Génie logiciel pour le génie linguiciel

Mathieu Lafourcade

THÈSE
présentée par

Mathieu Lafourcade

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER — GRENOBLE 1

(ARRÊTÉS MINISTÉRIELS DU 5 JUILLET 1984 ET DU 30 MARS 1992)

Spécialité
INFORMATIQUE

Génie Logiciel pour le Génie Linguiciel

1er décembre 1994

Jury :

Président	Farid	OUABDESSELAM
Directeur	Christian	BOITET
Rapporteurs	Jean-Claude	BOUSSARD
	Pierre	COINTE
Examinateurs	Jacques	COURTIN
	Joëlle	COUTAZ

THÈSE PRÉPARÉE AU SEIN DU LABORATOIRE GETA (IMAG, UJF & CNRS)

21eme épreuve
Portable Digital Format
Version finale - Final version
Novembre 1994 - November 1994

Remerciements

C'est avec chaleur que je voudrais ici remercier toute l'équipe du GETA, qui m'a apporté aide, sympathie, gentillesse et humour. Pour Gilles et Hervé, je pense avec joie à certains fou rires salvateurs.

Je souhaite remercier particulièrement mon "chef", Christian Boitet, pour son intuition dans le choix d'un sujet passionnant, pour sa patience et son appui. Le tout sans rentrer en transe ou me morigéner.

Je remercie également François Peccoud et Internet, sans qui rien n'aurait été possible.

Je n'oublie surtout pas les membres du jury, Jean-Claude Boussard, Pierre Cointe, Jacques Courtin, Joëlle Coutaz et Farid Ouabdesselam qui ont bien voulu être rapporteurs, invités et président et lire le manuscrit en l'accompagnant de nombreuses suggestions pertinentes.

C'est avec reconnaissance que je pense à Arno qui a une part non négligeable dans cette aventure. J'ai une pensée affectueuse envers tous les amis et amies qui ont subi avec abnégation les multiples tentatives d'explication de mon sujet.

Enfin, Brigitte et Pierrette pour leur disponibilité et leur tendre soutien durant ces années difficiles qu'elles ont dû ressentir comme particulièrement ennuyeuses.

D'avance, je remercie Zaharin, Kim, Sally, et toute l'équipe de l'UTMK.

Table des matières

Liste des figures

Liste des acronymes, abréviations, …

API	Application Programming Interface
ATN	Augmented Transition Network
ATR	Advanced Telephony Research
CLOS	Common Lisp Object System
CMU	Carnegie Mellon University
DAG	Direct Acyclic Graph
DOP	Dictionary Object Protocol
EGL	Environnement de Génie Logiciel
GAM	Generic Automaton Manager
GETA	Groupe d'Étude pour la Traduction Automatique
GFP	Geta Frame Protocol
GPSG	Generalized Phrase Structure Grammar
HPSG	Head-driven Phrase Structure Grammar
IMAG	Institut de Mathématiques Appliquées de Grenoble
KBMT	Knowledge Based Machine Translation
LEAF	Lattice, Engine And Feature
LFG	Lexical Functional Grammar
LGI	Laboratoire de Génie Informatique
LIDIA	Large Internationalisation de Documents par Interaction avec l'Auteur
LSPL	Langage Spécialisé pour la Programmation Linguistique
LT	Langage de Transcription
MOP	MetaObject Protocol

ODILE	Outil d'Intégration Extensible de Dictionnaires et de Lemmatiseurs
THAM	Traduction Humaine Assistée par Ordinateur
TA	Traduction Automatique
TALN	Traitement Automatique des Langues Naturelles
TAFD	Traduction Automatisée Fondée sur le Dialogue
TAO	Traduction Assistée par Ordinateur
WOOD	William's Object Oriented Database

Introduction

Quelles techniques de Génie Logiciel doit-on et peut-on mettre en œuvre pour spécifier et implémenter des systèmes de Traitement Automatique des Langues Naturelles (TALN) en général, et de Traduction Automatique (TA) en particulier ? Le génie logiciel aborde la conception (analyse et implémentation) d'un système informatique sous plusieurs angles : définition des fonctionnalités, définition des composants, définition des relations entre composants. Ces techniques peuvent évidemment s'appliquer aux systèmes de TALN, mais notre premier souci est la réutilisabilité des composants.

Un système de TA (de seconde génération) est décomposable en outils informatiques qui permettent à des "développeurs linguistes" (grammairiens et lexicographes) de construire des applications linguistiques.

Une application linguistique (par exemple un traducteur de documents techniques d'aviation entre l'anglais et le français) requiert non seulement une expertise linguistique (français et anglais), mais aussi une expertise liée au domaine d'application. Dans les systèmes de première génération, ces connaissances étaient directement "câblées" dans les programmes informatiques.

Les systèmes de TA sont relativement lourds. Par exemple, un générateur de systèmes de TA de seconde génération comme ARIANE-G5 ([Boitet, Guillaume & Quézel-Ambrunaz 1985], [Guilbaud 1990] et [Quézel-Ambrunaz 1989]) a demandé un effort d'environ 25 hommes/années pour développer la partie logicielle et au moins autant pour chaque application linguistique (un couple de langues dans un seul sens).

Bien qu'il s'agisse d'un générateur et que des applications de niveau pré-industriel aient été réalisées, c'est aussi un outil de laboratoire. De manière générale, les systèmes de TALN (issus de laboratoires) doivent pouvoir être à la base de véritables applications utilisables, tout en servant à la recherche. Pour cela, il est nécessaire que ces systèmes soient capables d'évoluer du point de vue informatique. Cette évolution consiste à pouvoir dériver et intégrer de nouveaux outils, en assurant une compatibilité ascendante.

C'est ce qui s'est passé durant les différentes étapes du développement d'ARIANE-G5, et en particulier lors du passage de CETA à ROBRA (77-78), puis d'ARIANE-78 à ARIANE-G5 (88-90), où de nouveaux outils ont été construits, et ont remplacé les anciens de façon quasi-transparente pour les linguistes. Cependant, l'expérience montre que des évolutions plus incrémentales et plus rapides sont souhaitables. Deux voies sont possibles et complémentaires. Tout d'abord, on peut offrir aux linguistes, à l'intérieur du système, des moyens de faire évoluer la syntaxe et la sémantique des langages spécialisés. Comme ils n'ont que peu (ou pas du tout) d'expertise informatique en dehors des outils et des modèles qu'ils ont l'habitude d'utiliser, la deuxième voie consiste à trouver des techniques logicielles qui permettent de répondre à de nouveaux besoins.

Le passé déjà long du TALN et de la TA permet de prendre du recul sur les Langages Spécialisés pour la Programmation Linguistique (LSPL). Il ne s'agit pas ici de savoir si leur usage se justifie, ni d'en inventer des types radicalement nouveaux, mais plutôt de déterminer les approches logicielles les plus efficaces pour les développer et les faire évoluer. Partant des divers types d'objets manipulés par les LSPL existants, nous avons défini LEAF, un modèle à objets qui permet de rendre compte de l'architecture de la plupart des systèmes modernes de TALN. LEAF répartit les composants qui interviennent dans un système de TALN en trois grandes classes : les treillis servent de représentation à des informations géométriques, les décorations sont liées aux informations algébriques, et les moteurs modifient les treillis et les décorations.

En liaison avec des linguistes, nous avons fait plusieurs expériences avec des LSPL existants en les reprenant pour les améliorer, la plupart du temps en CLOS (Common Lisp Object System). Ces expériences montrent clairement que l'intégrabilité, la généricité et l'extensibilité sont trois propriétés essentielles, tant au niveau de la définition des LSPL que de leur implémentation.

L'intégrabilité est la possibilité d'ajouter facilement de nouveaux composants dans un système. ODILE a été notre première expérience de "carrossage générique". Il s'agissait d'intégrer dans un même logiciel deux composants existants — un lemmatiseur et un outil dictionnaire — qui n'avaient pas été développés dans cette perspective, et devaient être facilement remplaçables.

Dans un cadre plus général, la difficulté vient de l'importante variété et de la grande taille des composants que l'on peut souhaiter intégrer. Les nombreux systèmes de TA existants illustrent les approches possibles. Nous avons approfondi le modèle de "tableau blanc" [Boitet & Seligman 1994] et obtenu ainsi une architecture qui semble bien résoudre les problèmes d'intégration. Ce modèle permet de définir un mode général de communication entre les composants. Nous raffinons notre tableau blanc en choisissant une structure de treillis LEAF appropriée comme support de l'information.

Une double réalisation (à base de classes et à base de prototypes) d'un Langage de Représentation Linguistique nous a permis d'approfondir l'analyse de la généricité dans les LSPL.

Les Langages de Représentation constituent une sous-classe des LSPL. Nous avons centré notre étude sur les formalismes de structures à attributs. Le plus souvent, il s'agit de traits booléens ou attributs simples. On trouve aussi des structures de traits complexes, typés ou non. Enfin, certains systèmes dont le cadre dépasse le TALN utilisent des prototypes.

Les frames, que nous considérons comme des sous-classes de prototypes, offrent une généricité bien adaptée à l'implémentation d'un système de décorations linguistiques. Cependant, il semble que les frames ne soient pas assez contraintes pour bien guider les développeurs linguistes. C'est pourquoi nous proposons une version étendue de la notion traditionnelle de "décoration", et l'implémentons en réalisant le langage DÉCOR. En définissant les types de décoration comme des contraintes sur les valeurs des traits, nous retrouvons les avantages des langages de classes tout en préservant l'unicité de la représentation. Notre extension consiste à ajouter aux "valeurs immédiates" des "valeurs par référence" et des "valeurs par formule". DÉCOR offre aussi la dynamicité, c'est-à-dire la possibilité de changer les relations entre prototypes durant l'exécution, ce qui peut être très utile lors de la mise au point de "linguiciels".

Les LSPL doivent être génériques et implémentés à partir de composants, eux-mêmes génériques. Dans notre première implémentation de DÉCOR, nous avons utilisé directement CLOS, qui n'a pas paru très bien adapté. Dans la seconde, plus satisfaisante, nous avons introduit une couche de prototypes, eux-mêmes écrits en CLOS. Nous avons aussi envisagé une réalisation à base de prototypes, mais fondée sur le MetaObject Protocol (MOP) offert par CLOS, et nous avons évalué son effet sur la généricité.

Nous avons analysé la nature et les limites de l'extensibilité quand on veut l'appliquer aux LSPL, en entreprenant la réingénierie de deux LSPL d'ARIANE-G5, ATEF et ROBRA. ATEF est un langage qui permet d'écrire des analyseurs morphologiques. Nous avons déterminé quelles fonctionnalités de l'ATEF existant pouvaient être étendues et généralisées avec profit. En découplant les informations linguistiques et le moteur, nous avons alors obtenu un nouvel outil dont les deux composants sont réellement modulaires. Les fonctions de contrôle du non-déterminisme peuvent maintenant être étendues grâce à un protocole extensible de "Piles à Filtres Dynamiques". La possibilité de produire différentes formes de sortie et d'en définir de nouvelles a aussi été réalisée de manière particulièrement simple, grâce à un autre protocole extensible.

ROBRA est un langage qui permet d'écrire des systèmes transformationnels travaillant sur des arbres décorés. Nous avons réutilisé tout ce qui avait été fait pour ATEF, en profitant de la généricité du gestionnaire d'automates.

En "génie logiciel pour le génie linguiciel", il faut non seulement offrir des "boîtes à outils", mais aussi se donner les moyens de construire des "familles d'outils", c'est-à-dire se doter de protocoles extensibles. Pour rendre des protocoles extensibles tout en continuant à les comprendre, nous avons dû les "stratifier", c'est-à-dire les définir selon plusieurs niveaux d'utilisation. Cependant, même avec cette approche modulaire, on atteint vite une "barrière de complexité statique". Les programmes obtenus ne sont pas plus lents, mais deviennent de plus en plus difficiles à comprendre, et donc à étendre dans le futur.

Pour anticiper sur notre conclusion, disons seulement ici que le “génie logiciel pour le génie linguiciel” s’est révélé être un domaine spécifique et important du génie logiciel. Nous espérons que cette étude, théorique et pratique, contribuera à trouver les “meilleures doses” des diverses techniques modernes visant à produire des outils plus intégrables, plus génériques et plus extensibles.

Première partie

Une première approche théorique et pratique
des problèmes du génie logiciel
pour le génie linguiciel

Introduction de la première partie

Les concepteurs de systèmes de traitement automatique des langues naturelles (TALN) et de Traduction Automatique (TA) ont depuis longtemps ressenti le besoin de fournir aux linguistes des Langages Spécialisés pour la Programmation Linguistique (LSPL)[1]. Contrairement aux langages de programmation généraux, les LSPL offrent des structures de données et de contrôle bien adaptées aux traitements linguistiques envisagés, et elles seules. De plus, on peut souvent fonder la sémantique des LSPL sur des modèles décidables, ce qui permet d'éviter certaines chausse-trappes des langages généraux, comme par exemple le problème de l'arrêt. En effet, les sources d'indécidabilité sont alors détectables statiquement. Toutefois, en pratique, certaines extensions ajoutées aux LSPL leur font perdre cet avantage.

Une question cruciale est de savoir quelle méthodologie de développement il est souhaitable d'adopter pour les systèmes de TA et les LSPL.

Avant d'envisager différentes approches pour la réalisation de systèmes de TALN, il est nécessaire de bien en cerner les composants. Le modèle LEAF identifie les treillis, les décorations et les moteurs comme les trois principales classes d'objets autour desquelles s'articule un système de TALN. Tous les LSPL connus utilisent des réalisations de ces classes d'objets.

Trois critères sont également considérées comme souhaitables. L'*intégrabilité* consiste à pouvoir ajouter aisément des éléments à un système. Les limites et les contraintes de cette notion doivent être explicitées. La *généricité* vise à concevoir des langages qui puissent trouver de multiples utilisations. La généricité visée ici n'est pas contradictoire avec le caractère "spécialisé" des LSPL. Enfin, l'*extensibilité* d'un LSPL est la possibilité d'en étendre aisément les fonctionnalités.

---

1 Sauf indications contraires, nous parlerons dans la suite simplement de *systèmes* pour "systèmes de TALN". Certains systèmes sont des *générateurs* qui permettent de concevoir des *applications*. Un système est généralement constitué de plusieurs *composants*. Les *LSPL* sont des langages de programmation qui permettent de définir ces composants.

Avec ODILE, nous avons proposé la notion de *carrossage générique* qui apporte une réponse spécifique au problème de l'intégration, par le biais de structures interfaces (API). En généralisant, on arrive à l'architecture de "tableau blanc" qui convient à des systèmes plus complexes et plus hétérogènes. C'est également un moyen de faire communiquer les trois composants définis par le modèle LEAF.

La généricité et l'extensibilité sont concrètement illustrées avec la réingénierie du langage LT. Nous constatons alors que ces deux caractéristiques posent de véritables problèmes de Génie Logiciel.

Chapitre 1

Génie linguistique et langages spécialisés

Pour un travail linguistique lourd comme la construction de systèmes de TA, il faut proposer aux linguistes des langages spécialisés, fondés sur des modèles décidables, plutôt que des langages algorithmiques, fonctionnels ou logiques généraux.

B. Vauquois

Les Langages Spécialisés pour la Programmation Linguistique (LSPL) permettent à des développeurs linguistes de réaliser des outils linguistiques (analyseurs, générateurs, correcteurs orthographiques, etc.). Les LSPL ont rapidement été sentis comme une alternative nécessaire aux langages de programmation généraux pour le développement de systèmes de Traitement des Langues Naturelles (TALN) ou de Traduction Automatique (TA). Leur histoire est longue et les approches adoptées pour leur conception sont variées.

L'étude des systèmes de TA (on pourra se référer à [Hutchins 1986], [Hutchins & Somers 1992] ou encore [Balkan 1992]) permet de déterminer pourquoi et comment ont été conçus les LSPL existants. On peut aussi essayer d'en déduire quelles sont les caractéristiques souhaitables d'un LSPL.

Les LSPL sont conçus pour l'implémentation d'outils linguistiques qui disposent de composants de nature variée. Cependant, il est souvent difficile de catégoriser clairement les composants d'un système de TALN. Plusieurs questions viennent à l'esprit. Un seul formalisme général doit-il servir de base à l'ensemble d'un système (c'est l'approche de PLNLP [Jensen, Heidorn & Richardson 1993]) ou au contraire est-il souhaitable de disposer de plusieurs LSPL spécifiques (c'est la thèse de [Vauquois & Boitet 1985a]) ? Quels sont les composants d'un système de TALN, et dans quelle mesure est-il nécessaire qu'ils puissent être développés à partir de langages spécialisés ? Quelle est la frontière entre les LSPL et les langages algorithmiques généraux ? Plus spécifiquement, un LSPL définit-il une structure de contrôle générale ou particulière ? Est-il associé à des structures de données spécifiques ?

