus un élément, où chaque élément est la forme phonologique lexicale [1] d'un morphème.

Le langage metagrammatical adapté pour l'implémentation de cette formalisation doit nous permettre de définir une séquence ordonnée de champs, et de contribuer des éléments dans ces champs.

11.1.1 Blocs élémentaires

La métagrammaire est exprimée au moyen de blocs élémentaires. Un bloc réalise simultanément une contribution à 2 dimensions de descriptions linguistiques: (1) la phonologie lexicale: contributions aux champs du domaine topologique, (2) la flexion: contribution de traits morphosyntaxiques. Par exemple:

2 ← é
temps = passé proxi = proche

contribue é au champ numéro 2 du domaine topologique, et les traits temps = passé et proxi = proche à la flexion. Les contributions de traits provenant de différents blocs sont unifiées : de cette manière, la dimension de flexion sert également comme un niveau de médiation et de coordination durant l'exécution de la métagrammaire.

Comme le tableau 11.2 l'illustre, la morphologie de l'ikota n'est pas proprement compositionelle: en effet, les contributions sémantiques des morphèmes sont déterminées au travers d'une coordination de contraintes mutuelles dans le niveau de flexion.

1. Nous adoptons ici la perspective à *2 niveaux* qui distingue phonologie lexicale et de surface [Koskenniemi, 1983]

Sujet → [1 ← m | p = 1, n = sg] ∨ [1 ← ò | p = 2, n = sg] ∨ ...

Tense → [2 ← é | temps = passé, proxi = proche] ∨ [2 ← é | temps = futur] ∨ [2 ← à | temps = présent] ∨ [2 ← à | temps = passé, proxi = ¬proche] ∨ [2 ← ábí | temps = futur, proxi = imminent]

Actif → [5 ← À | actif = +, prog = -] ∨ [5 ← Á | actif = +, prog = +] ∨ [4 ← ébwɛ̀ | actif = -]

Aspect → [4 ← ÀK | temps = futur, prog = -] ∨ [| temps = ¬futur, prog = +]

Proximal → [6 ← nÁ | proxi = jour] ∨ [6 ← sÁ | proxi = distant] ∨ [| proxi = aucun ∨ proche] ∨ [| proxi = imminent, temps = futur]

Manger → [3 ← ɖʒ | vclasse = g1]

Donner → [3 ← w | vclasse = g2]

RV → *Manger* ∨ *Donner*

Verbe → *Sujet* ∧ *Temps* ∧ *RV* ∧ *Aspect* ∧ *Actif* ∧ *Proximal*

FIGURE 11.1 — Métagrammaire de la morphologie verbale de l'Ikota

Les traits morphosyntaxiques. Nous utilisons p et n pour *personne* et *nombre*; temps avec pour valeurs possibles `passé`, `présent`, et `futur`; proxi pour le *marqueur proximal* (`aucun`, `imminent`, `jour`, `proche`, `distant`); vclasse pour la classe verbale (`g1`, `g2`, `g3`); et deux traits polaires: actif pour la *voix* et prog pour l'*aspect progressif*: prog=- marque un évènement en déroulement.

Signes phonétiques lexicaux. Une étude attentive des données disponibles sur l'ikota suggère que l'on peut mieux rendre compte des régularités parmi les classes verbales en introduisant une voyelle *lexicale* `A` qui est réalisée, au niveau surfacique, par a pour `vclasse=g1`, ɛ pour `vclasse=g2`, et ɔ for `vclasse=g3`, et une consonne lexicale `K` qui est réalisée par tʃ pour `vclasse=g2`, et k sinon.

Règles. La Figure 11.1 montre un fragment de notre métagrammaire préliminaire de la morphologie verbale de l'ikota. Chaque règle définit comment une abstraction peut être réécrite. Par exemple *Temps* peut être réécrit par un bloc quelconque représentant une disjonction de 5 blocs. Pour produire le lexique des formes fléchies décrites par notre métagrammaire, le compilateur XMG calcule toutes les réécritures non-déterministes possibles en partant de l'abstraction *Verbe*.

11.1.2 Exemple de dérivation

Considérons comment ódʒàkàná (*(demain), tu mangeras*) est dérivé par notre système formel en partant de l'abstraction *Verbe*. Premièrement, *Verbe* est remplacé par *Sujet* ∧ *TempsRV* ∧ *Aspect* ∧ *Actif* ∧ *Proximal*. Puis chaque élément de cette conjonction logique (l'ordre est sans

Verbe → *Sujet* ∧ *Temps* ∧ *RV* ∧ *Aspect* ∧ *Actif* ∧ *Proximal*

→ [1 ← ò; p = 2, n = sg] ∧ [2 ← é; temps = futur] ∧ [3 ← ɖʒ; vclasse = g1] ∧ [4 ← ÀK; temps = futur, prog = -] ∧ [5 ← À; actif = +, prog = -] ∧ [6 ← nÁ; proxi = jour]

→ [1 ← ò 2 ← é 3 ← ɖʒ 4 ← ÀK 5 ← À 6 ← nÁ; p = 2 prog = - temps = futur vclasse = g1; n = sg actif = + proxi = jour]

FIGURE 11.2 — Une dérivation avec succès

Verbe → *Sujet* ∧ *Temps* ∧ *RV* ∧ *Aspect* ∧ *Actif* ∧ *Proximal*

→ [1 ← ò; p = 2, n = sg] ∧ [2 ← é; temps = futur] ∧ [3 ← ɖʒ; vclasse = g1] ∧ [; temps = ¬futur, prog = +] ∧ [5 ← À; actif = +, prog = -] ∧ [6 ← nÁ; proxi = jour]

→ **failure!**

FIGURE 11.3 — Une dérivation avec échec: conflits sur temps et prog

importance) est, à son tour, remplacé. Par exemple, *Sujet* est alors remplacé par un bloc de la disjonction correspondante: le compilateur XMG essaie toutes les possibilités; l'une d'entre elles choisira le 2ème bloc. La Figure 11.2 montre l'étape initiale, une étape au milieu, et le l'étape finale de la dérivation. La phonologie lexicale de l'entrée lexicale résultante est obtenue en concatenant, dans l'ordre linéaire du domaine topologique, les items contribués aux différents champs ici: ò+é+ɖʒ+ÀK+À+nÁ.

La Figure 11.3 montre un exemple d'une dérivation rencontrant un échec, donc, qui ne mène pas à la production d'une entrée du lexique. L'échec est dû à des valeurs contradictoires pour les traits temps (`futur` et ¬`futur`) et aussi prog (+ et -). Les échecs nous permettent d'éviter la surgénération de la métagrammaire.

11.1.3 Phonologie de surface

Pour l'instant, notre métagrammaire modélise uniquement le niveau lexical de la phonologie. Le niveau surfacique peut en être dérivé par post-traitement. Pour notre exemple, puisque vclasse=`g1`, le `A` lexical devient a en surface, et le `K` devient k. Ainsi nous obtenons:

ò+é+ɖʒ+àk+à+ná

et finalement (par effacement de voyelle) óɖʒàkàná.

11.2 Création des briques

11.2.1 Une brique pour décrire des champs ordonnés

Pour ce DSL, nous avons créé la brique `fields`, permettant la déclaration des champs topologiques et la définition des contraintes d'ordre entre ces champs (grâce à l'opérateur de précé-

dence linéaire >>). Un solveur énumère les modèle du problème sous contraintes formulé. Le fichier de définition de langage de la brique `fields` est donné dans la figure 11.4.

```
%%

FieldDecl : Field {$$=$1}
          | FieldPrec {$$=$1};


Field : 'field' id {$$=fields:field($2)};

FieldPrec : id '>>' id {$$=fields:fieldprec($1,$3)};


%%
```

Figure 11.4 — Le fichier `lang.def` de la brique `fields`

```
field f_sujet
field f_temps
field f_racine
field f_voix
field f_aspect
field f_theme
field f_eloignement

f_sujet >> f_temps
f_temps >> f_racine
f_racine >> f_voix
f_voix >> f_aspect
f_aspect >> f_theme
f_theme >> f_eloignement
```

Figure 11.5 — Exemple de définition d'une suite de champs ordonnés

Un exemple de définition d'une suite de champs topologiques est donné dans la figure 11.5. L'unique modèle de cette description est la suite de champs suivante :
`f_sujet-f_temps-f_racine-f_voix-f_aspect-f_theme-f_eloignement`

11.2.2 Une brique pour la description morphologique

La brique `morphtf` définit le DSL de la dimension morphologique, que l'on souhaite délimiter dans cet assemblage par la balise `<morph>`. Nous voulons réaliser via cette dimension la contribution de matériel dans un champ topologique, ou celle d'une valeur pour un trait morphosyntaxique.

Un exemple de contribution à la dimension morphologique issu de la métagrammaire verbale de l'Ikota est donné en figure 11.6. Cette classe, correspondant à l'abstraction *Sujet* de la figure

```
class Subject_Clitic
{
  <morph>{
    { p:1; n:sg ; f_sujet <- "m"}
    |
    { p:2; n:sg ; f_sujet <- "o" }
    |
    { p:3; n:sg ; f_sujet <- "a" }
    |
    { p:1; n:pl ; f_sujet <- "min"}
    |
    { p:2; n:pl ; f_sujet <- "bih" }
    |
    { p:3; n:pl ; f_sujet <- "b" }
  }
}
```

FIGURE 11.6 — Exemple de contribution à la dimension `<morph>`

11.1, place dans le champs `f_sujet` le morphème adapté en fonction de la personne et du nombre associés aux traits `p` et `n`. À la première personne du pluriel, on place par exemple le préfixe "min" dans ce champs.

```
%%

MorphStmt : InField {$$=$1}
          | Eq {$$=$1};

InField : id '<-' Expr {$$=morphtf:infield($1,$3)};

Eq : Expr ':' Expr {$$=morphtf:eq($1,$3)};

Expr : id {$$=$1}
     | string {$$=$1}
     | int {$$=$1}
     | bool {$$=$1};

%%
```

FIGURE 11.7 — Fichier `lang.def` de la brique `morph`

Morphologie à deux niveaux Comme nous l'avons vu en sous-section 11.1.3, les formes que l'on contribue aux champs topologiques sont des formes lexicales, puisque nous utilisons l'approche à deux niveaux de [Koskenniemi, 1983]. Nous obtenons donc après exécution du code généré des formes fléchies pour lesquelles il reste à déterminer la forme de surface. N'ayant

pas de solution immédiate pour traiter cette dérivation, nous avons utilisé un script en post-processing sur le lexique généré pour obtenir le lexique des formes de surface.

11.3 Assemblage du compilateur

```
mg:
  _Stmt: control
  _EDecls: decls
decls:
  _ODecl: feats fields
control:
  _Stmt: dim_tf dim_iface stardim
  _Expr: value
stardim:
  _Stmt: control
  proxy: dim_iface
avm:
  _Value: value
  _Expr: value
value:
  _Else: avm adisj avm.Dot control.Call
iface:
  _AVM: avm
dim_tf:
  tag: "morph"
  Stmt: morphtf
  Expr: value
dim_iface:
  tag: "iface"
  Stmt: iface
  Expr: value
```

FIGURE 11.8 — Fichier `compiler.yaml` pour l'assemblage du compilateur `morphtf`

L'assemblage du compilateur utilisé dans le cadre de ces travaux est présenté dans la figure 11.8. Les deux briques créées pour son développement, `fields` et `morphtf`, sont respectivement définies comme une brique de déclaration et comme une dimension.

Concrètement, la brique `fields` est branchée sur la brique `decls`, comme les autres briques permettant des déclarations globales (au même niveau que celle des types). La brique `morphtf` est quand à elle placée comme paramètre du constructeur de brique `dim`, l'étiquette fournie pour la dimension étant `morph`.

11.4 Conclusion

Nous avons, dans ce chapitre, donné la méthodologie pour le développement d'un DSL pour la description de la morphologie en utilisant des champs topologiques. Deux briques ont été crées pour rendre l'assemblage du compilateur possible : une brique permettant de définir globalement des champs topologiques, et de les ordonner, et une autre rendant possibles les contributions à ces champs. Enfin, nous avons réalisé l'assemblage du compilateur en utilisant ces nouvelles briques.

Ce DSL a permis de développer un lexique de taille moyenne (plusieurs centaines de formes) pour la morphologie verbale de l'Ikota (voir [Duchier *et al.*, 2012b], [Duchier *et al.*, 2012a], [Magnana Ekoukou, 2014]).

Application à la sémantique 12

Sommaire

12.1 Problématique linguistique **171**
12.1.1 Factorisation de squelettes d'arbres et de frames 172
12.1.2 une nouvelle dimension <frame> 173
12.2 Création de briques **173**
12.2.1 Une brique pour définir des hiérarchies de types 173
12.2.2 Une brique pour décrire des frames 174
12.3 Assemblage du compilateur **175**
12.4 Conclusion **176**

Dans ce chapitre, nous présentons l'application de l'approche métagrammaticale à un formalisme sémantique basé sur les structures de traits typées. Nous expliquons ensuite comment intégrer au compilateur les nouveaux éléments nécessaires à la description de ces nouvelles structures. Cette intégration passe par la création de deux nouvelles briques (travaux initiés dans [Lichte *et al.*, 2013]). Nous terminons en donnant l'assemblage du DSL adapté à cette nouvelle tâche de génération.

12.1 Problématique linguistique

Nous avons jusqu'ici proposé des cadres de description pour la syntaxe et la morphologie. Nous souhaiterions maintenant enrichir ces cadres pour permettre de également la sémantique (ici, le sens d'un énoncé). Nous allons nous baser sur un cadre hérité de Montague, dans lequel le calcul sémantique est guidé par la structure syntaxique. Pour cette dernière, nous reprendrons le formalisme TAG utilisé précédemment. Pour la représentation du sens, nous utiliserons des structures de traits typées appelées frames (voir [Petersen, 2007]). Les types utilisés dans les frames sont organisés dans une hiérarchie, qui définit les règles d'unification entre types.

Nous souhaitons donc décrire des objets lexicaux tels que celui présenté dans la figure 12.1, c'est à dire à un patron d'arbre associé à un syntagme, une structure de traits typée, et la hiérarchie de type associée, exemple tiré de [Kallmeyer et Osswald, 2012b], [Kallmeyer et Osswald, 2013].

Les squelettes de syntagmes tels que (a) composent le lexique d'une grammaire TAG, on les appelle aussi *arbres élémentaires*. Comme dans les exemples vus dans la section 3.3 (page 35), l'arbre élémentaire de (a) contient une ancre lexicale, sur laquelle peut venir se placer l'élément du lexique *throws*.

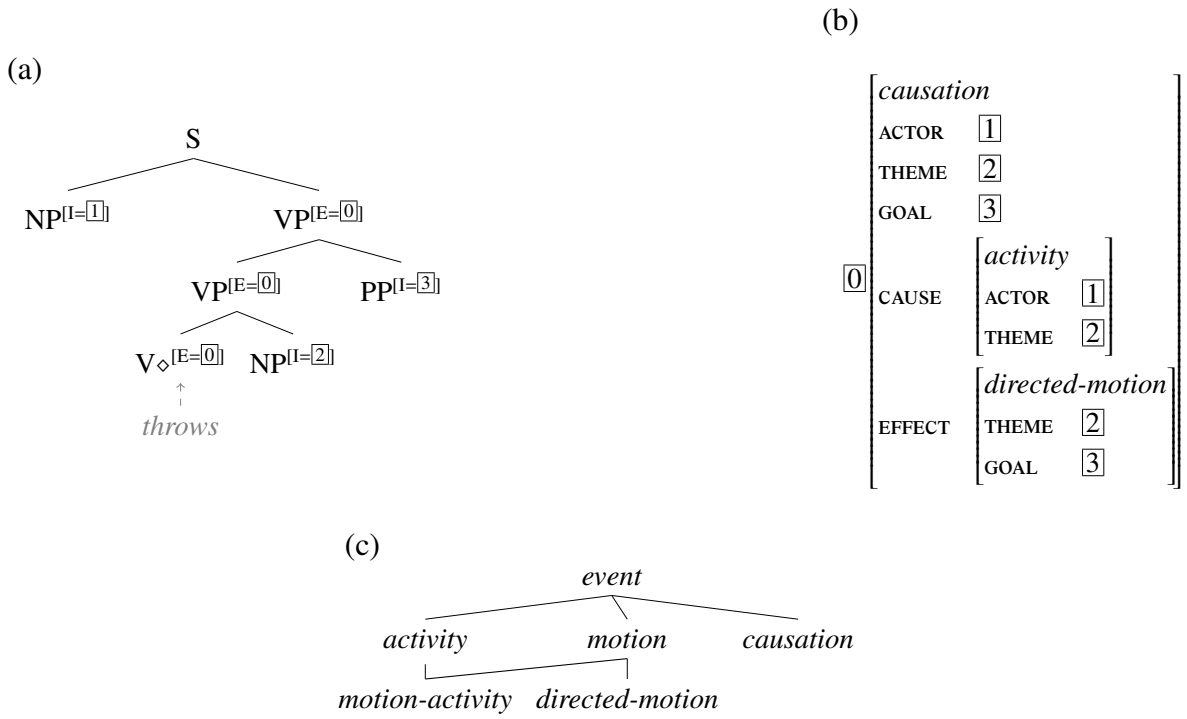


FIGURE 12.1 — Un patron d'arbre associé à un syntagme, sa contrepartie sémantique et la hiérarchie de type associée

Les structures de traits typées telles que (b) sont une représentation standard des frames en question (voir [Petersen, 2007]), qui, d'après la Frame Theory [Fillmore, 1982], sont considérées comme des représentations de concepts mentaux. Comme on peut le voir dans l'exemple de la figure 12.1, des traits décrivent les participants et les composants sémantiques, tandis que les structures de traits correspondent à des objets conceptuels, réduits au type (*causation*, *activity*, ...) auquel elles sont associées. De plus, les types sont organisés dans une hiérarchie, qui détermine l'ensemble des traits sémantiques appropriés et la possibilité d'unifier des structures de traits. Pour finir, les nombres encadrés indiquent des réentrances, c'est à dire des identités de structure [1]. C'est également le cas pour la structure de traits de la figure 12.1.

Les structures de traits typées ont été largement étudiées dans le cadre du développement de grammaires et du parsing, notamment en HPSG ([Carpenter, 1992], [Carpenter et Penn, 1996], [Götz *et al.*, 1997], [Malouf *et al.*, 2000], [Flickinger, 2000], [Copestake, 2002]). Leur intégration aux DSL proposés par XMG présente donc également un avantage dans la mesure où elles interviennent dans différents formalismes.

12.1.1 Factorisation de squelettes d'arbres et de frames

Les objets lexicaux richement structurés tels que ceux de la figure 12.1 rendent nécessaire une certaine factorisation métagrammaticale, dans le cas de la création et de la maintenance d'une grammaire à large couverture [Xia *et al.*, 2010].

En plus des bénéfices en terme de conception de grammaire, comme l'ont mentionné [Kallmeyer et Osswald, 2012a], [Kallmeyer et Osswald, 2012b], [Kallmeyer et Osswald, 2013], la

1. elles permettent de faire correspondre des structures de traits à des graphes plutot qu'à des arbres

factorisation métagrammaticale peut également refléter des analyses constructionnelles, dans l'esprit de la Construction Grammar [Kay, 2002], [Goldberg, 2006]. Sous cette perspective le matériel lexical et les "constructions" utilisées participent à la représentation du sens.

Si l'on prend ces deux aspect en considération, [Kallmeyer et Osswald, 2013] propose la factorisation des squelettes d'arbres et de la frame de la figure 12.1 en 12.2, où les boites représentent des facteurs résultants ou des classes (dans le sens de classes XMG), constituées d'un arbre et d'un fragment de frame. [2] Ceci illustre le fait que le couple arbre-frame dans la figure 12.1 résulte de l'instantiation de la classe POConstr, qui combine les classes n0Vn1 et DirPrepObj-to. Notons que la figure 12.2 montre une partie de la factorisation proposée seulement, puisque la classe n0Vn1 par exemple résulterait de la combinaison de trois autres classes (Subj, VSpine, DirObj). Combiner deux classes signifie essentiellement que les informations associées sont unifiées, et qu'un modèle minimal est calculé. Finalement, un aspect de la construction peut etre trouvé dans la classe POConstr : elle contribue un fragment de frame, mais pas de fragment d'arbre.

La représentation graphique de la factorisation métagrammaticale dans la figure 12.2 reste à un niveau assez informel, et la question de la traduction en code XMG se pose.

12.1.2 une nouvelle dimension <frame>

Le langage de description employé dans la dimension `<frame>` du compilateur utilisé pour ce travail a pour but d'être le plus proche possible des structures vues dans la figure 12.2. La dimension `<frame>` contient des descriptions de structures de traits typées, associées à d'éventuelles étiquettes dans l'optique de leur faire référence. L'exemple de la figure 12.3 illustre ceci.

Les hiérarchies utilisées sont quand à elles définies d'après un ensemble de contraintes données dans les déclarations. Les types que l'on considère sont des types conjonctifs (voir [Carpenter et Pollard, 1991]), ce qui signifie que pour un ensemble de type élémentaires e, l'ensemble des types accessibles (si l'on ne considère aucune contrainte sur ceux ci), est l'ensemble des parties de e.

La hiérarchie calculée est dans l'implantation actuelle un modèle maximal, ce qui signifie que seuls les types conjonctifs interdits par les contraintes sont retirés de l'ensemble des types accessibles.

Un exemple de déclarations de contraintes de types est donné dans la figure 12.4.

Une différence essentielle entre les dimensions `<syn>` et `<frame>` se trouve au niveau du solveur. Les descriptions de la dimension `<frame>` sont totalement spécifiées (même si une description peut être composée de plusieurs structures déconnectées).

12.2 Création de briques

12.2.1 Une brique pour définir des hiérarchies de types

La brique `hierarchy`, que nous avons créée pour ce travail, permet la déclaration dans les entêtes de types de structures de traits et de contraintes sur la hiérarchie de types. Un exemple de description utilisant cette brique a été donné dans la figure 12.4.

2. Les arètes doubles indiquent des contraintes d'identité, les arêtes discontinues et le symbole <* à l'intérieur des arbres indiquent respectivement des relations de dominance et de précédence non strictes

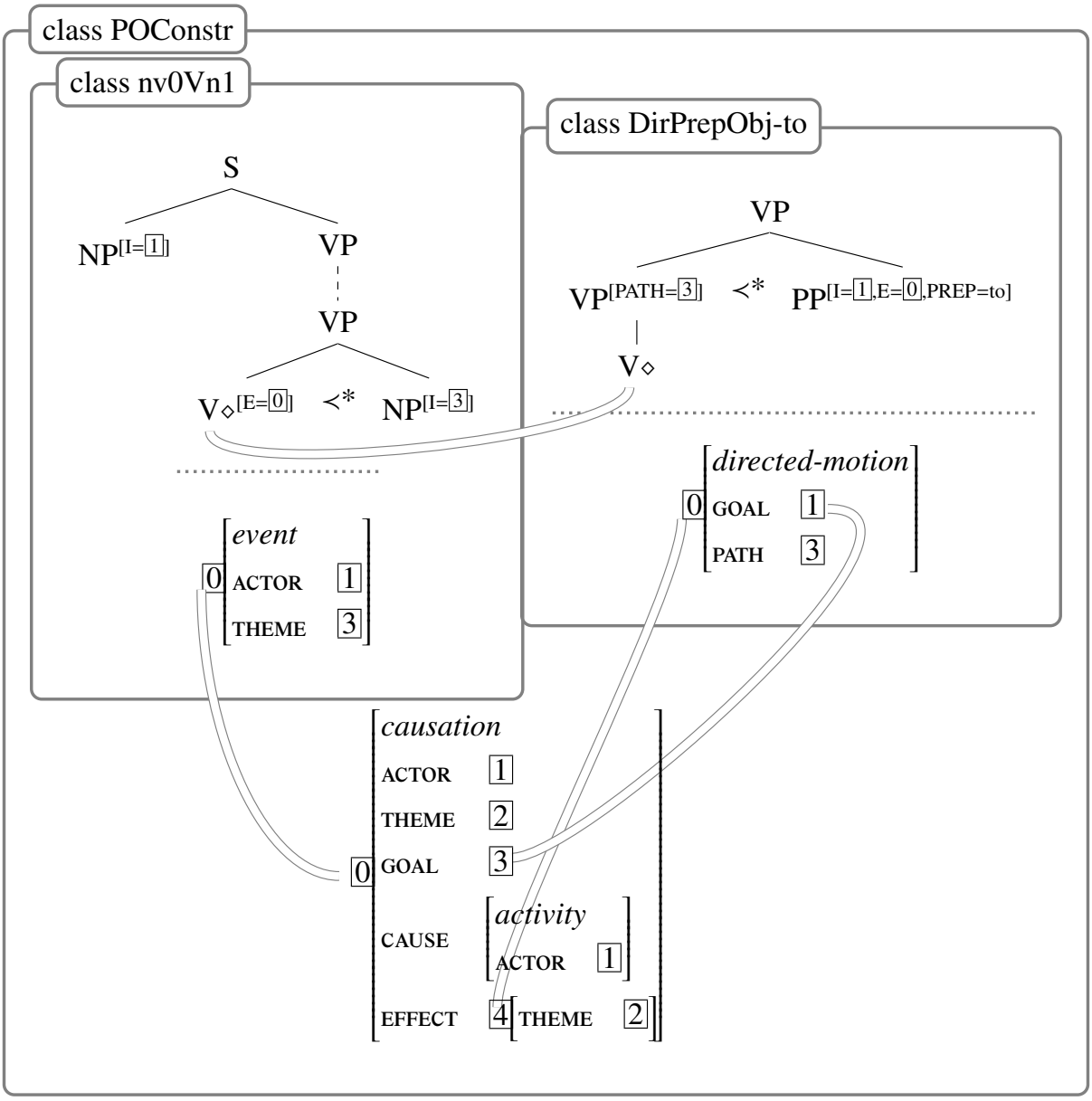


FIGURE 12.2 — Factorisation métagrammaticale du patron d'arbre et de la frame de la figure 12.1.