À toutes ces questions, on peut répondre selon trois axes : l'intégrabilité, la généricité et l'extensibilité. On ne pourra cependant préciser ces notions qu'après avoir caractérisé les acteurs qui interviennent dans les systèmes de TALN, ce que nous faisons à l'aide du modèle LEAF.

1.1. Langages Spécialisés pour la Programmation Linguistique

Les LSPL sont antérieurs aux Environnements pour le TALN[2]. Dans les années 50, aucun langage de haut niveau, comme ALGOL ou FORTRAN[3], n'existait pour le TALN. La complexité des manipulations liées aux systèmes de l'époque rendait les linguistes dépendants de programmeurs experts qui implémentaient leurs procédures en langage machine (ASM et MacroASM d'IBM). Les linguistes ne tentaient que très rarement de programmer eux-mêmes.

Face à ce problème, une solution a été trouvée au MIT avec le langage COMIT [Hutchins 1986]. Il s'agit, sans doute, du premier LSPL. J. Friedman [Friedman, Bredt, Doran & al. 1971], en présentant son formalisme d'implantation des grammaires transformationelles de Chomsky, parlait plutôt de métalangage et de modèle calculatoire que de langage spécialisé[4].

1.1.0. Motivations liées aux LSPL

Les langages algorithmiques généraux sont, en général, trop puissants et trop peu contraints pour être adaptés au TALN. La définition d'un LSPL cherche à fournir les primitives strictement nécessaires (et suffisantes) à la description et la réalisation d'une application de TALN. Il ne s'agit pas de fournir plus.

Un langage spécialisé fournit un niveau d'abstraction plus élevé (et plus adéquat) que les langages algorithmiques par rapport au domaine auquel il s'intéresse. De plus, il offre aux linguistes un langage qui leur permet de créer de multiples applications et de surmonter quelques unes des limitations inhérentes à la conception initiale.

Structure de données

Les structures de données que manipulent les LSPL sont des chaînes de caractères, des arbres, des graphes, ou encore des structures attribuées. Les objets fournis dans les langages de programmations classiques sont trop généraux ou inadaptés. De plus, les opérateurs qui portent sur ces structures de données ne correspondent pas aux besoins des linguistes.

Par exemple, COMIT a été le premier langage dédié à la gestion de chaînes de caractères et de filtrage (par schémas). La première version a été mise au point en 1957 (deux ans avant la première implémentation raisonnablement utilisable de LISP). Il est à noter que les langages comme LISP et PROLOG (que l'on ne peut pas considérer comme des LSPL) ont été, à l'origine, conçus pour la recherche en intelligence artificielle et en linguistique informatique.

---

2 Ces derniers sont d'ailleurs apparus bien avant les Environnements de Génie Logiciel (EGL).

3 Ces langages dédiés au calcul numérique restent peu adaptés aux manipulations symboliques.

4 La terminologie a évolué. B. Vauquois parlait aussi de "métalangage" en 1967.

Les structures d'enregistrement fournies par les langages généraux (de type PASCAL, par exemple) peuvent sembler proches des structures attribuées mais elles n'offrent pas un certain nombre de possibilités appréciées des linguistes : valeurs par défaut, contraintes sur des co-occurences de traits, héritage. Certains langages à objets se rapprochent beaucoup plus de ce qui serait désirable, mais ne fournissent pas de structures de contrôle adaptées.

Structures de contrôle

Les structures de contrôle d'un LSPL sont adaptées à la recherche et à la résolution d'ambiguïtés. Les différentes formes de non-déterminisme (unaire et n-aire), le filtrage sur des chaînes de caractères ou des arbres sont des aspects importants de la linguistique informatique.

Par exemple, les LSPL ATEF et ROBRA ([Chauché 1975] et [Boitet 1982]) offrent respectivement un non-déterminisme unaire et n-aire.

Langages Spécialisés et Langages Généraux

Un langage spécialisé permet de définir facilement des traitements spécifiques, mais souvent au prix d'un certain manque d'efficacité. De plus, il est nécessaire de développer ce type de langage en même temps qu'un environnement de développement linguiciel adéquat. Est-il toujours justifié d'utiliser des langages spécialisés alors qu'existent de nombreux langages généraux très efficaces ? Un certain nombre d'exemples permettent de fournir des éléments de réponse.

Dans le système SYSTRAN, il n'y a pas de LSPL pour la partie "programme"[5], mais un métalangage a été défini pour les dictionnaires[6]. Cela pose de véritables problèmes de maintenance et portabilité.

Par contre, dans certains cas, l'utilisation d'un LSPL ne se justifie plus. Par exemple, quand l'application linguistique est complètement figée, on peut trouver intéressant de reprogrammer le système dans des langages de programmation généraux plus efficaces. Par exemple, l'analyseur morphologique de TAUM-Aviation a été réécrit en Pascal une fois figé. Certains linguistes-informaticiens qui ont effectivement une double compétence peuvent aussi se permettre des développements en langages généraux. Nous pensons, par exemple à E. Wehrli qui a développé ITS en Modula ([Wehrli 1990], et [Wehrli 1992]).

De manière générale, nous constatons que, dans certains contextes linguistiques simples (l'analyse morphologique de l'anglais, par exemple), le fait que le modèle (et la réalisation) soit complètement figé ou que l'on recherche des méthodes d'implémentation plus efficaces amènent à choisir un langage général pour une nouvelle implémentation. À l'inverse, si le système continue d'évoluer (et ne risque pas d'être figé), ou s'il est complexe, il est intéressant de disposer d'un pouvoir expressif de haut niveau.

De nombreux systèmes sont basés sur des langages spécialisés eux-mêmes implémentés à l'aide de langages de programmation généraux. SPANAM est écrit en ATN (programmé en PL/I.) BEAST (Basic English Analyser System) de Straub et Rogers

---

5 Les linguistes utilisent cependant un jeu de macros spécialisées (en ASM 360/370/390).

6 Les dictionnaires sont de grande taille et représentent un effort de codage considérable.

(1979) est écrit en COBOL. LISP est utilisé à NTT. IBM avec McCord utilise une "slot-grammar" proche des DCG. Ce formalisme est ouvert sur PROLOG[7].

1.1.1. Typologie des langages

Dans la plupart des LSPL, une structure de données particulière est choisie comme base de la représentation linguistique, par exemple, les graphes-Q pour les Systèmes-Q [Colmerauer 1970] ou les cartes dans MIND [Kay 1973]. Ces structures sont presque toujours étiquetées ou décorées. Ici de multiples formalismes peuvent être utilisés. Le calcul de ces structures et de ces décorations peut être fondé sur un modèle décidable.

On peut proposer la nomenclature suivante :

	Structures de données	Opérateurs	Structures de contrôle	Modèle décidable
Langages de description	•			•
Langages de représentation	•	•		•
Langages généraux	•	•	•	
Langages spécialisés	•	?	?	•

Un langage de description se contente de décrire des données. Les langages de description peuvent être opposés aux langages de représentation qui associent des opérateurs aux structures de données. Dans le cadre de notre travail, nous considérerons que l'originalité principale des langages spécialisés est que leurs moteurs sont fondés sur des classes d'automates abstraits, le plus souvent décidables. Pour les LSPL, les structures de données et leur opérateurs éventuels sont adaptés au travail linguistique.

Les langages de représentation de la connaissance constituent une sous-classe des langages de représentation.

Certaines caractéristiques ajoutent une combinatoire dans les types de langage : langage à objets, langage algorithmique, fonctionnel ou logique.

1.1.2. Quelques LSPL

Les langages spécialisés ou les formalismes qui permettent d'exprimer des objets ou des traitements linguistiques sont nombreux. Leur syntaxe vise à faciliter leur compréhension par des linguistes. Le but de cette section est de cerner les caractéristiques d'un bon LSPL et d'analyser les problèmes de réalisation liés à ces langages.

Les LSPL présentés sont : les ATN, les Systèmes-Q, GRADE, et les LSPL des systèmes de TAO METAL et ARIANE.

---

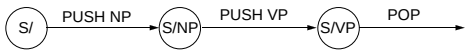
[7] Il est possible de rajouter des clauses PROLOG. Il y a donc au niveau de l'utilisateur linguiste, un mélange entre le formalisme et le langage d'implémentation.

ATN

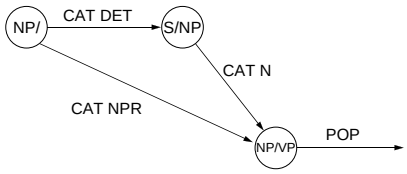
Les ATN (Augmented Transistion Network) ont été définis dans [Woods 1970]. Une implémentation intéressante a vu le jour à BSO (Buro voor System Ontwikkeling) [Vosse, Doedens, Zuijlen & al. 1988]. Le système DLT [Schubert 1988] et le LSPL REZO de TAUM-Aviation [Stewart 1975] ont été développés à partir de ce formalisme.

L'idée principale des ATN est l'utilisation d'un graphe pour la représentation des traitements. La grammaire est organisée selon un réseau consitué d'arcs et de nœuds. Il s'agit en fait d'un automate. L'automate analyse de gauche à droite la suite de symboles grammaticaux. Chaque symbole est examiné et l'automate change d'état en empruntant des transitions (état de départ et d'arrivée reliés par un arc).

S → NP VP



NP → DET N | NPR



Il existe 5 types d'arcs :

- CAT
  définit une transition qui peut être prise si la catégorie du mot courant dans la chaîne d'entrée est celle indiquée ;
- TST
  agit comme CAT mais la condition de transition est exprimée directement ;
- JUMP
  agit comme TST mais ne consomme rien sur la bande d'entrée ;
- POP
  indique que l'état dont est issu l'arc est final. Le traitement sort de la récursion ;
- PUSH
  est l'arc à la base de la récursion.

L'automate peut prendre un arc si la bande d'entrée n'est pas vide et si les conditions portées par l'arc sont vérifiées. Les actions sont exécutées lors du passage de l'état de départ à l'état d'arrivée de la transition. Les actions et les conditions sont basées sur des registres.

Considérons le fragment d'ATN suivant (tiré de [Woods 1970]):

```
(S   (PUSH NP/ T (SETR SUBJ *)
                 (SETR TYPE (QUOTE DCL)) (TO Q1))
     (CAT AUX T  (SETR AUX *) (TYPE TYPE (QUOTE Q)) (TO Q2)))

(Q1  (CAT V T    (SETR AUX NIL)
                 (SETR V *) (TO Q4))
     (CAT AUX T  (SETR AUX *) (TO Q3)))

(Q2  (PUSH NP/ T (SETR SUBJ *) (TO Q3)))

(Q3  (CAT V T    (SETR V *) (TO Q4)))
```

```
(Q4  (POP(BUILDQ(S+++(VP+)) TYPE SUBJ AUX V)T)
     (PUSH PP/   (SETRVP(BUILDQ(VP(V+)*)V)) (TO Q5))

(Q5  (POP(BUILDQ(S++++) TYPE SUBJ AUX VP)+)
     (PUSH NP/T T  (SETR VP(APPEND(GETR VP) (LIST *))) (TO Q5)))
```

Le premier argument de chaque expression est l'état considéré. S est l'état initial. PUSH NP/ est l'appel récursif du sous-réseau d'état initial NP/. T est une condition (ici toujours vrai). Les actions débutent par le symbole SETR. L'état d'arrivée de la transition est Q1 (ou Q2 selon le test).

Systèmes-Q

Les Systèmes-Q[8] [Colmerauer 1970] consituent un exemple de LSPL bien défini. La syntaxe n'est pas ouverte sur un langage général et l'ensemble du système a été implémenté de nombreuses fois.

Le système est réversible si toutes les règles ont les mêmes variables dans les parties droites et gauches.

Voici un exemple simple de systèmes-Q pour l'analyse et la synthèse du français donné par Colmerauer[9]:

```
** FORME ACTIVE OU PASSIVE.
PI(V(A*), I*, J*)  ==   SUJ + I* + V(A*) + OBJ + J*.
              --   ==   SUJ + J* + EST + V(A*) + PPA + I*.

** NOMINALISATION D'UNE PHRASE.
PH(V(A*), I*, J*)  ==   SN(V(A*)) + DDE + J* + PPA + I*.

** FORMES DEFINITIVES DES PREPOSITIONS.
SUJ + PH(U*)       == LE + FAIT + QUE + PI(U*).
PPA + PH(U*)       == PAR + LE + FAIT + QUE + PI(U*).
DDE + PH(U*)       == DE + CE + QUE + PI(U*).
SUJ + SN(I*)       == N(I*).
OBJ + SN(I*)       == N(I*).
PPA + SN(I*)       == PAR + N(I*).
DDE + SN(I*)       == DE + N(I*).

** NOMS CONCRETS.
N(A*)     == A* / A* -DANS- BEAUCEVILLE, BENOIR, GATENATE, RICHARD.

** NOMS ABSTRAITS.
N(V(DETRUITS))     == LA + DESTRUCTION.
N(V(ETONNE))       == L + ETONNEMENT.
N(V(CONSTATE))     == LA + CONSTATATION.

** VERBES.
V(A*)     == A* / A* -DANS- DETRUIT, ETONNE, CONSTATE.
```

Ce système-Q permet de générer toutes les paraphrases d'une phrase donnée (utilisant les mots indiqués). Un traitement-Q (un enchaînement de plusieurs systèmes-Q) a été défini, pour cet exemple, selon une phase d'analyse, une phase de génération, et deux phases pour sortir les résultats en clair.

Le graphe de chaînes qui représente la phrase :

---

8 Q pour Québec.

9 La forme donnée ici permet la génération. En mettant le mot-clé -INV- en tête, on fait marcher les règles de droite à gauche, et on obtient l'analyseur.

```
-0- RICHARD + CONSTATE + QUE + LA + DESTRUCTION + DE + BEAUCEVILLE +
PAR + BENOIT + ETONNE + GAETANE + . -1-.
```

donnera de nombreuses paraphases, dont par exemple :

```
-0- RICHARD + CONSTATE + QUE + LE + FAIT + QUE + BENOIT + DETRUIT +
BEAUCEVILLE + ETONNE + GAETANE + . -1-.
```

GRADE

GRADE [Nakamura 1988] est un LSPL développé dans le cadre du projet MU (1982-1986). Ce langage spécialisé a été utilisé dans le système de TA MU de l'université de Kyoto (1982-86), et dans le système dérivé MAJESTIC opérationnel au JICST depuis 1992. Il s'agit d'un langage d'écriture de systèmes transformationnels fortement inspiré de ROBRA. On écrit en GRADE des règles de réécritures basées sur des transformations d'arbres. Un réseau de sous-grammaires permet d'exprimer le contrôle.

Les règles sont composées de plusieurs parties, par exemple :

```
CHECK_NOUNS.rr;

directory_entry;
  owner(J.NAKAMURA); version(V01L02); last-update(84/4/19);

prop_def;
  J-CAT: type(u), value(NOUN VERB ADJ);

var_import;
  @NONSNP;

var_init;
  @UME;

matching_instruction;
  level(3,10); left_to_right; bottom_to_top; depth;
  order(2, noskip); tree;

matching_condition;
  %((X0 X1 BKK1 X2 BKK2 X3 BKK3 X4 #2));
    X0.NONSP = @NONSP.VAL;
    X1= 'N' | 'NP'; X2= 'N' | 'NP';
    X3= 'N' | 'NP'; X4= 'N' | 'NP';

substructure_operation;
  if X1.J_PASS='YES';
    then @A<=call_sg(CHECK_PATTERN
         %((X2 BKK2 X3 BKK3 X4 #2) list);
    else @A<=call_sg(FOUR_OR_MORE_NOUNS
         %((X1 BKK1 X2 BKK2 X3 BKK3 X4 #2) list);
    end_if;
  #A.UME <= @UME.VAL;

creation;
  if X1.J_PASS='YES';
    then %((X0 X1 BKK1 @A));
    else %((X0 @A));
    end_if;

end_rr.CHECK_NOUNS
```

Figure 1.1 : Exemple de règle de réécriture dans GRADE (règle "CHECK_NOUNS").

La partie entrée de répertoire (*directory_entry*) contient le nom de la grammaire et de son auteur et des informations diverses (version, révision). Cette partie est purement informative et ne participe pas aux traitements.

La partie définition de propriétés (*prop_def*) permet de déclarer les noms des propriétés et leurs valeurs. Les variables globales sont déclarées dans la partie *var_import* et les variables locales dans *var_init.* La variable @NONSP doit être déclarée dans les règles qui utilisent la règle CHECK-NOUNS.rr. La variable @UNM dont la portée est locale n'est visible que dans la règle courante ou les règles appelées.

Certaines variables peuvent être tactiques dans la mesure où elles participent au contrôle de l'application des règles (ces variables n'ont pas d'interprétaion linguistique). Les registres peuvent être utilisés de cette manière dans les ATN. La notion de variable tactique est également présente dans la plupart des applications des LSPL d'ARIANE.