Pour représenter les types conjonctifs, nous avons choisi (comme [Carpenter et Penn, 1996] par exemple) d'utiliser des vecteurs de bits. Ces derniers sont des listes de taille t, où t est le nombre de types élémentaires (ceux qui ont été déclarés). À chaque position, le chiffre 1 indique la présence du type élémentaire associé à cet indice dans le type conjonctif représenté par le vecteur. Le vecteur composé uniquement de 0 est donc le type le plus générique (le type vide), et celui composé uniquement de 1 est le plus spécifique (composé de tous les types élémentaires).

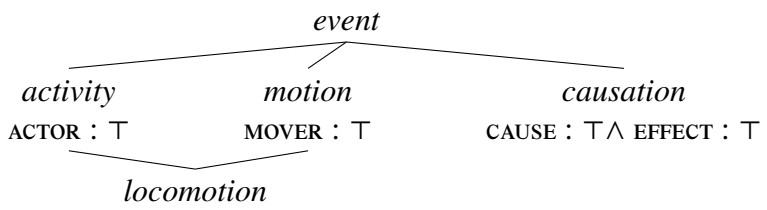
12.2.2 Une brique pour décrire des frames

La brique `frame` contient le support pour la définition des objets sémantiques que nous souhaitons pouvoir déclarer dans la nouvelle dimension `<frame>`. La grammaire hors contexte

$$\boxed{0}\begin{bmatrix} \textit{causation} & \\ \text{ACTOR} & \boxed{1} \\ \text{THEME} & \boxed{2} \\ \text{ACTOR} & \boxed{3} \\ \text{CAUSE} & \begin{bmatrix} \textit{activity} & \\ \text{ACTOR} & \boxed{1} \end{bmatrix} \\ \text{EFFECT} & \boxed{4}\begin{bmatrix} \text{THEME} & \boxed{2} \end{bmatrix} \end{bmatrix}$$

```
<frame>{
?X0[causation,
    actor:?X1,
    theme:?X2,
    goal:?X3,
    cause:[activity,
           actor:?X1],
    effect:?X4[theme:?X2]
]}
```

FIGURE 12.3 — Spécification de la frame de POConstr




```
frame_constraints = {
    activity -> event, activity -> [actor:+],
    motion -> event, motion -> [mover:+],
    causation -> event, causation -> [cause:+,effect:+],
    locomotion -> activity motion}
```

FIGURE 12.4 — Exemple de spécification de contraintes sur les types de structures de traits

contenue dans son fichier `lang.def` est donnée dans la figure 12.6.

Les structures de traits manipulées grâce à cette brique diffère de celles vues jusqu'ici. On utilise toujours une variable attribuée pour représenter une structure de traits, mais elle est associée en plus de ses traits à un type. Ce type est également représenté par une variable attribuée, son attribut étant le type (encodé sous la forme d'un vecteur de type). Le mécanisme d'unification spécifique à ces attributs est similaire à l'opération logique *et* bit à bit.

Le DSL défini dans cette brique permet des descriptions du type de celle donnée en 12.3.

12.3 Assemblage du compilateur

L'assemblage du compilateur utilisé dans le cadre de ces travaux est présenté dans la figure 12.7. Les deux briques ont créées pour son développement, `hierarchy` et `frame`, sont intégrées de la même façon que les nouvelles briques du chapitre 11. La brique `hierarchy` est connectée sur `decls` pour que les déclarations de hiérarchies de type soient réalisées globalement (avec celles des types). La brique `frame` est donnée comme paramètre du constructeur `dim` : le DSL défini par la brique est donc disponible dans les classes, délimité par la balise `<frame>`. L'utilisation de ce constructeur signifie également que l'on peut combiner plusieurs instructions de ce DSL avec le langage de contrôle (dont une nouvelle instance est créée par le constructeur `dim`).

```
%%

Hierarchy : FTypes {$$=$1} | FConsts {$$=$1};

FTypes : 'frametypes' '=' '{' (id // ',')+ '}'
         {$$=hierarchy:ftypes($4)};

FConsts : 'frameconstraints' '=' '{' (FConst // ',')+ '}'
          {$$=hierarchy:fconstraints($4)};

FConst : IDsOrAttr FOp IDsOrAttr
         {$$=hierarchy:fconstraint($2,$1,$3)};

FOp : '->' {$$=$1} | '<->' {$$=$1} ;

IDsOrAttr : (id)+ {$$=ids($1)} | bool {$$=$1}
          | id '=' id {$$=pathEq($1,$3)}
          | id ':' id {$$=attrType($1,$3)}
          | id ':' bool {$$=attrType($1,$3)};

%%
```

FIGURE 12.5 — Fichier `lang.def` pour les hiérarchies de types

12.4 Conclusion

Nous avons conçu un nouveau DSL permettant la description d'objets sémantiques (frames), ainsi que celle d'une hiérarchie de types à partir de contraintes. Pour l'implémentation de ce DSL, nous avons créé deux briques réutilisables, l'une pour définir les hiérarchies de types et l'autre pour décrire les objets sémantiques. Enfin, nous avons utilisé ces briques pour la description du compilateur pour le DSL.

D'un point de vue plus général, nous avons permis la description de structures qui pourraient être utilisées dans des cadres différents. Les structures de traits typées pourraient en effet intervenir dans la conception d'une métagrammaire HPSG, par exemple.

L'assemblage de ce compilateur dédié est en cours de raffinement, via son utilisation par des linguistes allemands.

```
%%

Frame :  (Var)? '[' id (',')? (Pair // ComaOrNot )* ']'
         {$$=frame:frame($1,$3,$5)};

ComaOrNot : ( ',' )? {$$=$1};

Pair : id ':' VarOrFrame {$$=frame:pair($1,$3)};

VarOrFrame : Var {$$=$1} | Frame {$$=$1};

Var : id {$$=$1}
    | '?' id {$$=$2};

%%
```

FIGURE 12.6 — Fichier `lang.def` pour la dimension `<frame>`

```
mg:
  _Stmt: control
  _EDecls: decls
decls:
  _ODecl: feats hierarchy
control:
  _Stmt: dim_frame dim_syn dim_iface
  _Expr: value
avm:
  _Value: value
  _Expr: value
value:
  _Else: avm adisj avm.Dot control.Call
syn:
  _AVM: avm
iface:
  _AVM: avm
dim_frame:
  tag: "frame"
  Stmt: frame
  Expr : value
dim_syn:
  tag: "syn"
  solver: "tree"
  Stmt: syn
  Expr: value
dim_iface:
  tag: "iface"
  Stmt: iface
  Expr: value
```

Figure 12.7 — Fichier `compiler.yaml` pour l'assemblage du compilateur `synframe`

Conclusion générale

Bilan

Au centre de cette thèse se trouve la notion de langages (informatiques et humains). Si le cadre initial était celui du traitement automatique des langues, et donc le langage naturel, nos travaux se sont orientés vers la conception de langages dédiés, ou DSL, dans le but de décrire et générer des collections de structures de données complexes. Le domaine d'application auquel nous nous intéressons reste néanmoins celui des données linguistiques.

Les langages de description pour ce domaine d'application sont nombreux, comme nous l'avons vu dans la première partie. Cette hétérogénéité est due à l'existence de différents niveaux de description linguistique (eg. syntaxe et sémantique), et à la multiplicité des approches pour décrire la langue à chacun de ces niveaux (par exemple syntaxe générative ou basée sur la théorie des modèles). La description des structures liées à une nouvelle approche nécessite la disponibilité ou la création d'un ou de plusieurs DSL. Ces langages, au même titre que les langages de programmation, ont une durée de vie limitée si leur communauté d'utilisateurs ne peut pas participer à leur évolution, et les adapter à des besoins spécifiques. Développer une nouvelle solution (un nouveau DSL) pour chacune de ces théories, ou chacun de ces niveaux est une tâche complexe. De plus, les frontières entre les niveaux de représentation linguistique étant floues, il est courant de voir un outil intégrer plusieurs de ces niveaux.

Dans ce contexte, nous avons proposé une méthodologie pour le développement flexible de DSL pour la génération de ressources langagières, l'idée étant de composer un DSL en réutilisant des parties de DSL existants.

La famille de DSL que nous avons choisie de composer s'inspire du langage XMG, puisque nous avons noté que cette approche métagrammaticale intégrait les concepts essentiels présents dans les méthodes de description de la langue, à savoir une vision modulaire sous forme de dimensions équipées de langages distincts qui peuvent cependant partager des espaces de noms. Ces DSL ont en commun d'être basés sur la notion de variables d'unification et de descriptions sous forme de contraintes.

Notre méthodologie s'appuie sur une approche modulaire. Nous avons montré que l'assemblage d'un compilateur pouvait être réalisé à partir de fragments de compilateur. Au delà de ceci, nous avons montré que l'assemblage des fragments de compilateur (les briques) pouvait être réalisé en se basant sur un unique fichier de description. Ce fichier indique comment assembler la grammaire hors contexte du DSL associé à ce compilateur, en spécifiant les interfaces entre les briques.

Pour pouvoir être capable d'assembler un DSL, il convient de rappeler le principe de la compilation d'une métagrammaire. Celui-ci se divise en deux grandes étapes : la compilation du code métagrammatical et la génération de la ressource décrite. Chacune de ces étapes est également

divisée en un ensemble de phases. Nous avons présenté, pour chacune de ces phases, les propriétés des langages entrants et sortants et les règles qu'une brique devait apporter pour réaliser les traductions nécessaires.

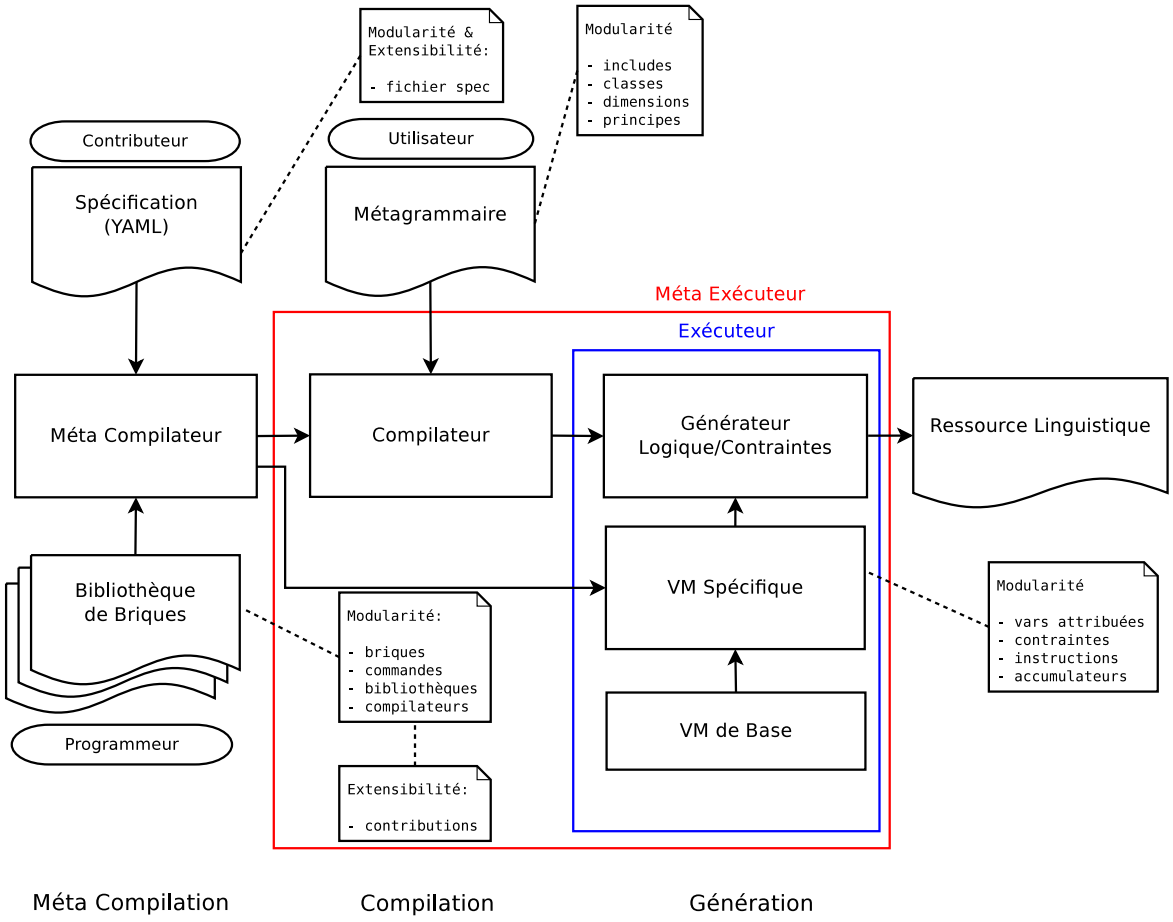


FIGURE 1 — Les points de modularité de XMG

La figure 1 rappelle l'architecture de notre approche à base d'assemblage de DSL. Elle fait apparaître deux outils différents. Le premier est un méta compilateur, réalisant l'assemblage d'un compilateur d'après une spécification. Le second est le compilateur (`méta exécuteur` sur le schéma) assemblé, utilisé par le linguiste pour produire une ressource linguistique à partir d'une métagrammaire. De plus, le schéma a été étendu pour résumer les différents points où la modularité se manifeste lors des différentes phases de méta compilation et de méta exécution.

Trois types d'utilisateurs distincts peuvent intervenir dans cette chaîne de compilation :

— *L'utilisateur* dispose des outils offerts par la famille de DSL visés par l'approche : les abstraction (par le biais des classes et des inclusions de fichiers), la séparation des descriptions en dimensions, et la possibilité de configurer les méthodologies de développement au moyen des principes

— *Le contributeur*, ou assembleur, dispose de la modularité offerte par l'assemblage déclaratif unique (via le fichier `compiler.yaml`) du DSL. Les instances multiples de briques, éventuellement configurées de manières différentes, ainsi que les constructeurs de briques, permettent de façonner finement les éléments combinés

— *Le programmeur* enrichit une bibliothèque de briques. Ces dernières permettent d'isoler des traitements spécifiques, des solveurs, de manière à les rendre réutilisables. Des outils d'extensibilité (présentés dans l'annexe A) sont également mis à la disposition du programmeur (par exemple, un générateur de squelette de brique)

Pour récapituler, le profil programmeur, correspondant à un informaticien familier avec l'architecture de XMG-2, peut créer de nouvelles briques, correspondant à de nouvelles fonctionnalités. Le profil contributeur peut quand à lui réaliser un assemblage de compilateur dédié à une tâche, en connectant les briques existantes à la manière d'un LEGO™. Enfin, le profil utilisateur, ou linguiste, produit la description métagrammaticale de la ressource, utilisant le DSL assemblé.

Nous avons montré (dans le chapitre 9) qu'offrir un point d'entrée dans la chaîne de développement à un profil contributeur (ou assembleur), constituant un intermédiaire en l'utilisateur linguiste et le programmeur expert, était l'une des innovations majeures de l'approche proposée. Nous permettons de cette manière la création, ou l'adaptation, de cadres de description adaptés à une tâche, ceci sans demander de connaissances techniques sur l'outil [3].

De plus, dans la dernière partie, nous avons montré comment notre approche pouvait être utilisée concrètement pour élaborer des DSL pour la génération de ressources langagières. Deux nouvelles dimensions, permettant respectivement la description de la morphologie en utilisant les champs topologiques et celle de la sémantique en utilisant les frames, ont ainsi été conçues et utilisées pour assembler de nouveaux compilateurs.

C'est ainsi qu'un ensemble de collaborations avec des linguistes ont été initiées, durant lesquelles il s'agissait dans un premier temps de définir un cadre métagrammatical adapté à la tâche de description, puis de contribuer à la définition de la description métagrammaticale. Ces collaborations nous ont permis d'expérimenter notre approche sur des cas linguistiques concrets et suffisamment variés (différents formalismes, différents niveaux de description linguistique), parmi lesquels figurent la description de la morphologie de l'ikota [Duchier *et al.*, 2012b] et du somali, celle de la syntaxe de langues créoles (le gwada et le são-tomense [Schang *et al.*, 2012]) et du serbe.

Perspectives

L'utilisation d'une approche modulaire s'est révélée efficace dans une partie de la procédure de génération, principalement : la phase de compilation du langage métagrammatical en code exécutable. En effet, notre approche permet d'assembler modulairement un compilateur pour un langage métagrammatical (DSL) de notre choix. Ce DSL peut ainsi être utilisé par un linguiste pour produire à partir d'un langage de haut niveau une ressource langagière (dite de bas niveau). Cette représentation à bas niveau peut être sujette à des traitements additionnels, comme les descriptions syntaxiques d'une grammaire TAG qui doivent être résolues pour produire les règles. La suite du processus fait donc intervenir des mécanismes pour lesquels la modularité telle que nous l'avons envisagée ne réalise qu'une partie du travail. La notion de plugins pour les solveurs, implémentant les principes (vus en sous-section 3.3.4), facilite néanmoins l'extensibilité pour le programmeur. Les nouvelles dimensions proposées ne nécessitant pas la conception de solveur aussi complexe que celui utilisé pour les grammaires d'arbres, les travaux concernant les librairies de résolution de contraintes n'ont pas eu l'occasion de s'approfondir. Pour

3. Une documentation est en cours de création pour permettre à des non informaticiens de jouer le rôle d'assembleur, ce qui n'est pas encore le cas.

cette raison, il serait bénéfique de s'intéresser dans le futur à des DSL manipulant de nouvelles structures sous-spécifiées (par exemple des graphes).

Un autre axe de recherche serait de valider l'approche proposée sur de nouveaux formalismes, et niveaux de description linguistique. Par exemple, dans nos travaux sur l'ikota, nous nous sommes contentés d'utiliser un post-processing pour dériver les formes de surface depuis les formes lexicales accumulées. L'approche de XMG basée sur les contraintes en fait une plateforme idéale pour l'intégration de la *morphologie à deux niveaux* puisque celle-ci est précisément une contrainte entre la phonologie lexicale et surfacique [Koskenniemi, 1983]. Une approche métagrammaticale peut également s'adapter à la description de structures n'appartenant pas au domaine de la linguistique. Dans l'annexe B, nous proposons une utilisation des métagrammaires pour la description de structures musicales.

L'approche modulaire proposée s'inspire des approches existantes pour le développement logiciel. Nous pensons que le futur du développement de ressources linguistiques gagnerait à s'orienter dans la même direction que celle suivie par le développement de programmes. Pour ce faire, le cadre de développement que nous proposons doit disposer d'outil similaires, par exemple un debugueur assemblé automatiquement, un environnement de développement intégré. Dans cet esprit, nous avons débuté la conception d'un ensemble d'outils facilitant l'extensibilité de XMG, que nous présentons dans l'annexe A. Cette métaphore avec le développement logiciel mérite d'être étudiée plus en détails.

Annexes

Annexe A. Outils d'extension

Sommaire

Introduction **185**
Commandes **185**
Compilateurs **185**
Création d'une brique **186**
Contributions **186**

Introduction

Dans ce chapitre, nous décrivons les outils fournis par XMG pour atteindre l'extensibilité. Ces outils tirent avantage de l'implémentation modulaire proposée dans les chapitres 7 et 8.

Commandes

De par la nature hétérogène des ressources générées par les compilateurs XMG, l'exploitation des résultats nécessite des outils très différents. Par exemple, si la sortie est un ensemble de descriptions d'arbres, un besoin classique est de vouloir explorer la grammaire en observant les arbres graphiquement à travers une visionneuse. Pour d'autres types de sortie, comme des lexiques de formes fléchies pour l'étude de la morphologie, l'utilisation de scripts peut favoriser leur visualisation.

Pour ces raisons, le compilateur doit pouvoir intégrer de nouvelles commandes, rendant disponible ces outils. Ces commandes sont intégrées indépendamment des briques de compilateurs avec lesquelles elles sont utilisées, pour la raison simple qu'elles peuvent se révéler utile avec différents types de compilateurs.

Parmi les commandes, on trouve aussi celles concernant le compilateur de compilateur, comme celle d'installation (`install`), celle de construction d'un compilateur (`build`), etc. Contribuer à XMG-NG avec une commande de ce type constitue également une extension.

Compilateurs

Pour un utilisateur non expert, des solutions métagrammaticales doivent pouvoir être distribuées sans nécessiter un assemblage ou un paramétrage. Dans cette optique, des compilateurs totalement assemblés et prêts à l'utilisation doivent également être à disposition.

Un compilateur est construit à partir de sa description (vue dans 7.2.3). Cette construction consiste en la génération de tous les fichiers de configuration, c'est à dire la gestion des imports de modules, des dimensions, des associations de solveurs aux dimensions.

Création d'une brique

Un squelette de brique peut être généré automatiquement par une commande du générateur de compilateur. Si cette brique contient une brique de langage, elle contient également les modules de compilation qui lui sont associés.

Pour chaque constructeur apparaissant dans les actions sémantiques de la brique de langage, l'unfolder doit comporter une règle. Si le constructeur correspond à un énoncé, cette règle doit provoquer l'accumulation de contraintes. Si le constructeur correspond à une expression, la règle doit à nouveau accumuler des contraintes, mais également renvoyer un identifiant, correspondant à la représentation de l'expression.

Une brique pour les structures de traits. Dans la brique avm, utilisons la brique de langage définie précédemment. Un répertoire rassemble les librairies utiles à la brique. Un module implémentant l'unification ouverte des traits décrite précédemment est l'une de ces librairies.

Contributions

Le compilateur étant composé d'un ensemble de modules, une extension consiste en l'installation de nouveaux modules. Ces modules sont fournis sous la forme de packages appelés contributions. Des modules de plusieurs types peuvent être contribués, puisqu'une contribution apporte un ensemble de fonctionnalités pouvant aller d'un principe à un compilateur complet.

La partie centrale du compilateur est distribuée comme une contribution : la contribution core. Cette contribution contient entre autres la commande pour installer de nouvelles contributions et ainsi étendre le compilateur, ainsi que celle permettant d'assembler des briques de compilateur.

Une contribution peut être dépendante d'autres. Notamment, toute contribution incluant un compilateur est dépendante de la contribution core, qui contient les éléments essentiels de l'approche métagrammaticale de XMG.

La commande `install` permet l'installation d'une contribution. Elle rend disponibles toutes les briques fournies par la contribution pour l'assemblage de compilateurs, mais également ses commandes ou compilateurs.

Annexe B. Application à la musique

Sommaire

Introduction **187**

Le compilateur **188**

Une dimension <music> 188

Principes 189

Des interactions entre dimensions 189

Sortie 190

Une métagrammaire pour les accords musicaux **190**

Intuition 190

Décrire les notes 190

Décrire des Intervalles 192

Décrire des Accords 193

Progressions d'accords 195

Introduction

Ce chapitre présente l'application de l'approche metagrammaticale à la musique ([Petitjean, 2012]), en particulier la description des accords. Le choix de cette grammaire peut s'expliquer de différentes façons : la première est que nous souhaitons prouver que l'utilisation d'XMG peut s'avérer utile dans des domaines extérieurs à la linguitique. Un autre aspect intéressant à propos de ces travaux est le fait que les règles manipulées dans la théorie musicale (le rythme, les harmonies, ...) sont radicalement différentes de celles que nous avons traitées jusqu'ici, meme si la musique peut etre comparée sous beaucoup d'aspects au langage naturel [Lerdahl et Jackendoff, 1983]. Pour la théorie musicale, on pourra se référer à [Grove et Sadie, 1980], par exemple. Néanmoins, si les grammaires à grande couverture peuvent reconnaitre une partie significative du langage, il est difficile d'imaginer qu'il puisse en etre de meme ici. En d'autres termes, une grammaire de phrases musicales ne semble pas etre suffisante pour reconnaitre une partie significative de l'ensemble des mélodies ou des séquences d'accords.

Le compilateur

Une dimension <music>

Dans le cadre des grammaires d'arbres adjoints ou des grammaires d'intéraction, le langage métagrammatical doit permettre de décrire des arbres (parfois sous spécifiés). Pour rappel, ce langage est le suivant:

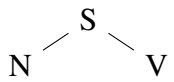
$$\begin{aligned} Desc &:= x \rightarrow y \mid x \rightarrow^{+} y \mid x \rightarrow^{*} y \mid x \prec y \mid x \prec^{+} y \mid x \prec^{*} y \\ &\quad \mid x\,[f{:}E] \mid x\,(p{:}E) \mid Desc\,;\,Desc \mid SynExpr = SynExpr \\ SynExpr &:= var \mid const \mid var.const \end{aligned}$$

Les unités élémentaires que nous manipulons sont les noeuds, qui peuvent etre déclarés avec des traits morphosyntaxiques et des propriétés propres au formalisme (en TAG par exemple, un noeud peut etre déclaré comme noeud pied, ancre, . . .).
Nous devons aussi accumuler des relations entre noeuds, pour décrire la structure de l'arbre. $\rightarrow$ et $\prec$ sont rrespectivement les opérateurs de dominance et de précédence entre noeuds, $^{+}$ and * représentent respectivement les fermetures transitive et reflexive transitive.