Les directives de filtrage (*matching_instruction* et *matching_condition*) permettent de spécifier le mode et les conditions d'application des règles de réécriture sur les arbres annotés.

La partie transformation spécifie des transductions d'arbres. Elle est composée de trois sous-parties. Une condition portant sur la structure et les valeurs de l'arbres permet de déterminer l'applicabilité de la règle. La partie *substructure_operation* définit des opérations sur l'arbre annoté applicable. La dernière partie (*creation*) spécifie la structure et les valeurs de propriétés de l'arbre annoté transformé.

LSPL de METAL

L'histoire du système METAL peut être divisée en deux périodes : avant et après 1978.

Les travaux sur le système METAL (Mechanical Translation and Analysis of Language) ont débuté en 1961 au Linguistics Research Center (LRC) à l'Université du Texas. Les recherches en linguistique fondamentale effectuées au LRC ont débouché sur une proposition de système de TA entre l'anglais et l'allemand avec règles de transfert syntaxique. Le système est alors réversible avec des règles bidirectionelles.

À partir de 1970, une seconde phase a été l'occasion d'explorer l'approche par pivot (avec un interlingue). Seules les représentations syntaxiques étaient gérées par un interlingue, les éléments lexicaux, eux, utilisant des règles de transfert.

En 1978, SIEMENS devint partenaire du LRC. Les motivations de l'industriel étaient d'augmenter la productivité de ses propres services de traduction mais aussi de proposer des systèmes de TA. L'architecture du système METAL fut alors modifiée pour une approche par transfert augmentée des fonctionnalités sophistiquées d'édition de texte et de banques terminologiques (bases de données TEAM de SIEMENS). Le système utilise des bases grammaticales différentes pour l'analyse et la génération. Il n'est donc plus réversible.

Commercialisé en 1989 pour la traduction allemand-anglais, le noyau du système est conçu pour fonctionner sur des mini-ordinateurs (machine Symbolics™ série 36) accompagnées de stations de travail pour traducteurs (machines SIEMENS SINIX™). Des versions pour micro-ordinateur sont prévues. Des versions avec des couples de langues différents ont également été développées.

L'ensemble des règles utilisées dans l'étape d'analyse d'une application de METAL constitue une grammaire syntagmatique (PSG). Les règles sont augmentées de tests sur les constituants individuels et sur leur interactions [Schneider 1989]. Les règles sont organisées selon des niveaux de priorité.

METAL utilise les structures de cartes (associées à un analyseur à priorités) pour représenter les résultats d'analyse. Des valuations permettent de n'explorer que les chemins les plus prometteurs (les autres chemins sont détruits). Les heuristiques sont mises au point par les linguistes.

Le système de décorations linguistiques de METAL est utilisé dans les dictionnaires monolingues et de transfert (un exemple d'article de dictionnaire est donné au §2.2.2). La définition de structures de traits se fait à l'aide de macro-fonctions LISP (figure 1.4).

```
(defeat a-cl  :set
              :values (FMNP1 FMNP2 FMNP3 FMNPS1 FMNPS2 FMNPS3)
              :allocate 7
              :prettyname "Anaphoric class")

(defcat cat   :optional (A-CL FNMP1 FNMP2 FNMP3)
              :optional ABB
              :required CA
              :optional PLC
              :default (PLC NF)
              :check-consistency t
              :pretty-name "Determiner")
```

Figure 1.2 : Exemples de définition de traits dans METAL.

Le premier trait (*a-cl*) est un ensemble contenant six valeurs possibles et sept combinaisons de valeurs. Le second contient un trait obligatoire et trois traits optionnels. Pour plus de détails sur les fonctions *defeat* et *defcat* voir le § 4.1.2.

Les dictionnaires d'analyse et de génération sont monolingues et contiennent des informations morphologiques, syntaxiques et sémantiques. Des dictionnaires bilingues de correspondances sont utilisés pour le transfert.

Les entrées lexicales dans les dictionnaires monolingues ont la forme de liste de couples attribut/valeur. Les attributs les plus courants sont la forme racine, la catégorie grammaticale, les variations morphologiques, le nombre et la personne. Aux entrées sont associées des préférences qui permettent un choix par défaut en cas d'ambiguïté. Les dictionnaires monolingues sont conçus indépendamment des phases des autres langues employées.

Voici deux exemples d'entrées de dictionnaires monolingues (tirés de [Hutchins & al. 1992]):

```
(Ausgade  CAT  (NST)
          ALO  (Ausgabe)
          PLC  (WI)
          TAG  (DP)
          CL   (P-N S-0)
          GD   (F)
          SX   (N)
          TY   (ABS Dur)
 )
```

```
(output   CAT  (NST)
          ALO  (output)
          PLC  (WI)
          TAG  (DP)
          CL   (P-S S-01)
          ON   (VC)
          SX   (N)
 )
```

Les entrées bilingues lient deux éléments monolingues et peuvent prendre des formes plus complexes en présence de phénomènes contrastifs :

```
(Ausgade (NST DP) 0
         output (NST DP) 0)
```

```
(vor (PREP ALL) 30
       in front of (PREP ALL) 0
       OPT TY * ABS DUR PNT)
 vor (PREP ALL) 20
          before (PREP ALL) 0
          GC D
          TY ABS PNT)
 vor (PREP ALL) 10
          ago (PREP ALL) 0
          GC D
          TY DUR)
 vor (PREP ALL) 0
     in front of (PREP ALL) 0)
 )
```

Le premier exemple (ci-dessus, à gauche) présente une restriction sur le champ de traitement de données (DP).

Dans le second exemple (ci-dessus, à droite), la forme cible anglaise est l'objet de restrictions sur les attributs sémantiques (TY) et de cas grammaticaux (GC). L'élément *in front of* peut être choisi si les noms n'ont pas (* indiquant la négation) pour l'attribut TY les valeurs ABS (abstrait), DUR (forme progressive — *durative*) ou PNT (forme non progressive — *punctual*). "before" peut être choisi si le cas a pour valeur D (Datif) et si TY a pour valeur ABS et PNT. Les valeurs numériques correspondent aux préférences et indiquent l'ordre dans lequel les alternatives doivent être considérées.

```
CLS      PP                  RCL
0        1                   2
(LVL 3)  (REQ CAn * anstatt ob onne statt um) (OPT MD * IMP)
--       --                  (OR (OPT NOAUX NIL)
--       --                  (REQ SPX)
--       --                  (REQ PX NIL))
AUTHOR "Root on 12/07/84 11:50:49"
TEST
     (OR (LCM $)
       (LCM PNCT)
       (LCM (CONJ:1 NIL (REQ CU COR AJT))))
     (OR (RCM $)
       (RCM PNCT)
       (RCM PAR)
       (RCM (CONJ:1 NIL (REQ CU COR AJT))))
     (XFM CLAUSE2)
     (FRM)
CONSTR
     (AND (RET 2 INT)
       (ADD KI WH)
       (ADD MD Q))
INTEGR
     (RES)
ENGLISH
     (AND (INT 2 DA T)
       (SEV 2 CON T))
     (SEF 1 MD)
     (CLSXFR)
     (ORO)
     (XFR)
     ------
((LHS CLS RHS (PP RCL)))
```

Figure 1.3 : Exemple de règle de METAL.

Les règles de METAL sont interprétées par LISP [Bennet & Slocum 1985]. Dans la règle présentée dans la figure 1.3 (tirée de [White 1985]), une sous-routine (FRM) LISP est invoquée dans TEST pour déterminer le sujet, les objets et les arguments d'un paramètre. L'écriture des constituants se fait dans l'ordre correspondant.

La définition et l'extension des LSPL posent de véritables difficultés. Dans le cas de METAL, le formalisme des règles a été mal étendu (et donc mal défini).

L'influence de LISP sur l'ensemble du système a sans doute été trop forte. Les syntaxes des formalismes de définition des structures de données (quand il ne s'agit pas à proprement parler de LSPL) devraient autant que possible cacher le langage d'implémentation.

Le portage en C++ de METAL a buté sur de nombreuse difficultés, essentiellement dues à l'ouverture vers LISP. Les linguistes informaticiens ont tôt fait de rajouter directement des fonctions LISP dans les règles et rendent ainsi l'ensemble du système non portable (vers d'autres langages d'implémentation) et non optimisable.

LSPL dans ARIANE-G5

Le générateur de systèmes de TA ARIANE-G5, dispose de 5 LSPL (ATEF, ROBRA, TRANSF/EXPANS, SYGMOR, TRACOMPL). L'architecture du système est présentée au §2.2.1. Les langages ATEF et ROBRA sont présentés en détail aux §7.1 et §8.1.

Ces langages sont basés sur des transducteurs abstraits. La structure de base est celle d'arbre décoré. ATEF accepte une chaîne en entrée et produit un arbre. SYGMOR, utilisé en génération morphologique, produit une chaîne à partir d'un arbre. Le système de décorations est présenté dans le §4.1.2.

1.1.3. Modèles et techniques d'implémentation

Les techniques d'implémentation, les modèles sous-jacents et les langages de programmation ont une influence prépondérante sur les caractéristiques mais aussi l'histoire des LSPL. Les LSPL se basent en général sur des modèles linguistiquement motivés. Ces modèles ont avantage à être décidables.

Modèles décidables

Le modèle de base auquel nous nous intéressons est celui du transducteur. D'autres approches existent dont celle des analyseurs ("parser"), la construction de graphes de dépendance à partir de tables de dépendance, etc. Nous privilégions l'utilisation de transducteurs car un analyseur se borne à accepter ou refuser son entrée et rien n'est produit en cas d'échec. En pratique, les textes d'entrée ne correspondent souvent que partiellement aux grammaires écrites par les développeurs linguistes (et encore moins aux grammaires normatives des théoriciens), et un transducteur peut traiter ce type d'entrée et rendre un résultat dans tous les cas.

On peut distinguer plusieurs catégories de modèles (formels) utilisables pour des LSPL d'analyse. Ces modèles sont tous basés sur l'écriture de règles.

- Transducteurs d'États Finis (FSA)
- Réseaux de Transitions Augmentées (ATN)
- Grammaires Hors-Contexte Augmentées (ACFG)
- Systèmes Transformationnels d'Arbres (STA)

Quelques LSPL représentatifs peuvent ainsi être catégorisés :

Modèle	LSPL/Système	LI	Caractéristiques	Références
FSA	ATEF	Assembleur	Non-déterminisme n-aire	[Chauché 1975]
ATN	ATN	LISP	Extension des automates à pile	[Woods 1970]
	REZO	Pascal	Spécialisation et généralisation des ATN[10]	[Stewart 1978]
	AL/DLT	C	Extension des ATN (contrôle)	[Doedens & Zuijlen 1988]
ACFG	DCG	Prolog	Règles HC converties en clauses PROLOG	[Pereira & Warren 1986]
	SAX	Prolog	DCG parallèle	[Matsumoto & Sugimura 1987], [Akasaka, Kubo, Fukumoto & al. 1989]
	Règles METAL	LISP	Très dépendant de LISP	[White 1985]
	PATR-II	LISP	Unification de DAG. Il s'agit surtout d'un format d'écriture	[Shieber 1986]
STA	Systèmes-Q	Fortran	Règles de réécriture pour DAG.	[Colmerauer 1970]
	CETA	Assembleur	Ancêtre de ROBRA	[Chauché 1975]
	ROBRA	PL/360	Réécriture d'arbre décorés	[Boitet 1982]
	GT	Fortran		[Friedman & al. 1971]
	GRADE	LISP	Proche de ROBRA	[Nakamura 1988]

La colonne LI indique le Langage d'Implémentation (ces considérations sont discutées au § 1.1.3.).

De manière générale, les ATN ont la puisssance d'un machine de Turing (et ne sont pas décidables pour l'arrêt), mais certaines restrictions apportées à REZO font que ce langage est décidable.

Les langages ATEF et ROBRA sont basés sur la notion de transduction. La nature de données d'entrée ou de sortie peut varier.

Utilisation de langages algorithmiques pour l'implémentation et la compilation

Le choix d'un (ou de plusieurs) langages d'implémentation pour un LSPL n'est pas facile. En effet, dans le cas de grands systèmes de TA, un LSPL donné n'est qu'un des nombreux composants. L'intégration des différentes phases (écrites avec des LSPL différents), ainsi que la représentation homogène des données n'est pas évidente à gérer. Une certaine efficacité est requise pour le linguiciel. Il n'est, par exemple, pas possible aux linguistes de mettre au point un système si la traduction d'une phrase test (courte en général) prend plusieurs dizaines de minutes voire des heures (comme avec le système Eurotra présenté en 1989 au MTS-II à Munich). Les compilateurs des LSPL

10 Le système accepte et produit des graphes-Q augmentés. L'augmentation a consisté à rajouter des traits booléens en plus des étiquettes.

doivent aussi être efficaces car les linguistes ne supportent pas d'attendre trop longtemps la compilation de leurs grammaires et de leurs dictionnaires.

Les Systèmes-Q [Colmerauer 1970] ont vu plusieurs implémentations différentes (ALGOL, FORTRAN, COMPASS - assembleur Cyber, et PASCAL). Une extension en PASCAL a été réalisée par TAUM-Aviation. L'efficacité de cette dernière version était alors la plus raisonnable.

AL, le langage de desciption d'ATN de BSO ([Vosse & al. 1988]), a été développé en C (Berkeley Unix) et représente environ 75000 lignes de code. Les informations liées au dictionnaire sont écrites en Quintus PROLOG. AL et DLT fonctionnent sur des stations de travail UNIX.

Les langages d'ARIANE (ATEF, ROBRA, TRANSF/EXPANS, SYGMOR, TRACOMPL) ont été développés en assembleur, en PL/360 et en PL/I sur mini-ordinateur IBM sous le système VM/SP. Le projet industriel EUROLANG a permi leur réimplémentation partielle en C++ sur station de travail UNIX. Il ne semble pas que la réécriture se base sur une conception à objets très poussée. L'université de Jiao Tong à Shangaï a également réécrit ATEF en C.

Les travaux de l'ICOT [Akasaka & al. 1989] ont d'abord été réalisés sur machines PSI (machine PROLOG de l'ICOT) puis des versions "ouvertes" (stations UNIX) ont été réalisées. LangLab [Tanaka 1986] est implémenté en PROLOG. Une méthode d'analyse ascendante, BUP, (développée par Matsumoto) a été incluse dans PROLOG afin de supprimer le problème des règles récursives à gauche.

Le système METAL dans son ensemble est implémenté en LISP. Le langage GRADE a eu deux versions en LISP (Kyoto LISP puis ZetaLisp) et une en C, actuellement utilisée.

De nombreux LSPL ou formalismes permettant d'exprimer des analyseurs basés sur les grammaires d'unification sont développés en LISP.

Le choix du langage d'implémentation doit tenir compte de l'efficacité d'exécution mais aussi de l'influence générale sur le système. L'analyse et l'implémentation du LSPL doivent éliminer ou au moins réduire au minimum cette influence. Nous avons vu les problèmes posés par une présence trop envahissante de LISP dans le système METAL.

Compilation vers une machine simulée

La compilation en termes d'une machine abstraite facilite la mise au point du langage. Avec LISP, l'idée des machines simulées est déjà relativement ancienne. WAM (Warren Abstract Machine) est un exemple d'une définition de la sémantique de PROLOG en termes d'une machine abstraite. L'introduction d'un niveau intermédiaire entre le LSPL et la forme compilée permet de décomposer l'analyse liée à la conception.

Le langage REZO [Stewart 1978] est basé sur une spécialisation pour l'analyse syntaxique des ATN. Cette approche permet au langage d'être compilé vers du pseudo-code puis interprété par une machine abstraite.

Le LSPL AL de BSO utilise une ATN Machine (AM) qui convertit un ATN exprimé en AL en une représentation interne. Cette machine facilite la mise au point d'analyseurs définis en termes de piles contenant des pistes (trails). Une piste est l'ensemble des étapes par lesquelles est passé l'ATN pour arriver à un certain état. Cette idée se rapproche de la notion d'arbre de contrôle dans les LSPL d'ARIANE.

Code intermédiaire interprété

Les données linguistiques d'ARIANE-G5 sont compilées sous une forme intermédiaire qui est elle-même interprétée par les automates définis par les différents LSPL du système.

L'utilisation du code intermédiaire est un bon compromis entre compilation et interprétation. En effet, les programmes externes sont compilés vers le code intermédiaire ce qui permet des vérifications statiques et du contrôle d'erreurs. L'interprétation offre une bonne portabilité.

1.2. Analyse du problème : quelles méthodologies de développement ?

L'histoire du TALN a donc bien mis en évidence la nécessité d'avoir des langages spécialisés. Cependant, bien que particuliers, les LSPL nécessitent comme tout langage de programmation un temps et des efforts de développement considérables. De plus, des LSPL mal conçus peuvent poser davantage de problèmes que des langages algorithmiques généraux.

Cela nous amène à nous poser deux questions de fond à propos des LSPL.

- Quels sont les objectifs à atteindre en termes de Génie Logiciel pour les LSPL ?
- Quels doivent être les principes à observer pour une bonne approche du problème de la conception de LSPL ?

Ces deux questions méritent d'être explicitées. La première question cherche à définir les buts que doivent atteindre des LSPL afin d'être satisfaisants. La seconde cherche à cerner quelles propriétés les LSPL doivent vérifier afin d'avoir des chances d'atteindre les buts fixés comme réponses à la première question.

Ces considérations, bien qu'appliquées aux LSPL, débordent largement sur les systèmes de TALN et de TA dans leur ensemble.

1.2.0. Objectifs à atteindre

Si on considère les efforts nécessaires au développement d'un système de TALN (ou de TA), il est impensable de "tout mettre par terre et de recommencer" régulièrement. Ces systèmes doivent être durables.

Durabilité

Le temps de développement de la première version du générateur de systèmes de traduction, ARIANE-G5, a été de l'ordre de 25 homme/années. Au total, la version actuelle, en a demandé près de 40. Il s'agit "seulement" des SLPL et de l'environnement de développement pour linguistes. À partir de cette base, le développement d'un système *opérationnel* de TA (avec tous ses dictionnaires, ses grammaires, les jeux de tests, les corpus, les représentations linguistiques, …) a un ordre de grandeur dix fois plus important.

Les LSPL doivent donc participer à l'augmentation de la durée de vie des systèmes. La réingénierie de vieux systèmes doit pouvoir se faire de manière incrémentale [Jacobson 1991]. Idéalement, on doit pouvoir faire évoluer un système en permanence, en mettant à jour régulièrement les composants qui ne sont plus adpatés à la situation. Dans le cadre de la recherche, une démarche exploratoire doit être possible à partir de l'existant.

Cette approche n'empêche pas de figer le système (sur le papier) pour aller vers une industrialisation [Bachut 1991].

Une question cruciale des systèmes de TA (et de TALN) est de savoir comment faire correspondre les architectures (aussi bien logicielle que linguistique) de TA au type d'utilisation [Boitet 1993a]. Cette question est liée à la durabilité des systèmes et amène à définir deux sous-objectifs proches (quoique distincts) : l'évolutivité et la flexibilité.

Évolutivité

Les LSPL doivent pouvoir changer afin de répondre à de nouveaux besoins des linguistes. Si pour chaque problème spécifique, il fallait définir un nouveau langage, les développements seraient rapidement trop lourds pour que l'ensemble du système soit viable. Une telle situation pourrait être considérée comme un échec du Génie Logiciel.

Flexibilité

Les langages spécialisés doivent pouvoir être utilisés pour réaliser des tâches diverses. En effet, une trop grande spécialisation n'offre pas de durabilité au long terme. Il suffit que les besoins changent ou que de nouveaux formalismes linguistiques apparaissent pour qu'un LSPL ne soit plus ressenti comme utile[11].

La contradiction entre spécialisation et flexibilité n'est qu'apparente. On peut en effet définir des langages de programmation qui peuvent être utilisés à la fois par des linguistes et des informaticiens. De plus, si la structure d'un LSPL n'est pas trop figée, il peut être adapté à d'autres formalismes pas trop éloignés.

Contraintes

Quelles sont les différentes contraintes liées à ces objectifs ? Vouloir définir des langages spécialisés flexibles n'est-il pas une gageure ? Un certain nombre de questions restent sans réponse quant aux spécificités des systèmes de TALN [Boitet 1991][12] (voir aussi [Vauquois 1988] et [Boitet 1993c]).

Ces buts peuvent être atteints par la recherche de trois bonnes propriétés durant l'analyse et le développement de LSPL. Il s'agit de l'intégrabilité, de la généricité et de l'extensibilité.

1.2.1. Intégrabilité

L'intégrabilité d'un système est sa faculté à pouvoir accepter un nouveau composant. Un composant est d'autant plus intégrable qu'il est facile de le rajouter comme nouvel élément d'un système.

Cette définition est à double sens car un composant peut être vu comme un système (s'il est composé de multiples sous-composants) et un système peut être vu comme un

---

11 Si on prend l'évolution naturelle comme métaphore, un LSPL constitue une espèce et les besoins en traitements linguistiques l'environnement. Si l'environnement change, une espèce trop spécialisée se trouvera alors brusquement inadaptée et disparaîtra rapidement. L'aspect très évolutif de l'environnement (la langue) est une caractéristique spécifique du TALN par rapport aux applications classiques du Génie Logiciel. C'est pourquoi nos LSPL ne se limitent jamais à implémenter un formalisme linguistique particulier lié à une théorie linguistique.

12 Nous penserons en particulier au problème 11 décrit dans cet article : "Comment faire correspondre les architectures de TA aux types d'utilisation ?".

composant d'un sur-système l'englobant. Nous faisons la distinction avec l'*ouverture* qui serait plutôt l'inclusion dans le système d'un ou plusieurs langages de programmation qui permettent des extensions directement par programmation.

Les composants doivent donc être raisonnablement indépendants. De plus, ils doivent pouvoir communiquer. Un "médiateur" semble donc être nécessaire.

Plusieurs travaux illustrent cette idée pour le système ARIANE. Dans [Gerber 1984], l'insertion de phases PROLOG est proposée pour la connection à une base de connaissance. Dans [Durand 1988], l'insertion de l'outil TTEDIT est étudiée. Cet outil permet de "réviser" les arbres (par exemple, de transfert ou de générateurs) durant le développement linguistique, afin de les rendre parallèles à l'arbre d'analyse. Les travaux sur la traduction basée sur le dialogue de [Blanchon 1994] présentent plusieurs exemples d'insertion : phase de désambiguïsation interactive ou de rétrotraduction. Tout récemment, [Assimi 1994] a exposé l'intégration d'un étiqueteur (AGTS) selon deux approches différentes.

Nous verrons dans le deuxième chapitre qu'une réponse architecturale peut être trouvée au problème de l'intégrabilité.

1.2.2. Généricité

La généricité d'un composant (ou d'un système) est d'autant plus grande qu'il est facile de le dériver pour de nombreuses applications spécialisées différentes.

Nous faisons la distinction entre généricité et généralité. Un composant est général s'il peut être utilisé tel quel pour plusieurs applications. La généricité insiste sur l'idée que le composant sert de "souche" à la création de nouveaux composants.

Par exemple, des automates peuvent avoir une multitude d'applications. Certains formalismes, conçus à l'origine pour représenter la connaissance, peuvent être utilisés avec profit dans des cadres purement informatiques.

Une application de la généricité sera abordée dans le troisième chapitre. Cette application de la généricité sera étendue dans la seconde partie.

1.2.3. Extensibilité

L'extensibilité d'un composant (ou d'un système) est d'autant plus grande qu'il est facile d'altérer ou de rajouter des fonctionnalités sans modification des objets déjà existant.

Un LSPL doit disposer de mécanismes qui permettent d'étendre son "comportement". Il doit aussi être facile de modifier un comportement par dérivation. Cette opération consiste à dupliquer un objet et à modifier légèrement le comportement de cette copie. Les techniques de programmation par objet avec héritage sont particulièrement adaptées à cette approche.

Un premier exemple d'extensibilité sera abordé dans le troisième chapitre, avec le langage LT. Les leçons de cette expérience seront généralisées dans la troisième partie.

1.3. LEAF, un modèle d'identification des composants

Pourquoi essayer d'identifier les composants intervenant dans un système de TALN ? Nous avons vu que les LSPL pouvaient être conçus selon des approches très variées et nous pensons qu'il est nécessaire de déterminer précisément comment peut se décomposer un système. Cette analyse nous permet de catégoriser les différentes fonctions des LSPL.

LEAF (Lattice, Engines and Features) est un modèle d'architecture logicielle destinée à apporter des réponses aux questions suivantes [Lafourcade 1993c] :

- Comment garder une trace de l'origine de l'ambiguïté ?

On veut disposer d'un système simple permettant au linguiste informaticien de représenter et de gérer les ambiguïtés, soit explicitement, soit implicitement.

- Quelle structure de données utiliser ?

On souhaite disposer d'une structure générique indépendante d'un formalisme linguistique et utilisable dans plusieurs domaines du TALN, comme en particulier, la reconnaissance de la parole, la reconnaissance de l'écriture et l'interprétation des textes écrits.

- Comment garantir la rapidité des traitements linguistiques ?

Les domaines comme l'interprétation de textes écrits (ou la reconnaissance de l'écriture et la reconnaissance de la parole), intégrés dans le processus de communication homme-machine, ne souffrent pas de délais trop importants. La rapidité décroit quand la richesse de représentation de la structure augmente (contrairement à la qualité du résultat).

- Comment simplifier le travail du linguiste informaticien ?

La visualisation des résultats d'analyse et la nature de l'interaction entre le linguiste et ces structures aura une influence sur la mise au point des applications linguistiques. L'accent est de plus en plus souvent mis dans la visualisation graphique de structures de données très complexes (c'est, par exemple, le cas dans le projet CYC [Lenat, Guha, Pittman & al. 1990]). Dans LEAF, la structure de treillis est facilement visualisable et il semble intéressant de pouvoir exhiber "à la demande" les informations portées par les nœuds et jugées pertinentes par l'utilisateur.

- Comment mettre en œuvre des techniques de génie linguiciel ?

Un environnement de programmation linguistique doit être dynamique, c'est-à-dire que la modification des attributs d'un objet, même au cours d'une analyse, devrait avoir un effet immédiat sur l'état du système (sans recompilation). La conception et la mise au point des linguiciels en seraient grandement facilitées.

Il nous semble également nécessaire de pouvoir centraliser les informations destinées aux développeurs linguistes. Il est plus aisé de faire communiquer plusieurs outils autour d'une seule structure de données partagées (technique du "tableau noir") que de définir des protocoles ad hoc entre chaque catégorie d'outil (morphologique, syntaxique, etc.).

1.3.0. Modèles d'analyse et d'architecture

L'analyse logicielle permet de décrire les objets qui composent une application informatique ainsi que leur comportement. De nombreux types d'analyse ont été proposés. En considérant une approche dérivée de celle de Rumbaugh (dont une

bonne analyse peut être trouvée dans [Hayes & Coleman 1991]), nous définissons les trois modèles d'analyse suivants :

- modèle objet ;
- modèle statique ;
- modèle dynamique.

Ces modèles s'attachent à décrire des aspects différents (on parlera plutôt de facettes) ; ils sont donc complémentaires.

Le modèle objet rend compte de la structure des objets mis en œuvre dans l'application et de leurs relations (telles que les compositions, les hiérarchies, les échanges de données). Le modèle dynamique décrit le comportement temporel des objets. Le modèle statique décrit comment la communication entre objets est réalisée.

Dans sa décomposition originale, Rumbaugh ne considère pas le modèle statique, mais introduit un modèle fonctionnel avec lequel il essaye de mettre en évidence le comportement d'un système dans son ensemble. Nous avons supprimé le modèle fonctionnel, car il introduit une distinction entre les composants et l'application dans son ensemble. Nous proposons de le remplacer en permettant que le modèle dynamique puisse être appliqué récursivement à tous les niveaux d'objets.

Ces modèles tiennent un rôle dans le processus d'analyse, dans la mesure où ils constituent une interprétation abstraite sur la nature et le comportement des objets prévus dans l'application. On peut dériver de ces modèles d'analyse des modèles d'architecture qui permettront de structurer la conception et la réalisation d'un système informatique. Il ne s'agit alors plus d'interprétation, mais de "schéma", à partir duquel une application informatique pourra être construite.

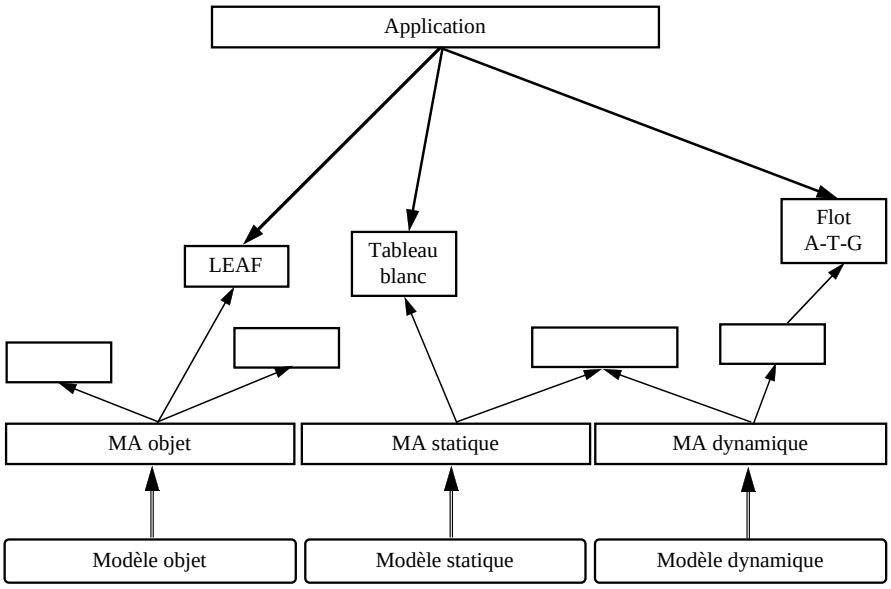


Figure 1.4 : Modèles d'analyse et modèles d'architecture.

Les trois modèles d'analyse permettent de décrire les objets d'une application selon trois facettes : leur composants, les modes de communication et les séquences de communication entre les composants. Des modèles d'architecture sont issus des modèles d'analyse et permettent de concevoir des applications selon des structures préétablies. Les modèles d'architecture (MA) sont plus ou moins spécifiés et peuvent étendre leur description sur plus d'une facette. Les applications seront alors construites selon des

architectures qui instancient les trois facettes et qui peuvent être vues comme des clients de modèles d'architecture. Le tableau blanc (voir chapitre 2) est présenté comme exemple d'architecture statique. Le flot Analyse-Transfert-Génération (ATG) est un exemple d'architecture dynamique utilisée en TA.

Les modèles d'architecture (MA) spécifient plus ou moins précisément quelle approche doit être adoptée pour chaque facette. Un modèle peut s'étendre sur plusieurs facettes. Par exemple, on peut imaginer une application de TA qui définit sa facette objet selon le modèle LEAF, sa facette statique selon le modèle de Tableau Blanc, et sa facette dynamique selon un flot Analyse-Tranfert-Génération (voir figure 1.4). Selon les niveaux d'analyse, une application peut éventuellement être vue selon plusieurs modèles d'architecture différents. Il n'est pas nécessaire que tous les objets d'une application se conforment au même modèle.

Un modèle d'architecture comme PAC ([Coutaz 1988]) est à la fois un modèle objet (avec les composants Présentation, Abstraction et Contrôle) et un modèle dynamique. Par contre, rien de spécifique n'est dit sur la facette statique. Plusieurs modèles d'architecture sont présentés dans [Coutaz 1988].

Enfin, le modèle Seeheim prévoit trois composants : Présentation, Contrôle et Dialogue pour la facette objet ; le modèle entrée/sortie s'intéresse plutôt aux facettes statique et dynamique ; le modèle Multiagents porte sur les trois facettes.

Le modèle objet d'architecture LEAF met en jeu trois types d'objets : un treillis, des décorations, des moteurs. Ce modèle décrit les composantes et les propriétés générales de ces trois objets, ainsi que leurs interrelations.

1.3.1. Treillis

Le type de treillis envisagé peut être comparé avec d'autres structures. Il est intéressant de proposer un exemple d'application pour l'interprétation des textes écrits.

Définition

Les arcs liant les nœuds sont explicites et interdisent la création de cycles. Il existe toujours un unique premier nœud (α), $\circledvdash$, et un unique dernier nœud (ω), $\circledashv$. Chaque nœud porte des décorations, un intervalle de temps et une valuation et les arcs portent un poids.

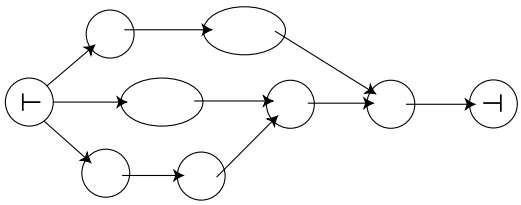


Figure 1.5 : Exemple de treillis.

Les différents chemins correspondent à différentes interprétations. Il n'y a ni nœud "pendant" ni cycle.

On peut représenter un treillis dans un espace à trois dimensions :

- la première dimension représente le temps (écoulement du texte).