Après l'accumulation de la description, cette dernière doit etre résolue. En TAG, le solveur calcule tous les modèles minimaux, c'est à dire qu'aucun noeud présent dans le modèle n'a été ajouté depuis l'accumulation.
Par exemple, la description XMG suivante est celle de 3 noeuds et leurs structures de traits, le noeud S dominant (immédiatement) les noeuds N et V, et N précédant V.

```
class CanSubj
declare ?S ?N ?V
  {
    <syn>{
      node S [cat=s];
      node N [cat=n];
      node V [cat=v];
      S -> N;
      S -> V;
      N >> V;
    }
  }
```

Le seul modèle minimal pour cette classe est le suivant :



Dans ce cas d'application à la musique, les unités élémentaires que nous accumulerons sont les notes. Les notes doivent etre ordonnées (dans une mélodie par exemple, mais nous pourrions aussi vouloir ordonner les notes constituant un accord), alors les structures d'arbres que nous avons décrites jusqu'ici semblent adaptées à la représentation de phrases musicales. La figure 2 montre un exemple d'arbre représentant une séquence d'accords. La racine rassemble

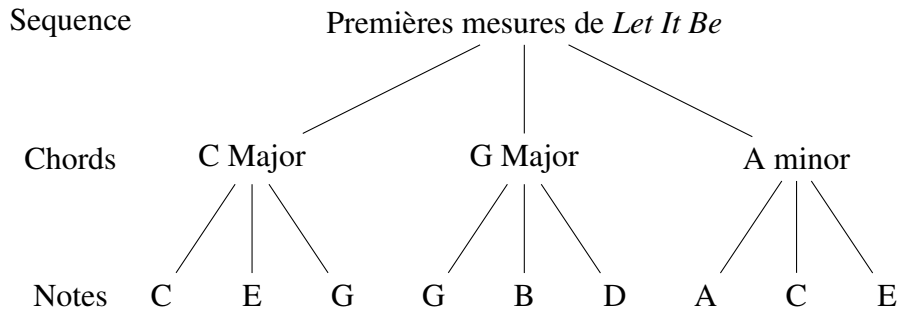


Figure 2 — Une représentation arborescente d'une séquence d'accords

les informations à propos de la séquence d'accords. Ses fils immédiats représentent les accords (par leur ton et leur mode dans l'exemple). Les feuilles sont les notes composant chaque accord.

Une variable de note peut etre déclarée comme un noeud, avec une structure de traits contenant l'information qui la caractérise. Ici, notre objectif est de décrire des arbres, tout comme pour la syntaxe du langage naturel. La principale différence est la nécessité de comparer des nombres entiers dans les structures de traits, notamment pour vérifier les intervalles entre les notes. Des opérateurs arithmétiques seront donc ajoutés au langage. Le langage de description musical sera en conséquence le meme que le langage de description d'arbres, à l'exception de l'ajout du '+' et du '-' dans les expressions :

$$
\begin{aligned}
MDesc \;&:= x \rightarrow y \mid x \rightarrow^{+} y \mid x \rightarrow^{*} y \mid x \prec y \mid x \prec^{+} y \mid x \prec^{*} y \\
&\quad \mid x\,[f{:}E] \mid x\,(p{:}E) \mid MDesc\,;\,MDesc \mid MExpr \;=\; MExpr \\
MExpr \;&:= var \mid const \mid var.const \mid MExpr + MExpr \mid MExpr - MExpr
\end{aligned}
$$

Principes

Dans ce travail, le principe arborescent, qui assure que chaque description produite est un arbre bien formé, sera activé. Parmi les principes plus spécifiques à la description musicale, on peut penser à un assurant que chauq mesure est cohérente avec la signature rythmique. Un autre pourrait appliquer des règles de réécriture aux séquences d'accord générées (comme il a été fait dans [Chemillier, 2004]) pour générer un ensemble de variations sur ces séquences.

Des interactions entre dimensions

Comme c'est le cas pour la syntaxe et la sémantique dans certains travaux précédents en traitement automatique des langues, l'accumulation produite par l'éxécution de la métagrammaire peut avoir lieu dans plusieurs dimensions, ces dernières pouvant partager de l'information.
Une mélodie et une séquence d'accords pourraient etre accumulées dans deux dimensions, avec des contraintes pour forcer leur harmonie. Une autre option serait de créer une dimension pour une autre clé : si nous supposons que la grammaire produite est écrite en clé de sol, une autre dimension pourrait contenir l'écriture de l'accumulation en clé de fa. Ceci pourrait aussi etre utilisé pour la transposition, dans le but de rassembler des instruments transposants (par exemple la clarinette en si bémol et le saxophone alto en mi bémol). Une autre dimension pourrait aussi encoder les doigtés pour réaliser les notes (ou accords) sur certains instruments. L'accumulation

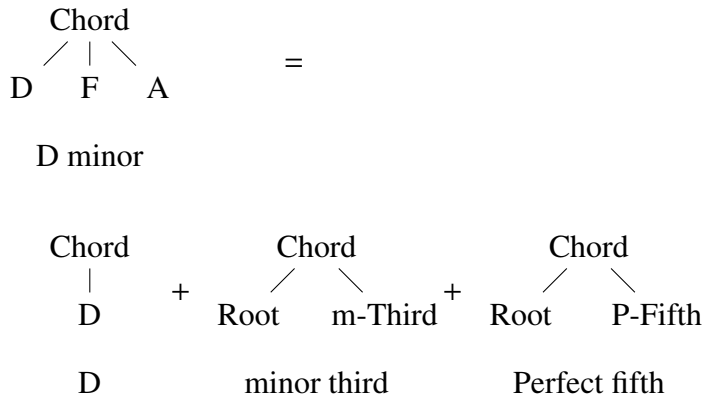


Figure 3 — Combinaison de fragments pour un accord en XMG

d'une représentation sémantique dans une dimension dédiée est aussi une perspective intéressante pour expérimenter les théories sur la sémantique de la musique. L'utilisation des modes majeur et mineur est par exemple habituellement cité comme influence pour les émotions ressenties par l'auditeur [Juslin et Sloboda, 2009].

Sortie

Comme pour la syntaxe, les descriptions d'arbres doivent etre résolues, mais leur exploration avec une interface graphique n'est pas la solution la plus adaptée. Pour cette raison, un module dédié du compilateur convertit la sortie XML en un fichier LilyPond (`http://www.lilypond.org/`).

Une métagrammaire pour les accords musicaux

Intuition

Comme cela a été fait pour le langage naturel, l'idée est de produire les arbres de la grammaire (ici, des arbres représentant des accords, ou des séquences d'accords) en assemblant des fragments d'arbres. La figure 3 montre un exemple de quelques fragments qui pourraient etre combinés pour produire un accord mineur. Un fragment représentant le choix de la racine est combiné avec un représentant l'intervalle de tierce mineure et un représentant l'intervalle de quinte juste.

Décrire les notes

La première abstraction à créer est l'unité élémentaire de la musique : la note. Depuis une unique classe, nous voulons générer n'importe quelle note, ainsi que l'information qui la caractérise. La structure de traits de la note contient le nom de la note, son altération (ici, nous ne considérerons que les notes ayant au plus une altération) et sa hauteur (en terme de demi-tons). Comme nous ne décrirons pas le rythme dans ce travail préliminaire, nous avons choisi de ne pas faire apparaitre la durée de la note. Ces traits ont les types suivants :

```
type NAME=     {A,B,C,D,E,F,G}
```

```
type NAMERANK= [1..7]
type ACCIDENTAL= {sharp,flat,none}
type HEIGHT=   [0..11]

feature name :      NAME
feature namerank :  NAMERANK
feature acc :       ACCIDENTAL
feature height :    HEIGHT
```

Dans la métagrammaire, 'name' est le nom de la note (c'est à dire do, ré ...). Son type est énuméré. 'acc' est l'altération de la note (dièse, bémol, ou sans altération), 'namerank' est un entier associé automatiquement au nom de la note, en commençant par *la*=1, 'height' est le nombre de demi-tons séparant la note du la. 'namerank' et 'height' dépendepent des deux traits précédents, et ne sont utilisés que dans la métagrammaire, pour comparer les notes (ils n'apparaissent pas dans la sortie).
Par exemple, la structure de traits d'un ré dièse sera [name=D, acc=sharp, namerank=4, height=5]. Les octaves ne sont pas prises en compte, ce qui signifie que l'unique sol que nous considérons est à la fois 7 demi tons au dessus du do et 5 demi tons en dessous.

Deux choses ont à etre contraintes :

- le nom et l'entier associé doivent correspondre

- la hauteur dépend de la note et de son altération

```
class note
export N
declare ?V ?ACC ?NR ?H ?HT
{
  <music>{
    N [name=V, namerank=NR, acc=ACC, height=H];
    {
      { V=A ; NR=1 ; HT=1 } |
      { V=B ; NR=2 ; HT=3 } |
      { V=C ; NR=3 ; HT=4 } |
      { V=D ; NR=4 ; HT=6 } |
      { V=E ; NR=5 ; HT=8 } |
      { V=F ; NR=6 ; HT=9 } |
      { V=G ; NR=7 ; HT=11 }
    };
    {
      { ACC=sharp;    H = HT + 1 } |
      { ACC=flat;     H = HT - 1 } |
      { ACC=none;     H = HT     }
    }
  }
}
```

XMG construit une description pour chaque cas. Comme prévu, l'exécution de la classe *Note* (avec 7 notes et 3 altérations) mène à 21 descriptions.

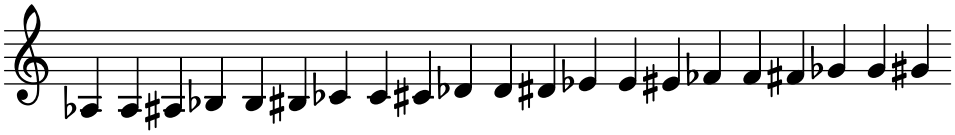


FIGURE 4 — Les 21 notes générées

Décrire des Intervalles

Un intervalle comprend deux notes, et est défini par son nom et son qualificatif. Le nom dépend du nombre de degrés séparant les notes (seconde, tierce, quarte, ...), et le qualificatif du nombre de demi tons entre les notes (majeur, mineur, juste ...). Le nom doit donc etre donné par la différence entre les 'nameranks' des notes, et le qualificatif par la différence entre les hauteurs. Ces deux éléments sont indispensables, puisque C-E est une tierce majeure, mais C-Fb non (meme si les deux intervalles peuvent sonner identiquement, c'est une quarte diminuée, puisque son degré est 4). Nous allons maintenant définir une classe qui décrit un arbre d'intervalle depuis une première note et deux entiers (symbolisant le degré et le qualificatif de l'intervalle).

```
class interval[Root, Unities, SemiTones]
 export RootN OtherN Chord
 declare ?Chord ?RootF ?RRank ?RHeight ?RootN ?RDiff ?SDiff
         ?Other ?OtherF ?ORank ?OHeight ?OtherN ?Name ?Acc
  {
                                Other=note[];
    RootF=Root.Feats;           OtherF=Other.Feats;
    RRank=RootF.namerank;       ORank=OtherF.namerank;
    RHeight=RootF.height;       OHeight=OtherF.height;
    RootN=Root.N;               OtherN=Other.N;
    RDiff= 7 - Unities;         SDiff= 12 - SemiTones;
    Name=RootF.name;            Acc=RootF.acc;
    {
      ORank=RRank + Unities | ORank=RRank - RDiff
    }
    ;
    {
      OHeight=RHeight + SemiTones | OHeight=RHeight - SDiff
    };
    <music>{
      node Chord[name=Name, acc=Acc];
      Chord -> RootN;
      Chord -> OtherN;
      RootN >>+ OtherN
    }
 }
```

Pour créer un intervalle, nous avons besoin d'une instance de la classe précédente. Par exemple, pour représenter une tierce majeure, nous avons à accumuler deux notes séparées d'exctement deux tons. Une première note doit etre déclarée, celle-ci sera la racine de l'intervalle, et un appel à *Interval* doit etre fait, avec la note déclarée, 2 et 4 comme paramètres.

```
class Mthird
 export RootN OtherN Chord
 declare ?I ?Root ?RootN ?Other ?OtherN ?Chord ?V1 ?V2
  {
    Root=note[];
    RootN=Root.N;
    V1=2;
    V2=4;
    I=interval[Root,V1,V2];
    OtherN=I.OtherN;
    Chord=I.Chord;
    <music>{
      node Chord[mode=major]
    }
  }
```

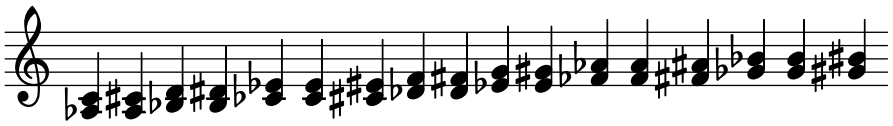


FIGURE 5 — Les 17 tierces majeures générées

Intuitivement, chacune des 21 notes décrites précédemment devrait etre la racine d'une tierce majeure, mais seulement 17 sont générées. La raison est que les quatre intervalles restants impliqueraient des double altérations. Nous appliquons la meme méthode pour la tierce mineure, avec un appel à la classe *interval* paramétrée par 3 demi tons au lieu de 4.