Appelons T(n) la projection du nœud n sur l'axe des temps. Pour tout arc du treillis, on a $T(n_d) < T(n_f)$ ou n_d est le nœud en début d'arc et n_f est le nœud en fin d'arc.

- la seconde dimension correspond au niveau d'ambiguïté.
- la troisième dimension (strate) correspond au niveau d'analyse.

Les nœuds de ce treillis peuvent être regroupés en niveaux (on parlera de *strates*) qui correspondent aux niveaux d'analyse. Les nœuds d'une strate peuvent être de nature différente, mais il est intéressant pour une architecture particulière de restreindre une strate à un seul type de nœuds.

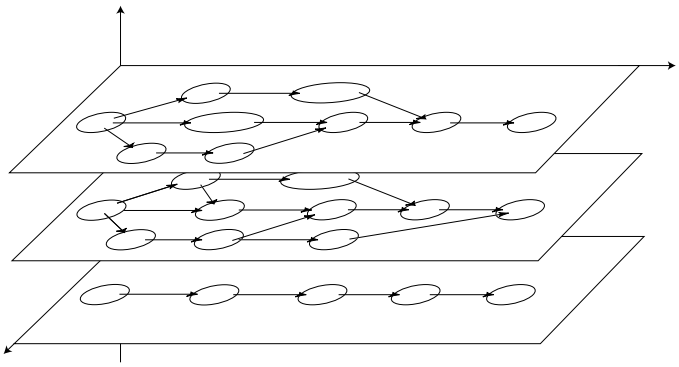


Figure 1.6 : Structure stratifiée.

Chaque strate correspond à un type de traitement. Les treillis sont alignés dans le temps et sont potentiellement ambigus — plusieurs chemins sont possibles entre les nœuds α et ω. La correspondance entre les niveaux peut se faire par alignement temporel ou par référence.

Autres types de structures

On peut comparer les treillis temporels avec deux autres types de structures couramment utilisées en TALN : les grilles et deux types de quasi-treillis, les cartes et les graphes-Q.

Les *grilles* n'ont pas d'arc, mais les nœuds correspondent à un intervalle temporel.

Un nœud N, défini sur un intervalle [t1, t2], est connecté à un nœud N', défini sur [t'1, t'2] si et seulement si t1 t'1 et t2 < t'2.

Les cycles sont impossibles (car t2 < t'2).

Les grilles ont souvent été utilisées en TALN. Par exemple, le reconnaisseur phonétique KÉAL [Quinton 1980] produisait une grille de mots. Certains programmes de reconnaissance de la parole utilisés à ATR produisent aussi des grilles de phonèmes. Habituellement, les nœuds portent les informations suivantes : un intervalle de temps, une étiquette et une valuation. La grille est aussi une structure de données adaptée à la sortie d'un reconnaisseur optique de caractères.

Les treillis définissent explicitement les arcs (et les nœuds). Il est intéressant de représenter les arcs afin de modéliser des séquences associées à des probabilités (via l'utilisation des valuations). Les grilles ne permettent pas ce type de représentation.

Les structures de *cartes* ont été introduites par M. Kay vers 1967 et utilisées dans le système MIND [Kay 1973]. Les nœuds sont alignés et il existe toujours un chemin entre deux nœuds donnés. Les arcs portent des étiquettes et une valuation. Les nœuds ne portent rien. Les cartes sont abondamment utilisées dans les analyseurs basés sur des grammaires hors-contexte.

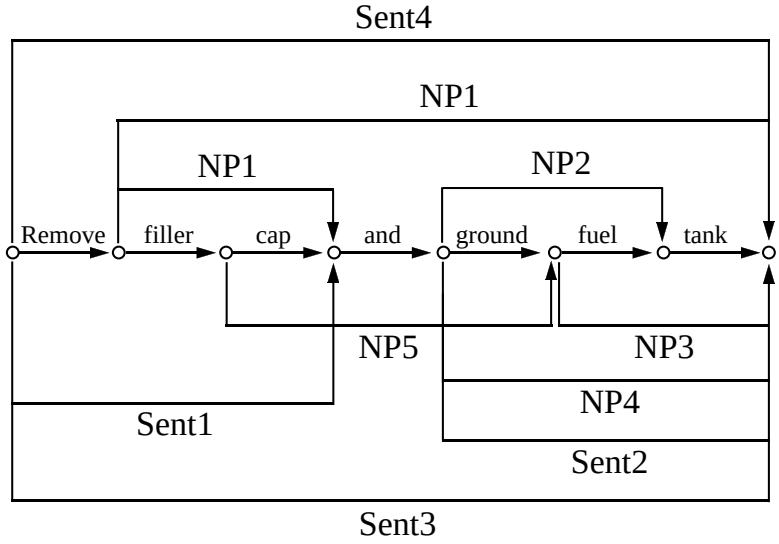
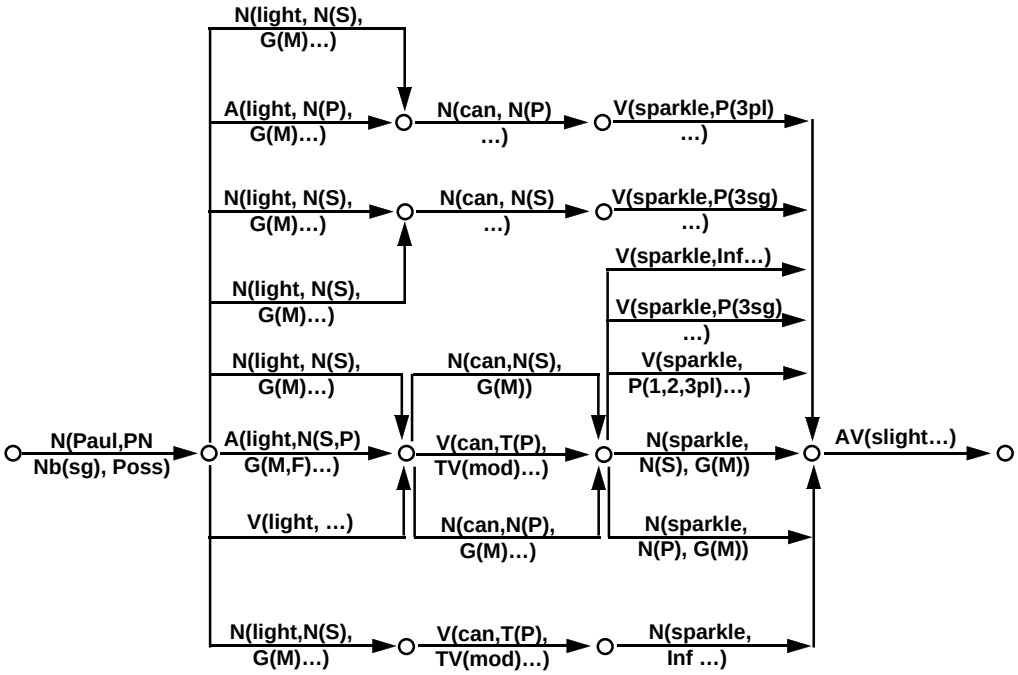


Figure 1.7 : Exemple de cartes.

Cette structure de cartes (tiré de [Boitet 1988]) est construite à partir d'une phrase syntaxiquement ambiguë (Remove filler cap and ground fuel tank.).

Le graphes-Q (figure 1.7) est la structure de données sur laquelle se base le système MÉTÉO. Contrairement aux cartes, il est possible d'introduire de nouveaux nœuds dans le graphe. Les différents chemins peuvent donc représentrer des alternatives complètes.



Paul's light can(s) sparkle(s) slightly.

Figure 1.8 : Un graphe-Q pour une phrase phonétiquement ambiguë.
(exemple tiré de [Boitet 1988])

Exemples

L'utilisation d'une structure multistrate a été proposée par [Mizogushi & Kondo 1982] pour la construction d'un environnement de compréhension du langage naturel, MLSE (Multi-Layered Software Environment). Les structures de données envisagées sont différentes selon les niveaux.

Le prototype KASUGA, décrit dans [Boitet & al. 1994], a été construit à ATR[13] dans le contexte de la traduction de la parole. Ce système utilise un treillis composé de trois strates : les phonèmes, les mots et les groupes syntagmatiques.

Dans la première strate, le treillis de phonèmes est représenté avec les phonèmes dans les nœuds. Ces phonèmes sont issus d'une structure de grilles à trois niveaux produite par un outil de reconnaissance de la parole (SP.REC). Les phonèmes sont assemblés en mots et en groupes syntagmatiques.

Dans [Lafourcade 1993c], nous avons défini une classe de treillis qui peut être utilisée pour l'interprétation des textes écrits. La strate de base est celle contenant les caractères. Les strates morphologiques se décomposent en mots et syntagmes, en catégories syntagmatiques, et enfin en une strate d'analyse morphologique complète. Les strates linguistiques contiennent des arbres concrets, des arbres abstraits et des structures de traits (figure 1.9).

13 ATR - Interpreting Telecommunications Research Laboratories — 2-2 Hikaridai Seika-cho Soraku-gun — Kyoto 619-02 Japan.

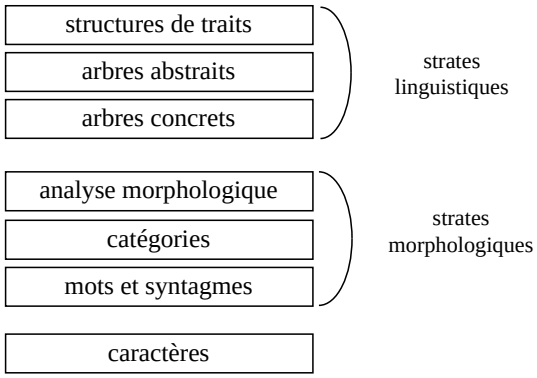


Figure 1.9 : Exemple de strates pour l'interprétation des textes écrits.

Si le texte est la sortie d'un outil de reconnaissance de caractères (OCR), il peut déjà présenter des ambiguïtés. Par exemple, dans le mot "dog", le "o" pourrait être reconnu comme un "a" et le "g" comme un "q" (figure 1.10 a et b).

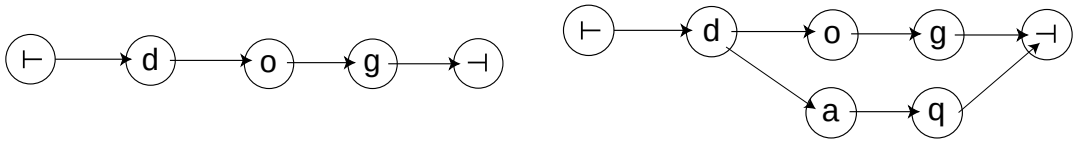


Figure 1.10 : Strate de caractères potentiellement ambiguë.

Dans le cas d'un système de reconnaissance de l'écriture manuscrite, l'entrée pourrait être encore plus ambiguë. La segmentation des lettres de dog poserait alors quelques problèmes (figure 1.11)[14].

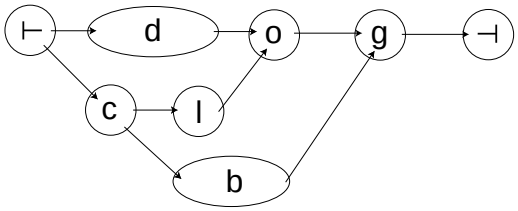


Figure 1.11 : Ambiguïtés liées à l'écriture manuscrite.

On constate que la couverture temporelle de chaque nœud n'est pas constante. Le nœud "d" couvre aussi les nœuds "c" et "l".

Il est nécessaire de représenter le texte d'entrée sous forme de "mots", ce qui implique une phase de segmentation. L'idée la plus simple consiste à regrouper les caractères entre les espaces parmi les nœuds du niveau inférieur. Une correction typographique est d'ailleurs possible à l'occasion (suppression des espaces en trop) ou une segmentation plus élaborée (à l'aide de lexique non structuré) afin de gérer les espaces manquants.

14 Merci à Arno Gourdol pour cet exemple.

Dans le cas de langues posant des problèmes de segmentation (comme le thaï), un tel processus, qui consisterait à construire la strate de mots, serait nécessaire.

La *strate de catégories* (morphologiques) permet de "débroussailler" à moindre coût l'analyse morphologique dont les résultats sont contenus dans les nœuds de la *strate d'information morphologique.*

Les nœuds des *strates linguistiques* contiennent chacun la racine d'un sous-arbre "concret" (arbre de dérivation) dont le mot des feuilles est, par définition, un intervalle d'une interprétation morphologique du texte d'entrée. Il est dit "concret" car il est en correspondance directe avec le treillis de caractères. Certains nœuds peuvent être factorisés en cas d'ambiguïtés. Les nœuds portent également des arbres abstraits en relation avec les arbres concrets. Enfin, des structures de traits décorent les nœuds de la strate des arbres concrets et peuvent être "dépliés" pour former une strate autonome [Lafourcade 1993c].

1.3.2. Décorations

Les *décorations* sont des informations portées par les nœuds du treillis. Les valuations qui sont potentiellement portées par les nœuds et les arcs ne sont pas considérées comme faisant partie des décorations.

Le terme de "décoration" prend ici un sens très général, dans la mesure où de tels objets peuvent être exprimés à l'aide de formalismes multiples : représentations arborescentes, listes d'attributs, ou encore structures de traits typés ou non. Un système de décorations pour le TALN peut se définir à l'aide d'un langage de représentation basé sur des paradigmes divers. On peut penser, par exemple, aux approches à objets (classes ou prototypes), fonctionnelles ou logiques.

Dans la second partie, nous définissons ce qu'est un *langage de représentation basé sur les décorations.* Ce type de langage de représentation nous semble bien adapté au TALN mais peut être utilisé pour d'autres types d'application. Nous précisons ce qu'on entend par structure de *décoration* (terme qui, cette fois, sera pris dans une acception précise). La construction d'un tel langage générique est étudiée, dans le seconde partie, avec une spécification et une implémentation du langage DÉCOR.

1.3.3. Moteurs

On appelle *moteur* tout processus effectuant des modifications géométriques (nœuds et arcs) et algébriques (les décorations et les valuations) sur le treillis. Un moteur est en général fondé sur un langage de représentation manipulant des décorations pour exprimer les traitements à effectuer. Par exemple, les moteurs basés sur les grammaires lexicales fonctionnelles (LFG) utilisent des structures de traits.

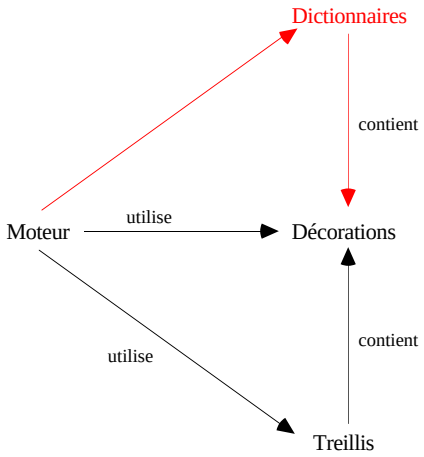


Figure 1.12 : Relations entre Treillis, Moteurs et Décorations

Les moteurs sont, par exemple, des réalisations à partir de LSPL.

La construction de LSPL est étudiée dans la troisième partie avec les languages spécialisés ATEF et ROBRA.

Un moteur est étroitement associé à un LSPL. Pourtant ces derniers constituent une composante indispensable des système de TALN. Dans l'optique adoptée ici, nous ne nous intéresserons pas directement à ces éléments qui sont considérés comme source d'informations pour les application des LSPL.

Chapitre 2

Problèmes d'intégration

It is, in any case, essential in the system that facilities can come and go as new components are added and old ones replaced. […] Linguistic data and linguistic formalisms are sufficiently complex to require the composition of new processes by the experimenter while seated at the console and a number of different languages and notational devices for different aspects of the problem.

M. Kay

L'intégrabilité (présentée au §1.2.1.) consiste à pouvoir ajouter facilement de nouveaux composants à un système. Cette idée s'exprime concrètement avec la notion de "carrossage générique" ou des "coquilles" peuvent accepter certains types d'outils.

Nous avons mené une première expérience avec un outil pour traducteurs occasionnels, ODILE. La recherche d'intégrabilité a pris ici la forme relativement simple d'un protocole auquel devaient se conformer les outils susceptibles d'être intégrés. Cette approche pose un certain nombre de problèmes qui, dans les faits, limitent les outils candidats à l'intégration.

L'analyse du problème dans le cadre de systèmes plus grands et plus complexes permet de dégager une approche architecturale nouvelle. L'approche dite de Tableau Blanc, est proposée comme solution au problème de l'intégration. Cette approche est comparée aux approches alternatives.

2.1. ODILE, une première approche

Le projet ODILE (Outil d'Intégration Extensible de Dictionnaires et de Lemmatiseurs) est un exemple d'intégration limitée pour un outil de TALN. Une description de l'outil sera donnée en introduction à une présentation technique des modalités d'intégration

de deux composants, PILAF[15] et WINTOOL[16]. Ces modalités d'intégration seront ensuite discutées.