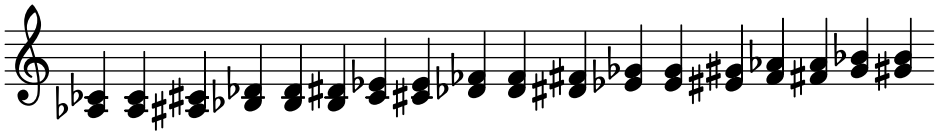


FIGURE 6 — Les 18 tierces mineures générées

Décrire des Accords

Triades. Considérons maintenant les accords à trois notes, appelés triades, dans leurs modes majeur et mineur. Ces derniers sont composés d'une note, appelée fondamentale, d'un intervalle de tierce entre la fondamentale et la deuxième note, et d'une quinte entre la fondamentale et la troisième note. La triade majeure est composée d'une tierce majeure et d'une quinte juste, et la triade mineure d'une tierce mineure et d'une quinte juste. Nous devons donc accumuler deux intervalles, et contraindre leurs bornes inférieures à unifier, le résultat de cette unification étant la fondamentale de l'accord.

```
class Major
 export Chord RootN FNode
 declare ?TNode ?FNode ?RootN ?Chord ?T ?F
  {
```

```
    T=Mthird[];
    F=fifth[];
    Chord=T.Chord;
    RootN=T.RootN;
    FNode=F.OtherN;
    <music>{
      T.OtherN >> F.OtherN;
      T.Chord = F.Chord;
      T.RootN = F.RootN
    }
  }
```

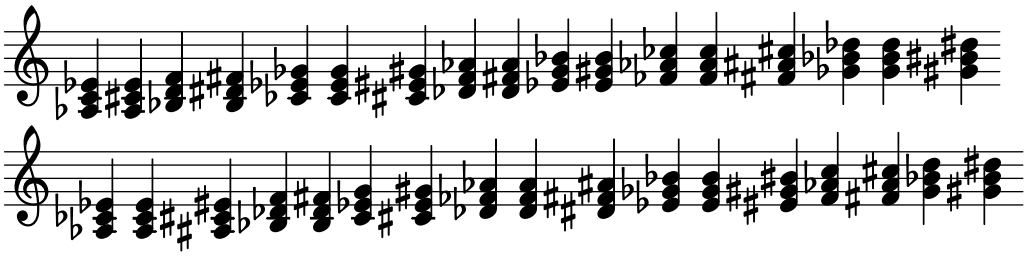


Figure 7 — Les 17 triades majeures et mineures générées

Accords de septième. Les accords de septième sont des accords à quatre notes : les trois notes d'une triade, et une quatrième, formant un intervalle de septième avec la fondamentale de la triade. Dans la spécification XMG, un accord de septième est donc défini comme la conjonction d'une triade et d'un intervalle de septième (la fondamentale de la triade unifiant avec la borne inférieure de l'intervalle). L'intervalle de septième doit donc etre défini.

Un accord de septième dominante est construit avec une triade majeure et un intervalle de septième mineure. Dans une septième mineure, les notes sont séparées de 6 unités et de 10 demi-tons. La classe décrivant la septième dominante essaie toutes les combinaisons entre triades majeures et intervalles de septième diminuée, en unifiant la fondamentale et en ordonnant les notes.

```
class Dseventh
declare ?M ?S
{
  M=Major[];
  S=mseventh[];
  <music>{
    M.Chord = S.Chord;
    M.RootN = S.RootN;
    M.FNode >> S.OtherN
  }
}
```

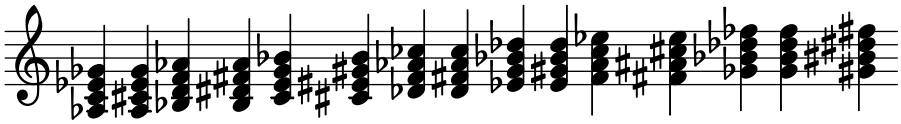


FIGURE 8 — Les 15 accords de septième diminuée générés

Accords relatif. Dans un morceau, lors d'une modulation, ou simplement dans une progression d'accords, il est courant de passer d'un accord à son accord relatif majeur ou mineur. Tout accord majeur a un accord mineur relatif. Sa fondamentale est une tierce mineure en dessous de celle de l'accord majeur. Cette contrainte est la seule nécessaire ici.

```
class Relative
declare ?Major ?Minor ?Third ?Relative
  {
    Major=Major[];
    Minor=minor[];
    Third=mthird[];
    Third.RootN=Minor.Chord;
    Third.OtherN=Major.Chord;
    <music>{
      node Relative;
      Relative -> Major.Chord;
      Relative -> Minor.Chord;
      Minor.Chord >> Major.Chord
    }
}
```

L'exécution de la classe *Relative* produit 16 descriptions, incluant celle de la figure 9.

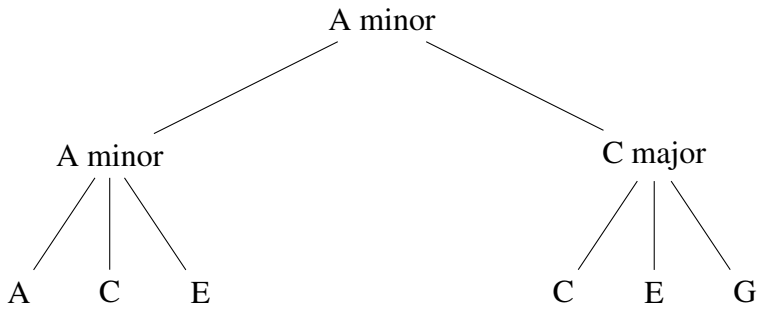


FIGURE 9 — Do majeur, et son relatif, la mineur

Progressions d'accords

Dans une gamme, l'ensemble des notes est inclus dans 3 triades majeures : celles basées sur le premier, le quatrième et le cinquième degrés de la gamme. Nous appellons ces triades accords communs, auquels on doit ajouter leurs mineures relatives. Les progressions d'accords sont généralement basées sur ces 6 accords [Ottman, 1998]. La séquence de la figure 2 est un exemple de progression d'accords en do : il inclue la triade majeure de la tonique (do majeur), sa mineure relative (la mineur), et la triade majeure basée sur le cinquième degré (sol majeur).

Nous avons déjà généré les ensembles d'accords relatifs, le travail restant est de rassembler trois paires d'accords relatifs : les fondamentales des accords majeurs doivent etre séparées de deux tons et demi (entre le premier et le quatrième degré) et trois tons et demi (entre le premier et le cinquième degré).

Bibliographie

[Abeillé, 1991] Abeillé, A. (1991). *Une grammaire lexicalisée d'arbres adjoints pour le français*. Thèse de doctorat, Université de Paris 7.

[Abeillé, 1993] Abeillé, A. (1993). *Les Nouvelles Syntaxes*. Armand Colin, Paris.

[Aho *et al.*, 1998] Aho, A., Sethi, R., Ullman, J. et Boullier, P. (1998). *Compilateurs: principes, techniques et outils*. Informatique Intelligence Artificielle. InterEditions.

[Autebert *et al.*, 1997] Autebert, J., Berstel, J. et Boasson, L. (1997). Context-free languages and push-down automata. *In* Rozenberg, G. et Salomaa, A., éditeurs : *Handbook of Formal Languages*, volume 1, pages 111–174. Springer.

[Bech, 1955] Bech, G. (1955). *Studien über das deutsche Verbum infinitum*. Det Kongelige Danske videnskabernes selskab. Historisk-Filosofiske Meddelelser, bd. 35, nr.2 (1955) and bd. 36, nr.6 (1957). Munksgaard, Kopenhagen. 2nd unrevised edition published 1983 by Max Niemeyer Verlag, Tübingen (Linguistische Arbeiten 139).

[Becker, 1993] Becker, T. (1993). *HyTAG: A new Type of Tree Adjoining Grammars for Hybrid Syntactic Representation of Free Word Order Language*. Thèse de doctorat, Universität des Saarlandes.

[Becker, 2000] Becker, T. (2000). Patterns in Metarules for TAG. *In Tree Adjoining Grammars. Formalisms, Linguistic Analysis and Processing*. CSLI, Stanford.

[Beesley et Karttunen, 2003] Beesley, K. et Karttunen, L. (2003). *Finite State Morphology*. Numéro v. 1 de CSLI studies in computational linguistics: Center for the Study of Language and Information. CSLI Publications.

[Bender *et al.*, 2002] Bender, E. M., Flickinger, D. et Oepen, S. (2002). The grammar matrix: An open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. *In* Carroll, J., Oostdijk, N. et Sutcliffe, R., éditeurs : *Proceedings of the Workshop on Grammar Engineering and Evaluation at the 19th International Conference on Computational Linguistics*, pages 8–14, Taipei, Taiwan.

[Blache, 2000] Blache, P. (2000). Constraints, linguistic theories, and natural language processing. *In* Christodoulakis, D., éditeur : *Natural Language Processing — NLP 2000*, volume 1835 de *Lecture Notes in Computer Science*, pages 221–232. Springer Berlin Heidelberg.

[Blackburn et Bos, 2005] Blackburn, P. et Bos, J. (2005). *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI.

[Bloomfield, 1933] Bloomfield, L. (1933). *Language*. University of Chicago Press.

[Bresnan, 1982] Bresnan, J. (1982). The passive in lexical theory. *In* Bresnan, J., éditeur : *The Mental Representation of Grammatical Relations*. The MIT Press, Cambridge, MA.

[Bresnan, 2001] Bresnan, J. (2001). *Lexical-Functional Syntax*. Blackwell Textbooks in Linguistics. Wiley.

[Bresnan et Kaplan, 1982] Bresnan, J. et Kaplan, R. M. (1982). *The Mental Representation of Grammatical Relations*. The MIT Press, Cambridge MA.

[Brown et Hippisley, 2012] Brown, D. et Hippisley, A. (2012). *Network Morphology: A Defaults-based Theory of Word Structure*. Cambridge Studies in Linguistics. Cambridge University Press.

[Brown *et al.*, 1993] Brown, P. F., Pietra, V. J. D., Pietra, S. A. D. et Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Comput. Linguist.*, 19(2):263–311.

[Candito, 1996] Candito, M. (1996). A Principle-Based Hierarchical Representation of LTAGs. *In Proceedings of the 16th International Conference on Computational Linguistics (COLING'96)*, volume 1, pages 194–199, Copenhagen, Denmark.

[Carpenter, 1992] Carpenter, B. (1992). *The Logic of Typed Feature Structures with Applications to Unification Grammars, Logic Programs and Constraint Resolution*, volume 32 de *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

[Carpenter et Penn, 1996] Carpenter, B. et Penn, G. (1996). Compiling typed attribute-value logic grammars. *In* Tomita, M. et Bunt, H. C., éditeurs : *Recent Advances in Parsing Technology*, pages 145–168. Kluwer Academic Publishers.

[Carpenter et Pollard, 1991] Carpenter, B. et Pollard, C. (1991). Inclusion, disjointness and choice: The logic of linguistic classification. *In Proceedings of the 29th annual meeting on Association for Computational Linguistics*, pages 9–16.

[Chemillier, 2004] Chemillier, M. (2004). Toward a formal study of jazz chord sequences generated by steedman's grammar. *Soft Comput.*, 8(9):617–622.

[Chomsky, 1957] Chomsky, N. (1957). *Syntactic Structures*. Mouton, The Hague.

[Chomsky, 1992] Chomsky, N. (1992). *A Minimalist Program for Linguistic Theory*. MIT occasional papers in linguistics. MIT Working Papers in Linguistics, Department of Linguistics and Philosophy, Massachusetts Institute of Technology.

[Chomsky, 1995] Chomsky, N. (1995). *The Minimalist Program*. Current studies in linguistics series. MIT Press.