La définition et la réalisation originale d'ODILE sont dues à [Tomasino 1990]. Une reconception et une réécriture de l'outil ont ensuite été réalisées par l'auteur [Lafourcade 1991c].

2.1.1. Description

Dans le cadre de la THAM (Traduction Humaine Assistée par la Machine), nous nous sommes intéressés à la création d'outils pour traducteurs occasionnels. Ce genre d'outil, destiné à un public non spécialisé en traduction ou en informatique vise à accélérer et simplifier le processus de traduction. Dans le cadre d'ODILE, il s'agissait d'éviter à l'utilisateur d'avoir à taper une "forme canonique" (par exemple cheval pour chevaux) pour accéder à des informations contenues dans un (ou plusieurs dictionnaires) [Lafourcade 1991b].

Exemple de session

Supposons qu'un utilisateur doive traduire un document. Pour cela, il traduit les paragraphes les uns après les autres directement sur une copie du document à traduire à l'aide de son traitement de texte. Il a aussi lancé ODILE dont la fenêtre apparaît derrière celle du texteur. Supposons qu'il hésite à traduire le fragment «mettent les pieds dans le plat»

Il copie ce fragment dans la zone de saisie d'ODILE et clique sur "Run"[17]. ODILE appelle un lemmatiseur et produit une liste de clés potentielles. L'utilisateur sélectionne la clé désirée et lance la recherche dans le dictionnaire. Si aucune des clés ne convient, il est toujours possible d'en entrer une "à la main".

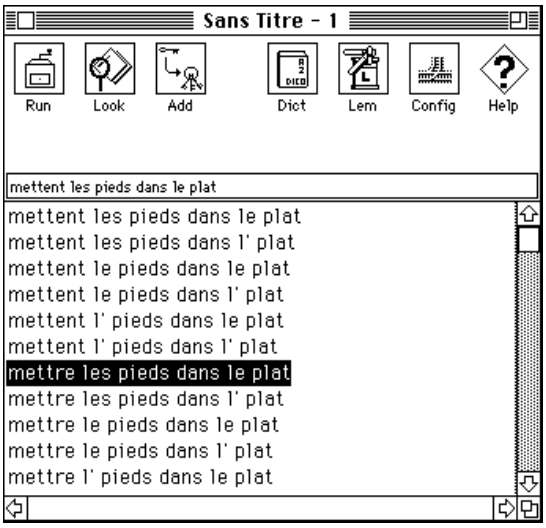


Figure 2.1 : Fenêtre d'ODILE.

Une liste combinatoire des différentes clés générées est proposé à l'utilisateur. Ces clés font l'objet d'une recherche dans l'outil dictionnaire.

---

15 Outil réalisé par l'Équipe TRILAN du laboratoire LGI-IMAG.

16 Outil du commerce, fourni par la société WINSOFT (Grenoble).

17 Ce processus peut-être automatisé.

La fenêtre de l'outil dictionnaire — WINTOOL en ce qui concerne le prototype réalisé — apparaît. Si la clé a été trouvée, l'article correspondant est affiché. Sinon, la fenêtre met en évidence la clé du dictionnaire immédiatement inférieure à la clé cherchée.

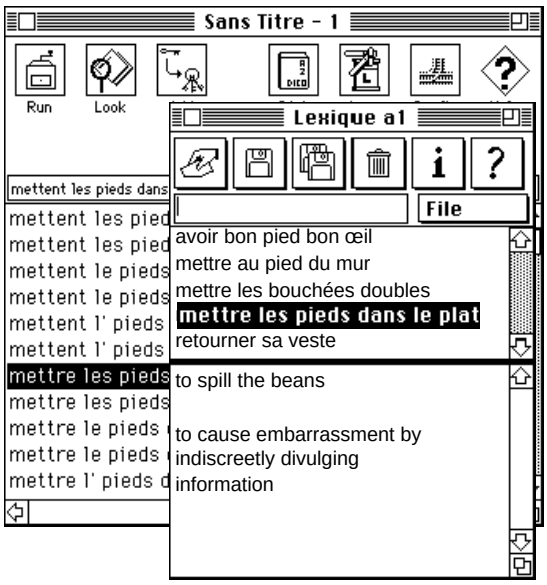


Figure 2.2 : Fenêtre de l'outil dictionnaire WINTOOL.

Si la clé potentielle est présente dans le dictionnaire, elle est affichée et l'utilisateur peut copier (ou modifier) les informations associées.

L'utilisateur peut copier les informations associées à l'article de dictionnaire et revenir au traitement de texte.

Le détail de la séquence de traitements effectués à partir de l'introduction d'un fragment dans la zone d'édition est explicité dans [Lafourcade 1991c].

Architecture

ODILE est donc une "carrosserie" dans laquelle peuvent venir se fixer un lemmatiseur et un outil dictionnaire. L'idée est de pouvoir intégrer et utiliser des lemmatiseurs ou des outils dictionnaires de n'importe quelle origine. Pour cela, nous avons été amenés à définir deux interfaces logicielles (API) spécifiques auxquelles doivent se conformer lemmatiseurs et dictionnaires. Ces API précisent les structures de données d'entrée et de sortie ainsi que les points d'entrée.

Les API (pour leur présentation se référer aux Annexes B) permettent à ODILE de contrôler ces deux outils. Les demandes de lemmatisation et de recherche dans l'outil dictionnaire se traduisent par des appels de fonctions ou de procédures. Par contre, les résultats de lemmatisation sont passés sous forme d'un fichier texte dont le contenu est directement lisible. Ce n'est pas le cas de l'outil dictionnaire, qui dispose de sa propre interface utilisateur et prend la main après qu'une demande de recherche ait été faite par ODILE.

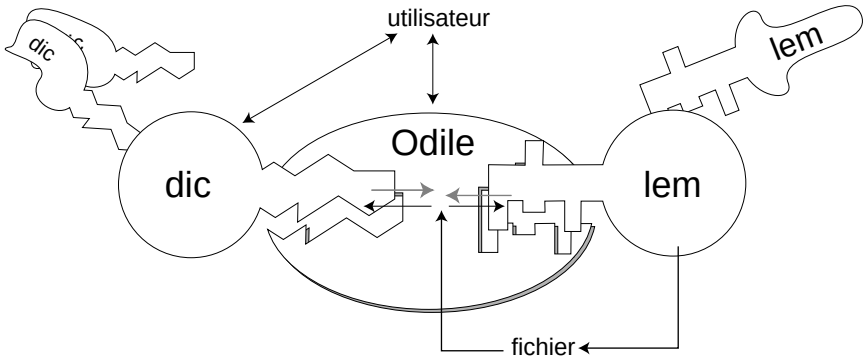


Figure 2.3 : Architecture d'ODILE.

Une interface logicielle (API) permet d'intégrer différents types de lemmatiseurs ou d'outils dictionnaires. Cependant, cette interface n'est pas particulièrement souple, et la communication entre les composants n'est pas normalisée.

Structure de données

La structure de données d'entrée pour les lemmatiseurs est simplement une chaîne de caractères "normalisée" (par deux blancs de suite).

La structure de données de sortie pour les lemmatiseurs est un graphe avant contenant les informations suivantes :

- Les nœuds représentent des formes ou des lemmes.
- Les arcs représentent les passages possibles entre les nœuds.
- Chaque nœud contient jusqu'à dix (10) informations (entre parenthèses sont indiqués le nom de la variable et son type) :

  1) le numéro du nœud (Numéro : entier) ;

  2) son identification, obligatoire (valChaîne : chaîne) et ne contenant pas de blanc ;

  3) son type, obligatoire (TypeChaîne) : Forme ou Lemme (F | L) ;

  4) sa nature, facultative (Nature) : Occurrence complète, Tournure, morceau Gauche, Interne, ou Droit (D | O | I | G | T) ;

  5) sa localisation, obligatoire (Origine : entier), désigne l'origine (0 désignant la position précédant le premier caractère) de ValChaine ;

  6) sa longueur, obligatoire (Longueur : entier), qui est le nombre de caractères couverts par le nœud (la longueur de la sous-chaîne traitée dans la chaîne d'entrée) ;

  7) sa catégorie morphologique (Catégorie : chaîne), facultative (par exemple : (Catégorie 'adv) ou (Catégorie 'subc) ou encore (Catégorie 'dept)) ;

  8) ses variables morphologiques (Variables : liste), facultatives (par exemple : (Variables('plus 'mas 'tre 'suj)) ou (Variables((nombre 'plus) (genre 'mas) (fonction 'suj))) ;

  9) des informations supplémentaires (Reste : liste), facultatives, qui pourraient être utilisées dans l'avenir (par exemple, relief typographique, domaine du document, etc.) ;

  10) les numéros des nœuds successeurs du nœud courant dans le graphe (Suivant : liste d'entiers), la liste vide (()) dénotant l'absence de successeur.

Cette structure est passée à ODILE sous forme d'un fichier texte contenant la suite des nœuds dans l'ordre ascendant des numéros. Chacun des nœuds est codé (à la LISP) comme suit :

```
(((Numéro Numéro) (ValChaîne ValChaîne)
  (TypeChaîne TypeChaîne) [(Nature Nature)]
  (Origine Origine) (Longueur Longueur)
```

```
  [(Catégorie Catégorie)] [(Variables ({Variables}+))]
  [(Reste ({Reste}+))])
(Suivants ({Suivant}+)
 )
```

Par exemple, l'analyse de la chaîne «Les poules couvent»donne le résultat suivant :

```
(((Numéro  1) (ValChaîne 'les) (TypeChaîne 'F) (Nature 'O)
(Origine 1) (Longueur 3)) (Suivants (4 5)))

(((Numéro  2) (ValChaîne 'le) (TypeChaîne 'L) (Nature 'O) (Origine 1)
(Longueur 3)) (Suivants (4 5)))

(((Numéro  3) (ValChaîne 'il) (TypeChaîne 'L) (Nature 'O) (Origine 1)
(Longueur 3)) (Suivants (4 5)))

(((Numéro  4) (ValChaîne 'poules) (TypeChaîne 'F) (Nature 'O)
(Origine 5) (Longueur 6)) (Suivants (6 7 8 9)))

(((Numéro  5) (ValChaîne 'poule) (TypeChaîne 'L) (Nature 'O)
(Origine 5) (Longueur 6)) (Suivants (6 7 8 9)))

(((Numéro  6) (ValChaîne 'couvent) (TypeChaîne 'F) (Nature 'O)
(Origine 11) (Longueur 7)) (Suivants ()))

(((Numéro  7) (ValChaîne 'couvent) (TypeChaîne 'L) (Nature 'O)
(Origine 11) (Longueur 7)) (Suivants ()))

(((Numéro  8) (ValChaîne 'couver) (TypeChaîne 'L) (Nature 'O)
(Origine 11) (Longueur 7)) (Suivants ()))

(((Numéro  9) (ValChaîne 'couver) (TypeChaîne 'L) (Nature 'O)
(Origine 11) (Longueur 7)) (Suivants ()))
```

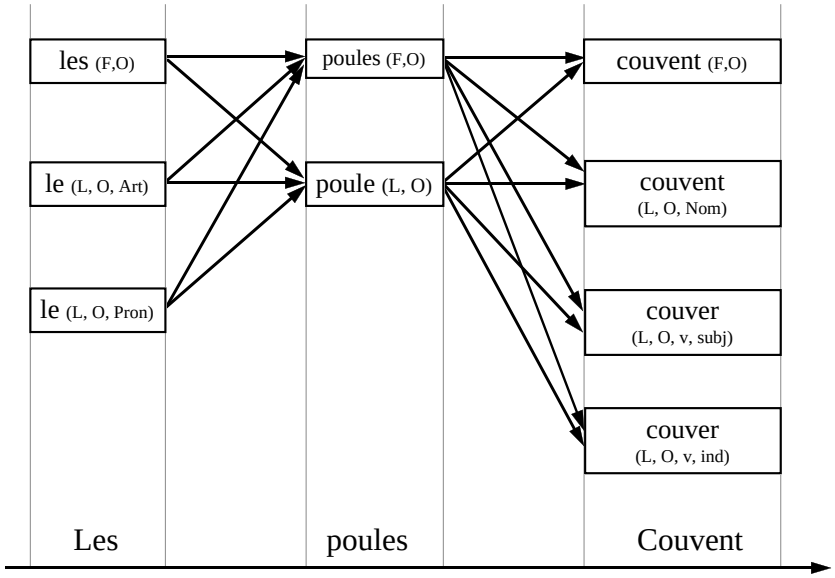


Figure 2.4 : Exemple de graphe de clés potentielles.

Ce graphe est produit par ODILE à partir des résultats fournis par le lemmatiseur.

2.1.2. Interface logicielle figée

La définition d'une API est particulièrement contraignante. Seuls les outils dictionnaires et les lemmatiseurs qui s'y conforment peuvent être intégrés. En pratique, ce facteur est très limitant.

L'intégration d'un nouvel outil dictionnaire ou d'un nouveau lemmatiseur requiert la maîtrise du code de ces outils ou au moins la possibilité d'écrire une sur-couche logicielle.

2.1.3. Architecture implicite

L'architecture proposé dans ODILE impose des contraintes très fortes. Il doit y avoir une bonne conformation entre les types définis par l'API et ceux manipulées par un lemmatiseur intégrable.

La communication entre les composants se fait directement par API. Seul le retour du lemmatiseur est effectué par fichier, ce qui assure une certaine généricité. Aucune architecture générale n'est prévue, mais seulement un format de communication entre des outils spécifiques (dictionnaires et lemmatiseurs).

Il n'est donc pas possible d'intégrer, sans un effort de développement important, des composants fonctionnant en environnements hétérogènes. Ce n'était d'ailleurs pas l'objectif initial du projet ODILE, qui visait seulement à intégrer un outil dictionnaire et un lemmatiseur.

2.2. Analyse du problème : variété et taille des composants que l'on peut souhaiter intégrer

Les environnements spécifiques (pour le TALN, par exemple) sont différents des environnements de Génie Logiciel classiques. Les composants suceptibles d'être intégrés en TALN sont très variés et manipulent des structures de données souvent incompatibles aussi bien informatiquement que du point de vue des représentations linguistiques très liées à des théories particulières. De plus, la taille de ces composants varie beaucoup. Une application informatique réalisée avec un environnement de Génie Logiciel est rapidement figée, ce qui n'est pas le cas pour les applications linguistiques. Les dictionnaires et les grammaires ne sont jamais complètement terminés, et il est nécessaire dans l'application linguistique de fournir au linguiste des fonctionnalités pour la mise à jour de ses données[18].

Certaines approches sont déjà génériques mais se limitent à des catégories particulières de formalismes. Par exemple, [Bouchard, Emirkanian, Estival & al. 1992a] proposent un environnement pour l'intégration de grammaires basées sur l'unification. Le système GDE (Grammar Development Environment) [Caroll, Briscoe & Grover 1991] définit et utilise un formalisme grammatical spécifique. L'approche adoptée dans PLNLP (Programming Language for Natural Language Processing) par [Jensen & al. 1993] est également basée sur un seul formalisme, APSG (Augmented Phrase Structure Grammar), à partir duquel il est possible de créer un système complet.

Dans [Kay 1973], l'idée de *composabilité* est mise en évidence. Selon lui, pouvoir ajouter de nouveaux composants (et supprimer les anciens) permet de faciliter le travail des linguistes. Cette intégration doit être indépendante des différentes théories ou écoles linguistiques. De plus, elle va de pair avec la notion de LSPL. L'expérience du système MIND a été un premier pas dans cette direction.

Dans le chapitre précédent, nous avons montré qu'il était souhaitable de définir un modèle qui identifie les types de composants nécessaires dans un sysème de TALN. Il

18 D'autres types de systèmes ne sont jamais figés, par exemple les systèmes experts.

faut aussi définir une approche générale pour la communication entre ces composants. C'est ce que nous entendons par architecture. On constate que, dans la plupart des systèmes de TALN, les deux concepts ne sont pas explicitement séparés. Nous pensons que c'est seulement par la définition d'une bonne architecture que l'on peut résoudre de manière générale le problème de l'intégration des composants dans un système.

L'idée que nous retenons consiste à rechercher une (ou plusieurs) techniques simples pour intégrer des composants de nature différentes (comme dans l'expérience ODILE) en environnement distribué et hétérogène. L'étude de systèmes de TA[19] de grande taille permet de faire un tour d'horizon des différentes approches adoptées selon deux critères :

- la facilité d'intégration d'un composant au système,
- la communication entre composants.

2.2.1. ARIANE

ARIANE-G5 est un générateur de systèmes de TAO de seconde génération. L'approche adoptée favorise le transfert sans pour autant le rendre obligatoire. Il est en effet possible de mettre au point des systèmes de TAO par pivot.