[Clark *et al.*, 2010] Clark, A., Fox, C. et Lappin, S. (2010). *The Handbook of Computational Linguistics and Natural Language Processing*. John Wiley & Sons.

[Copestake, 2002] Copestake, A. (2002). *Implementing Typed Feature Structure Grammars*. CSLI Publications, Stanford, CA.

[Crabbé, 2005] Crabbé, B. (2005). *Représentation informatique de grammaires fortement lexicalisées : Application à la grammaire d'arbres adjoints*. Thèse de doctorat, Université Nancy 2.

[Crabbé et Duchier, 2004] Crabbé, B. et Duchier, D. (2004). Metagrammar redux. *In* Christiansen, H., Skadhauge, P. R. et Villadsen, J., éditeurs : *Constraint Solving and Language Processing, First International Workshop (CSLP 2004), Revised Selected and Invited Papers*, volume 3438 de *Lecture Notes in Computer Science*, pages 32–47, Roskilde, Denmark. Springer.

[Crabbé *et al.*, 2013] Crabbé, B., Duchier, D., Gardent, C., Le Roux, J. et Parmentier, Y. (2013). XMG : eXtensible MetaGrammar. *Computational Linguistics*, 39(3):591–629.

[Crabbé *et al.*, 2014] Crabbé, B., Duchier, D., Parmentier, Y. et Petitjean, S. (2014). Constraint-driven Grammar Description. *In* Blache, P., Christiansen, H., Dahl, V., Duchier, D. et Villadsen, J., éditeurs : *Constraints and Language*, pages 93–121. Cambridge Scholar Publishing.

[de La Clergerie, 2005] de La Clergerie, É. V. (2005). Dyalog: a tabular logic programming based environment for nlp. *Constraint Solving and Language Processing*, page 18.

[de Saussure, 1915] de Saussure, F. (1915). *Cours de linguistique générale*. Payot, Paris.

[Debusmann, 2006] Debusmann, R. (2006). *Extensible Dependency Grammar: A Modular Grammar Formalism Based On Multigraph Description*. Thèse de doctorat, Saarland University.

[Duchier *et al.*, 2004] Duchier, D., Le Roux, J. et Parmentier, Y. (2004). The MetaGrammar Compiler: An NLP Application with a Multi-paradigm Architecture. *In Second International Mozart/Oz Conference - MOZ 2004*, volume 3389 de *Lecture Notes in Computer Science*, pages 175–187, Charleroi, Belgium. Springer Verlag.

[Duchier *et al.*, 2005] Duchier, D., Le Roux, J. et Parmentier, Y. (2005). XMG : Un Compilateur de Méta-Grammaires Extensible. *In 12e Conférence Annuelle sur le Traitement Automatiques des Langues Naturelles - TALN 2005*, pages 13–22, Dourdan, France.

[Duchier *et al.*, 2012a] Duchier, D., Magnana Ekoukou, B., Parmentier, Y., Petitjean, S. et Schang, E. (2012a). Décrire la morphologie des verbes en ikota au moyen d'une métagrammaire. *In 19e conférence sur le Traitement Automatique des Langues Naturelles (TALN 2012) - Atelier sur le traitement automatique des langues africaines (TALAf 2012)*, pages 97–106, Grenoble, France. Association pour le Traitement Automatique des Langues. This article has been published in the Proceedings of the JEP-TALN-RECITAL 2012 conference. Available on-line at http://www.jeptaln2012.org/actes/TALAF2012/TALAF2012-2012.pdf.

[Duchier *et al.*, 2012b] Duchier, D., Magnana Ekoukou, B., Parmentier, Y., Petitjean, S. et Schang, E. (2012b). Describing Morphologically-rich Languages using Metagrammars: a Look at Verbs in Ikota. *In Workshop on "Language technology for normalisation of less-resourced languages", 8th SALTMIL Workshop on Minority Languages and the 4th workshop on African Language Technology*, page _, Istanbul, Turkey.

[Duchier et Niehren, 2000] Duchier, D. et Niehren, J. (2000). Dominance constraints with set operators. *In Proceedings of the First International Conference on Computational Logic*, LNCS. Springer.

[Duchier *et al.*, 2011] Duchier, D., Parmentier, Y. et Petitjean, S. (2011). Cross-framework grammar engineering using constraint-driven metagrammars. *In 6th International Workshop on Constraint Solving and Language Processing (CSLP'11)*, pages 32–43, Karlsruhe, Germany.

[Duchier *et al.*, 2012c] Duchier, D., Parmentier, Y. et Petitjean, S. (2012c). Metagrammars as Logic Programs. *In 7th International Conference on Logical Aspects of Computational Linguistics (LACL 2012, demo session)*, pages 1–4, Nantes, France.

[Ekman et Hedin, 2007] Ekman, T. et Hedin, G. (2007). The jastadd system — modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14 – 26. Special issue on Experimental Software and Toolkits.

[Evans et Gazdar, 1989] Evans, R. et Gazdar, G. (1989). Inference in datr. *In Proceedings of the Fourth Conference on European Chapter of the Association for Computational Linguistics*, EACL '89, pages 66–71, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Fillmore, 1982] Fillmore, C. J. (1982). Frame semantics. *In* The Linguistic Society of Korea, éditeur : *Linguistics in the Morning Calm*, pages 111–137. Hanshin Publishing.

[Finkel *et al.*, 2002] Finkel, R., Shen, L., Stump, G. et Thesayi, S. (2002). 1 katr: A set-based extension of datr*. Rapport technique, University of Kentucky.

[Finkel et Stump, 2002] Finkel, R. et Stump, G. (2002). Generating hebrew verb morphology by default inheritance hierarchies. *In Proceedings of the ACL-02 Workshop on Computational Approaches to Semitic Languages*, SEMITIC '02, pages 1–10, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Flickinger, 1987] Flickinger, D. (1987). *Lexical Rules in the Hierarchical Lexicon*. Thèse de doctorat, Stanford University.

[Flickinger, 2000] Flickinger, D. (2000). On building a more effcient grammar by exploiting types. *Natural Language Engineering*, 6(1):15–28.

[Flickinger *et al.*, 1985] Flickinger, D., Pollard, C. et Wasow, T. (1985). Structure-sharing in lexical representation. *In Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*.

[Fradin, 2003] Fradin, B. (2003). *Nouvelles approches en morphologie*. Linguistique Nouvelle. Presses Universitaires de France - PUF.

[Gaiffe *et al.*, 2002] Gaiffe, B., Crabbé, B. et Roussanaly, A. (2002). A new metagrammar compiler. *In Proceedings of the Internation Workshop in Tree-Adjoining Grammar and Related Formalisms (TAG+6)*, pages 101–108, Venice, Italy.

[Gardent, 2008] Gardent, C. (2008). Integrating a Unification-Based Semantics in a Large Scale Lexicalised Tree Adjoining Grammar for French. *In Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 249–256, Manchester, UK. Coling 2008 Organizing Committee.

[Gardent et Kallmeyer, 2003] Gardent, C. et Kallmeyer, L. (2003). Semantic construction in feature-based tree adjoining grammar. *In 10th conference of the European Chapter of the Association for Computational Linguistics*.

[Gardent et Parmentier, 2006] Gardent, C. et Parmentier, Y. (2006). Coreference handling in XMG. *In Proceedings of the COLING/ACL 2006 Main Conference*, pages 247–254, Sydney, Australia. Association for Computational Linguistics.

[Gazdar et Pullum, 1982] Gazdar, G. et Pullum, G. K. (1982). Natural languages and context free languages. *Linguistics and Philosophy*, 4:471–504.

[Geeraerts *et al.*, 2010] Geeraerts, Cuyckens et Janda (2010). Inflectional morphology.

[Goldberg, 2006] Goldberg, A. (2006). *Constructions at Work. The Nature of Generalizations in Language*. Oxford Univ. Press, Oxford.

[Götz *et al.*, 1997] Götz, T., Meurers, D. et Gerdemann, D. (1997). *The ConTroll manual*. Seminar für Sprachwissenschaft, Universität Tübingen, Tübingen. Draft of 17. September 1997 for ConTroll v.1.0b and XTroll v.5.0b.

[Grove et Sadie, 1980] Grove, G. et Sadie, S. (1980). *The New Grove dictionary of music and musicians*. Numéro vol. 13 de The New Grove Dictionary of Music and Musicians. Macmillan Publishers.

[Guénot, 2006] Guénot, M.-L. (2006). *Éléments de grammaire du français pour une théorie descriptive et formelle de la langue*. Thèse de doctorat, Université de Provence.

[Guillaume et Perrier, 2009] Guillaume, B. et Perrier, G. (2009). Interaction Grammars. *Research on Language and Computation*, 7(2-4):171–208.

[Harabagiu et Moldovan, 2003] Harabagiu, S. et Moldovan, D. (2003). Question answering. *In The Oxford Handbook of Computational Linguistics*, chapitre 31. Oxford University Press, Oxford, England.

[Harris, 1955] Harris, Z. (1955). *Methods in Structural Linguistics*. University of Chicago Press.

[Henriques *et al.*, 2005] Henriques, P. R., Pereira, M. J. V., Mernik, M., Lenic, M., Gray, J. et Wu, H. (2005). Automatic generation of language-based tools using the LISA system. *Software, IEE Proceedings* -, 152(2):54–69.

[Hindley, 1969] Hindley, R. (1969). The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60.

[Holzbaur, 1992] Holzbaur, C. (1992). Metastructures versus attributed variables in the context of extensible unification. *In PLILP*, pages 260–268.

[Hulden, 2009] Hulden, M. (2009). Foma: a finite-state compiler and library. *In EACL (Demos)*, pages 29–32.

[Joshi et Schabes, 1997] Joshi, A. K. et Schabes, Y. (1997). Tree adjoining grammars. *In* Rozenberg, G. et Salomaa, A., éditeurs : *Handbook of Formal Languages*. Springer Verlag, Berlin.

[Juslin et Sloboda, 2009] Juslin, P. et Sloboda, P. (2009). *Handbook of Music and Emotion : Theory, Research, Applications: Theory, Research, Applications*. Affective Science. OUP Oxford.

[Kallmeyer *et al.*, 2008] Kallmeyer, L., Lichte, T., Maier, W., Parmentier, Y. et Dellert, J. (2008). Developing a TT-MCTAG for German with an RCG-based Parser. *In The sixth international conference on Language Resources and Evaluation (LREC 08)*, pages 782–789, Marrakech, Morocco.

[Kallmeyer et Osswald, 2012a] Kallmeyer, L. et Osswald, R. (2012a). An analysis of directed motion expressions with Lexicalized Tree Adjoining Grammars and frame semantics. *In* Ong, L. et de Queiroz, R., éditeurs : *Proceedings of WoLLIC*, numéro 7456 de Lecture Notes in Computer Science LNCS, pages 34–55. Springer.

[Kallmeyer et Osswald, 2012b] Kallmeyer, L. et Osswald, R. (2012b). A frame-based semantics of the dative alternation in Lexicalized Tree Adjoining Grammars. *In* Piñón, C., éditeur : *Empirical Issues in Syntax and Semantics 9*, pages 167–184. ISSN 1769-7158.

[Kallmeyer et Osswald, 2013] Kallmeyer, L. et Osswald, R. (2013). Syntax-driven semantic frame composition in Lexicalized Tree Adjoining Grammar. Unpublished manuscript.

[Kaplan et Maxwell, 1996] Kaplan, R. M. et Maxwell, J. T. (1996). Lfg grammar writer's workbench. Rapport technique, Xerox PARC.

[Karttunen, 2003] Karttunen, L. (2003). Computing with realizational morphology. *In* Gelbukh, A. F., éditeur : *CICLing*, volume 2588 de *Lecture Notes in Computer Science*, pages 203–214. Springer.

[Kay, 2002] Kay, P. (2002). An informal sketch of a formal architecture for Construction Grammar. *Grammars*, 5:1–19.

[Koehn, 2010] Koehn, P. (2010). *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA, 1st édition.

[Koskenniemi, 1983] Koskenniemi, K. (1983). *Two-Level Morphology: a general computational model for word-form recognition and production*. Thèse de doctorat, University of Helsinki.

[Koster et Beney, 1991] Koster, C. H. A. et Beney, J. (1991). On the borderline between grammars and programs. *In PLILP*, pages 219–230.

[Kumar, 1990] Kumar, D., éditeur (1990). *Current Trends in SNePS - Semantic Network Processing System, First Annual SNePS Workshop, Buffalo, NY, USA, November 13, 1989, Proceedings*, volume 437 de *Lecture Notes in Computer Science*. Springer.

[Lerdahl et Jackendoff, 1983] Lerdahl, F. et Jackendoff, R. (1983). *A Generative Theory of Tonal Music*. The MIT Press series on cognitive theory and mental representation. Mit Press.