Cinq LSPL sont utilisés dans ce système. Pour une description complète d'ARIANE, le lecteur peut se référer à [Boitet 1982], [Boitet, Guillaume & Quézel-Ambrunaz 1982] et [Boitet & al. 1985]

Architecture

Les processus mis en jeu dans ARIANE sont organisés séquentiellement. Les trois étapes principales sont l'analyse, le transfert et la génération. Un certain nombre de phases composent chaque étape. Chaque phase est écrite avec un LSPL adapté.

En analyse, on trouve :

• AM	Analyse Morphologique	obligatoire	écrite en ATEF
• AX	Analyse Expansive	facultative	écrite en EXPANS
• AY	Analyse Expansive	facultative	écrite en EXPANS
• AS	Analyse Structurale	obligatoire	écrite en ROBRA

En transfert, on trouve :

• TL	Transfert Lexical	obligatoire	écrite en EXPANS
• TX	Transfert Expansif	facultative	écrite en EXPANS
• TS	Transfert Structural	obligatoire	écrite en ROBRA
• TY	Transfert Expansif	facultative	écrite en EXPANS

En génération, on trouve :

• GX	Génération Expansive	facultative	écrite en EXPANS
• GS	Génération Syntaxique	obligatoire	écrite en ROBRA
• GY	Génération Expansive	facultative	écrite en EXPANS
• GM	Génération Morphologique	obligatoire	écrite en SYGMOR

Cet ensemble de phases est saturé, il n'est pas possible d'en rajouter de nouvelles à partir du moniteur offert aux linguistes. L'ordre de ces phases est imposé.

---

19 Les sources d'informations ne manquent pas [Balkan 1992], [Hutchins 1986] et [Hutchins & al. 1992]

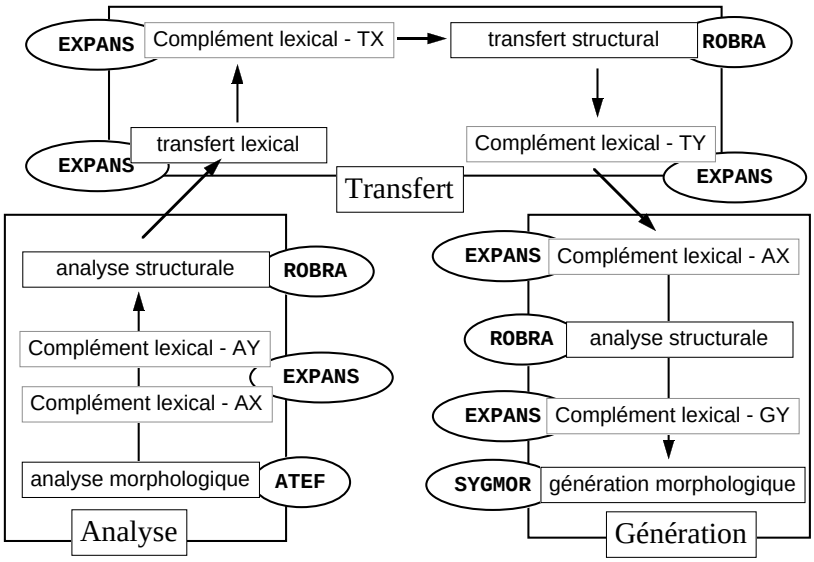


Figure 2.5 : Organisation générale du processus de traduction en ARIANE-G5

Les articulations, qui peuvent être vues comme des changements de coordonnées linguistiques entre les phases, doivent être décrites avec le LSPL TRACOMPL. Ces articulations sont optionnelles.

Les articulations possibles sont :

- AMAX, AMAY, AMAS, AXAY, AXAS ;
- TLTX, TLTS, TXTS, TSTY, TSGX, TSGS, TYGX, TYGS ;
- GXGS, GSGY, GSGM, GYGM.

Les phases d'articulation ne sont nécessaires que pour composer des phases provenant de linguiciels utilisant des jeux de décorations différents.

On peut remarquer que l'ajout dans le moniteur de phases supplémentaires augmenterait la combinatoire des articulations. De plus, les phases d'expansion consistent principalement en de la consultation de dictionnaires, et ne sont pas équivalentes aux phases obligatoires. Il serait peut-être souhaitable de pouvoir enchaîner deux phases d'analyse structurale. Ceci n'est pas possible[20] directement à partir du moniteur.

Structures de données

Les structures de représentation doivent être des arbres décorés. Seule l'entrée de l'analyse morphologique et la sortie de la génération morphologique sont des textes (chaîne de caractères).

Les structures de données échangées entre les phases sont donc parfaitement homogènes, à la représentation linguistique près. Ce problème est réglé lors des phases d'articulation.

---

20 Avec le LSPL ROBRA, il est possible cependant d'enchaîner jusqu'à sept systèmes transformationnels mais ils doivent partager le même jeu de décorations.

2.2.2. METAL[21]

METAL est conçu pour traduire de la documentation technique comportant de nombreux hors-textes (schémas, tables, …). Les données à traduire doivent d'abord être extraites des documents par une phase de déformatage impliquant l'identification et le marquage des "unités de traduction". Les textes traduits seront réinsérés en fin de traitement. Le système METAL a donc été intégré à une chaîne complète d'édition de documents[22] (cf figure 2.6).

Architecture

L'enchaînement des processus s'effectue selon un flot continu (avec une alternative en fin de chaîne de traitement).

Il y a une distinction entre l'ensemble des phases du système (de l'aquisition du texte à la production du texte de sortie) et la phase de traduction (analyse, transfert, génération)[23]. Les phases de transfert et de génération ne sont pas clairement séparées (on peut comparer cette situation à ARIANE où les trois phases sont clairement indépendantes du point de vue logiciel). En effet, certaines des opérations réalisées durant le transfert relèvent habituellement de la génération. En fin de compte, METAL est plutôt considéré comme un système de génération "1.5" que comme un système de seconde génération.

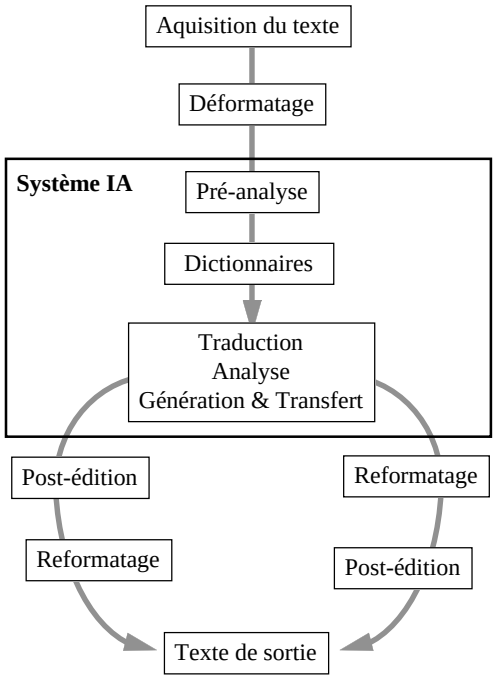


Figure 2.6 : Enchaînement des processus dans METAL.

L'acquisition des textes peut se faire par OCR.

---

21 L'historique du système METAL est présenté dans le §1.1.2.

22 Le projet EUROLANG a adopté la même approche.

23 Une phase d'Intégration existe entre l'analyse et le transfert afin de résoudre les anaphores intra et inter-phrase.

L'ensemble des processus se présente sous forme de fonctions LISP. Au niveau de l'implémentation, cela est considéré comme un progrès par rapport aux versions antérieures, écrites avec des langages de programmation de plus bas niveau. Le système est donc plus accessible aux linguistes. Comme les fonctions LISP sont contrôlées par les données linguistiques, il n'y a plus de séparation stricte entre les données algorithmiques et les données linguistiques [Hutchins 1986].

L'enchaînement des phases consiste en l'appel successif de fonctions LISP par un "shell". Par exemple, l'analyse est appelée par la fonction PARSE. L'ensemble du système consiste donc en un programme LISP. Cette situation rend obligatoire un développement informatique lourd pour tout ajout de composants dans l'enchaînement des phases. Aucun mécanisme permettant au linguiste d'être autonome n'a été prévu à cet effet.

2.2.3. KBMT

Le projet KBMT-89 a démarré en mai 1988 à l'Université Carnégie Mellon (CMU) et a abouti à un premier prototype en février 1989 [Goodman & Nirenburg 1991]. L'approche adoptée pour ce système de TA est celle de l'interlingua, avec "ontologie" (base de connaissances) autonome. Comme son nom l'indique, KBMT (Knowledge Based Machine Translation) est basé sur la connaissance. Les connaissances sont gérées par un système approprié (ONTOS). Le corpus choisi était le manuel du PC américain et du PC 5550 japonais. Les langues visées étaient le japonais et l'anglais — avec traduction dans les deux sens.

L'ontologie liée au domaine comportait environ 1500 concepts. ONTOS est basé sur un gestionnaire de frames, FRAMEKIT. Le langage utilisé pour représenter les modèles du domaine est une extension de FRAMEKIT. Le formalisme utilisé pour l'écriture des grammaires est une variante de celui des grammaires lexicales fonctionelles (LFG).

L'ensemble du système est implémenté en LISP.

Architecture

L'analyse produit une structure multiple interlingue qui est désambiguïsée par l'augmenteur. Cette phase d'augmentation est à la fois automatique et interactive. Une structure unique interlingue sert de base à la génération.

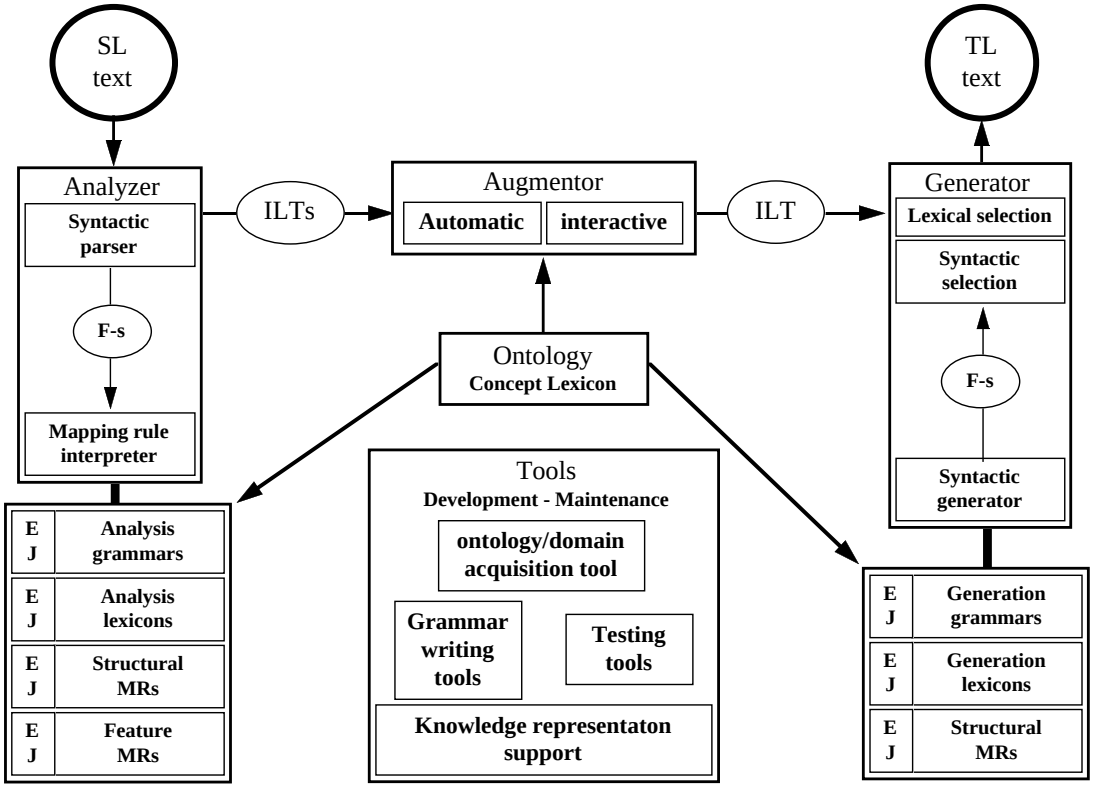


Figure 2.7 : Architecture globale de KBMT-89.

L'analyse du texte en langue source et la génération en langue cible utilisent un ensemble de bases de connaissances qui incluent des grammaires, des dictionnaires et des règles de correspondance. L'analyse produit des représentations interlingues (ILT) potentiellement ambiguës. L'augmenteur lève ces ambiguïtés. La représentation ILT unique est passée en génération.

Des outils permettent de construire le dictionnaire de concepts (l'ontologie). Ce dernier est utilisé par l'augmenteur.

L'organisation générale du système est très proche de celle d'ARIANE, si ce n'est que la phase de transfert est logiquement remplacée par la phase d'augmentation. Il s'agit également d'une architecture en flot, l'ordre des étapes ne peut pas être modifiée.

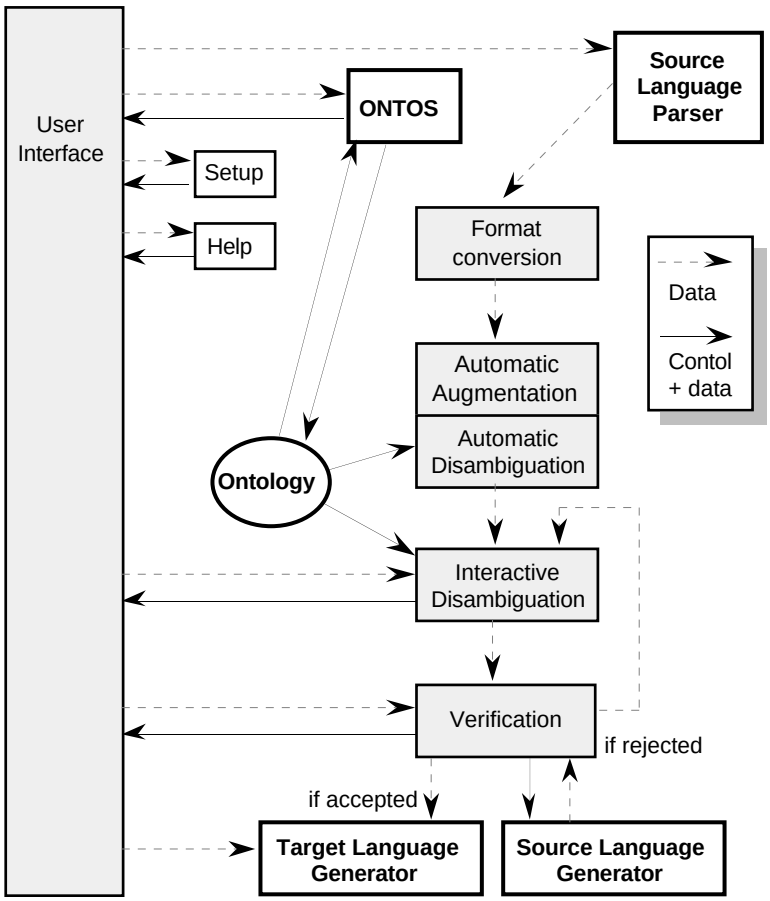


Figure 2.8 : Architecture de l'augmenteur de KBMT-89.

L'interaction n'est pas centralisée. Comme dans ODILE, chaque constituant a un type de communication spécifiquement lié à sa fonction.

Structures de données

La représentation de la connaissance de KBMT-89 s'exprime selon un langage de représentation de la connaissance basé sur les frames, FRAMEKIT [Goodman & al. 1991]. L'interlingue introduit des contraintes sémantiques et des frames disposant de marques.

Prenons comme exemple la phrase suivante : «Confirm that the power unit switches of the system unit and the printer are in the 'off' position when the connection of each device is complete[24], traduite du japonais vers l'anglais. Ce qui suit est un exemple (partiel mais représentatif) d'ILT correspondant à cette phrase qui sert d'entrée à l'augmenteur :

```
(make-frame clause1
  (ilt-type (value clause))
  (clauseid (value clause1))
  (propositionid (value proposition1))
  (discourse-cohesion-marker (value (conditional clause2)))
  (speechactid (value speech-act1)))

(make-frame proposition1
  (ilt-type (value proposition))
```

24 En japonais "Kaku sochi no setsuzoku go shynryo shitara shisutemu yunitto to purinta no dengen suittchi ga 'kiru' gawa no natte iru koto o kakunin shite kudasai".

```
(clauseid (value clause1))
(propositionid (value proposition1))
(aspect (value aspect1))
(complete (value yes))
(is-token-of (value *connect))
(agent (value unkown))
(them (value role2))
(time (value time1)) …
```

Les structures de frame sont présentées dans le chapitre 4.

Le système est basé sur la théorie LFG et représente de façon similaire les structures fonctionnelles et les structures de consituants.