[Lichte *et al.*, 2013] Lichte, T., Diez, A. et Petitjean, S. (2013). Coupling trees, words and frames through XMG. *In Proceedings of the ESSLLI 2013 workshop on High-level Methodologies for Grammar Engineering*.

[Lieber et Štekauer, 2014] Lieber, R. et Štekauer, P., éditeurs (2014). *The Oxford Handbook of Derivational Morphology*. Oxford Handbooks in Linguistics. Oxford University Press, New York.

[Lindén *et al.*, 2009] Lindén, K., Silfverberg, M. et Pirinen, T. A. (2009). Hfst tools for morphology - an efficient open-source package for construction of morphological analyzers. *In* Mahlow, C. et Piotrowski, M., éditeurs : *SFCM*, volume 41 de *Communications in Computer and Information Science*, pages 28–47. Springer.

[Łupkowski et Purver, 2010] Łupkowski, P. et Purver, M., éditeurs (2010). *Aspects of Semantics and Pragmatics of Dialogue. SemDial 2010, 14th Workshop on the Semantics and Pragmatics of Dialogue*. Polish Society for Cognitive Science, Poznań. ISBN 978-83-930915-0-8.

[Magnana Ekoukou, 2014] Magnana Ekoukou, B. (2014). *Description de l'ikota (B25), langue bantoue du Gabon (à paraître)*. Thèse de doctorat, Université d'Orléans.

[Malouf *et al.*, 2000] Malouf, R., Carroll, J. et Copestake., A. (2000). Efficient feature structure operations without compilation. *Natural Language Engineering*, 6(1):29–46.

[Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 édition.

[Mitkov, 2003] Mitkov, R. (2003). *The Oxford Handbook of Computational Linguistics (Oxford Handbooks in Linguistics S.)*. Oxford University Press.

[Montague, 1970] Montague, R. (1970). *English as a formal language*. Ed. di Comunità.

[Ohori, 1995] Ohori, A. (1995). A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895.

[O'Sullivan *et al.*, 2008] O'Sullivan, B., Goerzen, J. et Stewart, D. (2008). *Real World Haskell*. O'Reilly Media, Inc., 1st édition.

[Ottman, 1998] Ottman, R. W. (1998). *Elementary harmony: theory and practice*. Prentice Hall.

[Parmentier, 2007] Parmentier, Y. (2007). *SemTAG : une plate-forme pour le calcul sémantique à partir de Grammaires d'Arbres Adjoints*. Thèse de doctorat, Université Henri Poincaré - Nancy I. Thèse de Doctorat.

[Parmentier *et al.*, 2006] Parmentier, Y., Le Roux, J. et Crabbé, B. (2006). XMG - An expressive formalism for describing tree-based grammars. *In 11th Conference of the European Chapter of the Association for Computational Linguistics - EACL 2006*, pages 103–106, Trento, Italy.

[Pereira et Warren, 1980] Pereira, F. et Warren, D. (1980). Definite clause grammars for language analysis —a survey of the formalism and a comparison to augmented transition networks. *Artificial Intelligence*, 13:231–278.

[Perrier, 2000] Perrier, G. (2000). Interaction Grammars. *In Proceedings of the 18th International Conference on Computational Linguistics (COLING 2000)*, volume 2, pages 600–606, Saarbrücken, Germany.

[Perrier, 2007] Perrier, G. (2007). A French Interaction Grammar. *In* Angelova, G., Bontcheva, K., Mitkov, R., Nicolov, N. et Simov, K., éditeurs : *International Conference on Recent Advances in Natural Language Processing - RANLP 2007*, pages 463–467, Borovets, Bulgarie. IPP & BAS & ACL-Bulgaria, INCOMA Ltd, Shoumen, Bulgaria.

[Petersen, 2007] Petersen, W. (2007). Representation of concepts as frames. *The Baltic International Yearbook of Cognition, Logic and Communication*, 2:151–170.

[Petitjean, 2012] Petitjean, S. (2012). Describing music with metagrammars. *In* Duchier, D. et Parmentier, Y., éditeurs : *Constraint Solving and Language Processing - 7th International Workshop, CSLP 2012, Orléans, France, September 13-14, 2012, Revised Selected Papers*, volume 8114 de *Lecture Notes in Computer Science*, pages 152–165. Springer.

[Petitjean, 2014] Petitjean, S. (2014). Xmg: A modular metagrammar compiler. *In* Colinet, M., Katrenko, S. et Rendsvig, R., éditeurs : *Pristine Perspectives on Logic, Language, and Computation*, volume 8607 de *Lecture Notes in Computer Science*, pages 36–48. Springer Berlin Heidelberg.

[Pollard et Sag, 1994] Pollard, C. et Sag, I. (1994). *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Stanford : CSLI Publications, Chicago.

[Prolo, 2002] Prolo, C. A. (2002). Generating the xtag english grammar using metarules. *In Proceedings of the 19th international conference on Computational linguistics*, pages 1–7, Morristown, NJ, USA. Association for Computational Linguistics.

[Pullum et Scholz, 2001] Pullum, G. et Scholz, B. (2001). On the distinction between model-theoretic and generative-enumerative syntactic frameworks. *In* de Groote, P., Morrill, G. et Retoré, C., éditeurs : *Logical Aspects of Computational Linguistics*, volume 2099 de *Lecture Notes in Computer Science*, pages 17–43. Springer Berlin Heidelberg.

[Rogers et Vijay-Shanker, 1994] Rogers, J. et Vijay-Shanker, K. (1994). Obtaining trees from their descriptions : an application to tree-adjoining grammars. *Computational Intelligence*, 10:401–421.

[Sagot et Walther, 2013] Sagot, B. et Walther, G. (2013). Implementing a formal model of inflectional morphology. *In* Mahlow, C. et Piotrowski, M., éditeurs : *Third International Workshop on Systems and Frameworks for Computational Morphology*, volume 380 de *Communications in Computer and Information Science*, pages 115–134, Berlin, Allemagne. Humboldt-Universität, Springer.

[Schang *et al.*, 2012] Schang, E., Duchier, D., Magnana Ekoukou, B., Parmentier, Y. et Petitjean, S. (2012). Describing São Tomense Using a Tree-Adjoining Meta-Grammar. *In 11th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+11)*, pages 82–89, Paris, France.

[Shieber, 1984] Shieber, S. M. (1984). The design of a computer language for linguistic information. *COLING-84*, pages 362–366.

[Shieber, 1985] Shieber, S. M. (1985). Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343. Reprinted in Walter J. Savitch, Emmon Bach, William Marsh, and Gila Safran-Navah, eds., *The Formal Complexity of Natural Language*, pages 320–334, Dordrecht, Holland: D. Reidel Publishing Company, 1987. Reprinted in Jack Kulas, James H. Fetzer, and Terry L. Rankin, eds., *Philosophy, Language, and Artificial Intelligence*, pages 79–92, Dordrecht, Holland: Kluwer Academic Publishers, 1988.

[Sowa, 1991] Sowa, J., éditeur (1991). *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, San Mateo, CA.

[Spencer et Zwicky, 2001] Spencer, A. et Zwicky, A. (2001). *The Handbook of Morphology*. Blackwell Handbooks in Linguistics. Wiley.

[Steedman, 2000] Steedman, M. (2000). *The Syntactic Process*. MIT Press, Cambridge.

[Stump, 2001a] Stump, G. (2001a). *Inflectional morphology: a theory of paradigm structure*. Cambridge Univ Pr.

[Stump, 2001b] Stump, G. T. (2001b). *Inflectional Morphology: a Theory of Paradigm Structure*, volume 93. Cambridge University Press.

[Tesnière, 1959] Tesnière, L. (1959). *Eléments de Syntaxe Structurale*. Klincksieck, Paris.

[Thomasset et Clergerie, 2005] Thomasset, F. et Clergerie, É. V. D. L. (2005). Comment obtenir plus des méta-grammaires. *In Proceedings of TALN'05*, Dourdan, France. ATALA, ATALA.

[Vacchi *et al.*, 2013] Vacchi, E., Cazzola, W., Pillay, S. et Combemale, B. (2013). Variability Support in Domain-Specific Language Development. *In* Erwig, M., Paige, R. F. et Van Wyk, E., éditeurs : *SLE - 6th International Conference on Software Language Engineering*, volume 8225 de *Lecture Notes in Computer Science*, pages 76–95, Indianapolis, IN, États-Unis. Springer. VaryMDE (bilateral collaboration with Thales Research & Technology).

[Van Roy, 1990] Van Roy, P. (1990). Extended dcg notation: A tool for applicative programming in prolog. Rapport technique, Technical Report UCB/CSD 90/583, UC Berkeley.

[Van Roy et Despain, 1992] Van Roy, P. et Despain, A. M. (1992). High-performance logic programming with the aquarius prolog compiler. *Computer*, 25(1):54–68.

[Walther, 2012] Walther, G. (2012). Measuring morphological canonicity. *Linguistica*, 51:157–180.

[Watt, 1974] Watt, D. (1974). *Analysis-oriented Two-level Grammars, by David Anthony Watt*. Glasgow University.

[Xia, 2001] Xia, F. (2001). *Automatic Grammar Generation from two Different Perspectives*. Thèse de doctorat, University of Pennsylvania.

[Xia *et al.*, 2010] Xia, F., Palmer, M. et Vijay-Shanker, K. (2010). Developing tree-adjoining grammars with lexical descriptions. *In* Bangalore, S. et Joshi, A. K., éditeurs : *Using Complex Lexical Descriptions in Natural Language Processing*, pages 73–110. MIT Press, Cambridge.

[XTAG Research Group, 2001] XTAG Research Group (2001). A Lexicalized Tree Adjoining Grammar for English. Rapport technique IRCS-01-03, IRCS, University of Pennsylvania.

Simon PETITJEAN

Génération Modulaire de Grammaires Formelles

Résumé : Les travaux présentés dans cette thèse visent à faciliter le développement de ressources pour le traitement automatique des langues. Les ressources de ce type prennent des formes très diverses, en raison de l'existence de différents niveaux d'étude de la langue (syntaxe, morphologie, sémantique, ...) et de différents formalismes proposés pour la description des langues à chacun de ces niveaux. Les formalismes faisant intervenir différents types de structures, un unique langague de description n'est pas suffisant : il est nécessaire pour chaque formalisme de créer un langage dédié (ou DSL), et d'implémenter un nouvel outil utilisant ce langage, ce qui est une tâche longue et complexe.
Pour cette raison, nous proposons dans cette thèse une méthode pour assembler modulairement, et adapter, des cadres de développement spécifiques à des tâches de génération de ressources langagières. Les cadres de développement créés sont construits autour des concepts fondamentaux de l'approche XMG (eXtensible MetaGrammar), à savoir disposer d'un langage de description permettant la définition modulaire d'abstractions sur des structures linguistiques, ainsi que leur combinaison non-déterministe (c'est à dire au moyen des opérateurs logiques de conjonction et disjonction). La méthode se base sur l'assemblage d'un langage de description à partir de briques réutilisables, et d'après un fichier unique de spécification. L'intégralité de la chaîne de traitement pour le DSL ainsi défini est assemblée automatiquement d'après cette même spécification.
Nous avons dans un premier temps validé cette approche en recréant l'outil XMG à partir de briques élémentaires. Des collaborations avec des linguistes nous ont également amené à assembler des compilateurs permettant la description de la morphologie de l'Ikota (langue bantoue) et de la sémantique (au moyen de la théorie des frames).

Mots clés : Traitement automatique des langues, Langages dédiés, Modularité.

Modular Generation of Formal Grammars

Abstract: The work presented in this thesis aim at facilitating the development of resources for natural language processing. Resources of this type take different forms, because of the existence of several levels of linguistic description (syntax, morphology, semantics, ...) and of several formalisms proposed for the description of natural languages at each one of these levels. The formalisms featuring different types of structures, a unique description language is not enough: it is necessary to create a domain specific language (or DSL) for every formalism, and to implement a new tool which uses this language, which is a long a complex task.
For this reason, we propose in this thesis a method to assemble in a modular way development frameworks specific to tasks of linguistic resource generation. The frameworks assembled thanks to our method are based on the fundamental concepts of the XMG (eXtensible MetaGrammar) approach, allowing the generation of tree based grammars. The method is based on the assembling of a description language from reusable bricks, et according to a unique specification file. The totality of the processing chain for the DSL is automatically assembled thanks to the same specification.
In a first time, we validated this approach by recreating the XMG tool from elementary bricks. Some collaborations with linguists also brought us to assemble compilers allowing the description of morphology and semantics.

Keywords: Natural language processing, Domain specific languages, Modularity.

Laboratoire d'Informatique Fondamentale d'Orléans
Bâtiment 3IA, rue Léonard de Vinci, B.P. 6759
45067 ORLEANS cedex 2, FRANCE