2.2.4. LIDIA

Le projet LIDIA ([Blanchon 1990a], [Blanchon 1990b] et [Blanchon 1994]) se situe dans le cadre de la TA pour l'auteur. Un auteur est amené à désambiguïser son document en vue d'une traduction multilingue vers des langues qu'il peut ignorer. Des processus permettant de repérer les ambiguïtés et d'engager un dialogue entre le système et l'utilisateur ont été réalisés.

Architecture

L'architecture de la maquette LIDIA-1 est distribuée (Figure 2.9). Sur la station de rédaction, l'auteur rédige des cartes HyperCard qui vont être l'objet de processus d'analyse, de désambiguisation et de traduction (transfert et génération). Ce noyau HyperCard est un client du *serveur LIDIA* qui encapsule un *superviseur LIDIA* et un *serveur de désambiguïsation.*

Le superviseur LIDIA effectue l'ordonnancement des traitements pour les différents objets à traiter. Le serveur de désambiguïsation produit des arbres de questions à partir des résultats d'analyse, et le superviseur LIDIA engage le dialogue de désambiguïsation à partir de cet arbre de questions. Le dialogue consiste en une série de questions qui permettent à l'utilisateur de choisir la bonne interprétation parmi celles proposées.

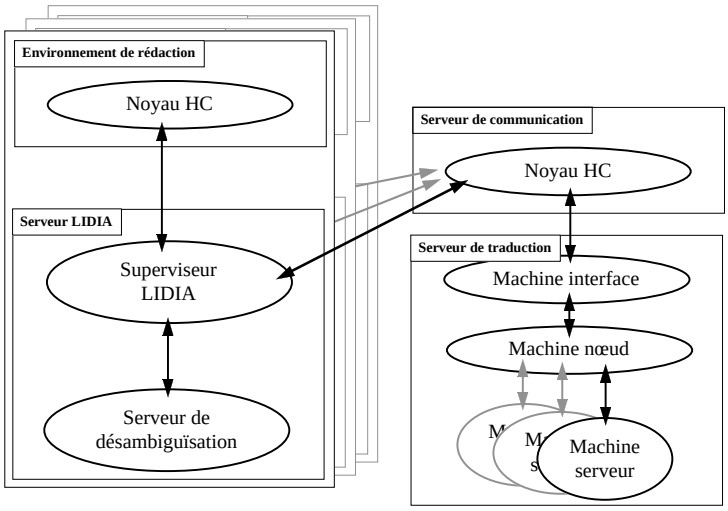


Figure 2.9 : Architecture LIDIA-1.

Les noyaux HC (HyperCard™) sont utilisés comme interface utilisateur pour le serveur de rédaction mais aussi pour le serveur de communication.

Les linguiciels qui effectuent les différentes étapes de la traduction (analyse, transfert et génération) sont des machines du serveur de traduction. Les machines serveurs sont des applications ARIANE.

Un serveur de communication réalise l'interface entre le serveur LIDIA et le serveur de traduction. Il gère deux files de tâches d'entrées et de sorties. Ces tâches se présentent sous la forme de fichiers requêtes lisibles. Ce dernier aspect est particulièrement important pour la mise au point. Le fichier requête contient une description de l'objet à traiter, l'identification de la langue source et des langues cibles et le texte (de l'objet) à traduire. Le fichier résultat contient de plus le résultat d'analyse.

Structures de données

L'analyseur produit une structure dite multisolution, multiniveau et concrète (mmc). Cette structure est un arbre.

La structure est *multisolution* car, pour chaque phrase, elle contient toutes les analyses vérifiant le modèle syntagmatique, syntaxique et logico-sémantique des grammaires mises en œuvre. Ce choix est nécessaire pour obtenir, par le processus de désambiguïsation, l'analyse qui correspond au choix de l'auteur.

La structure est *multiniveau* car les décorations portées par les nœuds représentent divers niveaux d'interprétation linguistique. Ces niveaux sont ceux des classes syntaxiques et syntagmatiques, des fonctions syntaxiques, et des relations logico-sémantiques.

Les propriétés des mots et des syntagmes sont également multiniveaux, car elles concernent aussi bien la surface (genre, nombre, temps, mode, …) que l'abstraction (sexe, pluralité, moment, modalité, …). Enfin, les valeurs lexicales ont 4 niveaux : forme, lemme, unité lexicale ou famille dérivationnelle, et acception interlingue.

La structure est dite *concrète,* car on y retrouve directement le texte analysé en parcourant le mot des feuilles de l'arbre de gauche à droite. Cette propriété facilite le processus de génération des questions.

Un exemple d'analyse de phrase ambiguë est donné dans la figure 2.10.

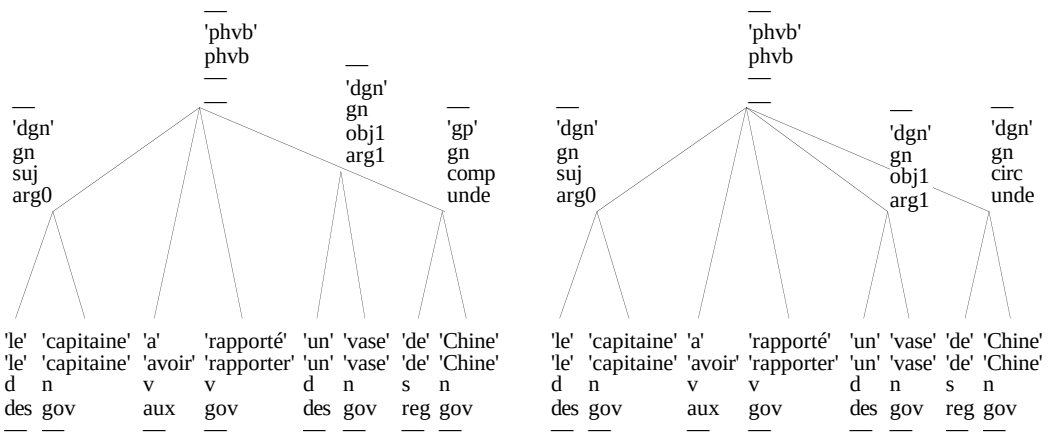


Figure 2.10 : Exemple de structure mmc.

Le capitaine a-t-il rapporté un vase depuis la Chine ou une porcelaine de Chine ?

Traitements linguistiques

L'organisation des traitements linguistiques est illustrée par la figure 2.11.

Après une étape de normalisation (correction orthographique, normalisation terminologique, etc.), le texte est analysé sur le serveur de traduction. L'analyse mmc produite sert de support à la désambiguïsation interactive (arbre de questions et dialogues) qui permet d'obtenir une structure unisolution multiniveau concrète (umc).

Une réduction (appelée aussi abstraction) de l'umc produit la structure sur laquelle est effectuée le processus de transfert. Les structures uma (unisolution multiniveau abstraite) et umc produites respectivement par les générations structurale et syntaxique permettent d'obtenir le texte traduit. Ces structures sont conservées en vue d'une rétrotraduction.

Les structures de données sont toutes échangées par fichiers. Ce choix constitue un des éléments de réponse pour les questions de portabilité et de mise au point.

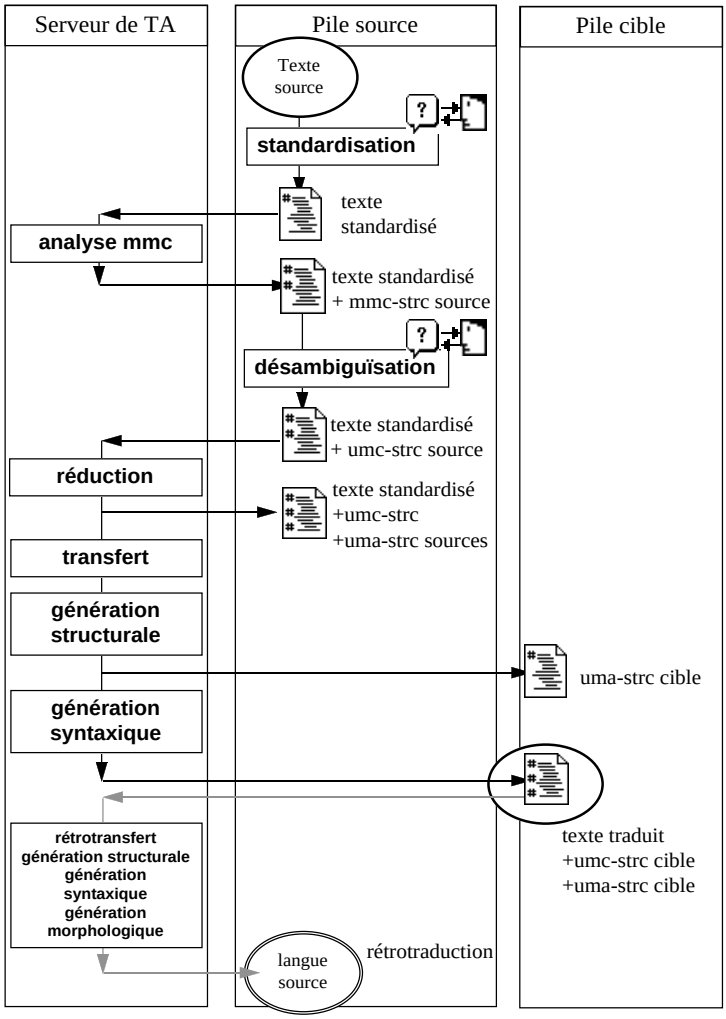


Figure 2.11 : Organisation générale du processus de traduction en LIDIA-1.

2.3. Une architecture de "tableau blanc"

L'expérience d'intégration menée avec ODILE et l'analyse des différents systèmes de TA que nous avons choisis, nous amènent à définir une architecture de tableau blanc ([Boitet & Seligman 1994], [Seligman & Boitet 1994]).

L'objectif de cette architecture est de faciliter l'intégration de composants de TALN. L'existant (les anciens composants) peut aussi exploiter cette architecture ; cependant,

il est souhaitable de définir et d'intégrer de nouveaux LSPL qui puissent pleinement en profiter.

2.3.1. Problèmes des autres approches

Les architectures séquentielles posent deux problèmes majeurs, à savoir la perte de l'information linguistique à travers les différentes phases de traitement et le manque de robustesse.

Complexité

La compéhension du fonctionnement des différents systèmes de TALN nécessite de se pencher longtemps sur les interrelations et les interfaces entre composants et sous-composants. Il est à notre avis souhaitable de disposer d'une architecture intégrant tous les composants selon la même approche, le contrôle des tâches entre ces composants étant centralisé par ailleurs.

Perte d'information

Si les composants d'un système sont simplement concaténés, il leur est difficile de partager des résultats partiels. L'interface entre les composants aboutit souvent à des pertes de certaines informations qui doivent être recalculées par la suite. Le travail est donc inutilement multiplié.

Dans ASURA par exemple, [Morimoto, Takezawa & Yato 1992], un analyseur LR est utilisé lors de la reconnaissance de la parole. Cependant, les structures syntaxiques trouvées sont détruites quand le résultat de la reconnaissance est passé au système de TA. Une analyse complète doit à nouveau être entreprise.

Manque de robustesse

Il y a souvent des problèmes de communication entre les différents composants.

Dans ASURA par exemple, si aucune phrase bien formée n'est trouvée, les structures syntaxiques partielles sont détruites avant l'analyse sémantique. Il est donc impossible de produire une traduction partielle ou d'utiliser les informations sémantiques pour essayer de compléter l'analyse.

L'approche par "tableau noir" [Erman & Lesser 1980] permet de régler le problème de la perte d'information, mais pas celui de la robustesse. De plus, le tableau noir introduit de nouvelles difficultés que nous détaillons.

Contrôle des accès concurrents

En principe, tous les composants ont accès au tableau noir. Des mécanismes de synchronisation sont nécessaires, et des composants rapides peuvent être considérablement ralentis.

Manque d'efficacité

Chaque composant effectue ses traitements directement sur le tableau noir. Il n'est donc pas possible pour un composant d'utiliser des structures de données optimales.

Surcharge de communication

Avec le tableau noir, la quantité d'information est souvent très importante. L'efficacité des systèmes distribués souffre de cette surcharge d'information qui amène à des communications de structures de données trop volumineuses.

Problèmes de mise au point

Il est difficile de développer un composant en tenant compte de l'ensemble des informations que peut contenir le tableau noir. De plus, l'ensemble du processus (de traduction, en ce qui nous concerne) est massivement parallèle.

2.3.2. Présentation générale

L'architecture de tableau noir est bien entendu à l'origine de celle de tableau blanc. On pourrait aussi citer l'architecture de Charts du système MIND [Kay 1973] et les travaux de [Colmerauer 1970] comme source d'inspiration.

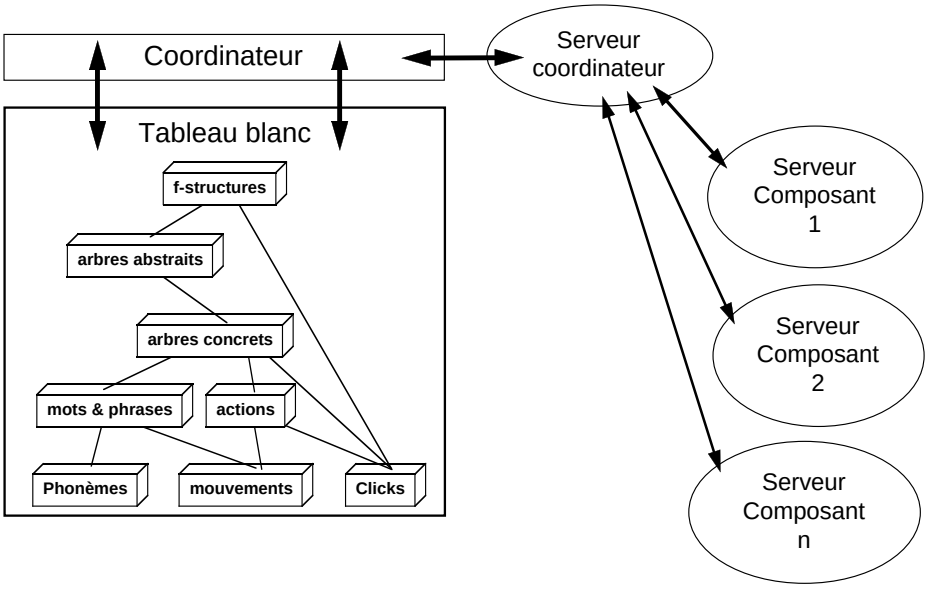


Figure 2.12 : Architecture générale de tableau blanc.

Du point de vue du coordinateur, les composants sont cachés par des serveurs qui reçoivent des requêtes. Les composants eux-mêmes peuvent formuler des requêtes et ne voient que le serveur du coordinateur. Le coordinateur n'est donc qu'un composant particulier qui centralise les flux de communication.

Centralisation

Dans l'architecture de tableau blanc, la structure de données globale est cachée aux composants. Seul un coordinateur a accès à cette structure. Cette idée est similaire à celle de communication par forum modéré (moderated forum-based communication) de [Andreoli & Pareschi 1991].

Avec cette approche, les problèmes liés au tableau noir disparaissent. Cela a deux conséquences supplémentaires :

- Les développeurs de composants sont encouragés à définir et à publier les structures de données utilisées en entrée et en sortie. Le niveau de description doit au moins être suffisant pour représenter ces données dans le tableau blanc.
- La tableau blanc est le lieu où se déroule l'interaction avec le linguiste. L'inspection des structures de données peut se faire à différents niveaux de détail.

La centralisation est un des aspects majeurs (hérité du tableau noir) de l'architecture de tableau blanc. Certains systèmes évitent délibérement l'approche centralisée (par exemple [Stefanini, Berrendonner, Lallich & al. 1992]). De plus, nous pensons que la centralisation n'est pas nécessairement synonyme d'approche séquentielle.

Une architecture sera qualifiée de tableau blanc à partir du moment où elle définit une structure de données commune, cachée des composants et à laquelle seul un coordinateur a accès. La structure de données utilisée n'est pas imposée.

Une structure de données adaptée pour le tableau blanc peut être la structure de treillis présentée au §1.3.1.

Coordination

Dans sa forme la plus simple, le coordinateur transmet les résultats d'un composant au composant suivant. Cependant, le coordinateur est en mesure de conduire des stratégies globales. Par exemple, seuls les résultats les plus probables peuvent être transmis dans un premier temps. Des contraintes sur la forme des résultats peuvent être liées aux structures.

Encapsulation

Les développeurs de composants peuvent choisir les algorithmes et les structures de données adaptés. Ils ne sont pas contraints d'utiliser une structure de données particulière comme pour le tableau noir. Les composants sont encapsulés dans des gestionnaires. Les gestionnaires cachent les composants et les transforment en serveurs. Les structures de données des composants sont traduites par le gestionnaire en structures acceptables par le tableau blanc, et vice-versa.

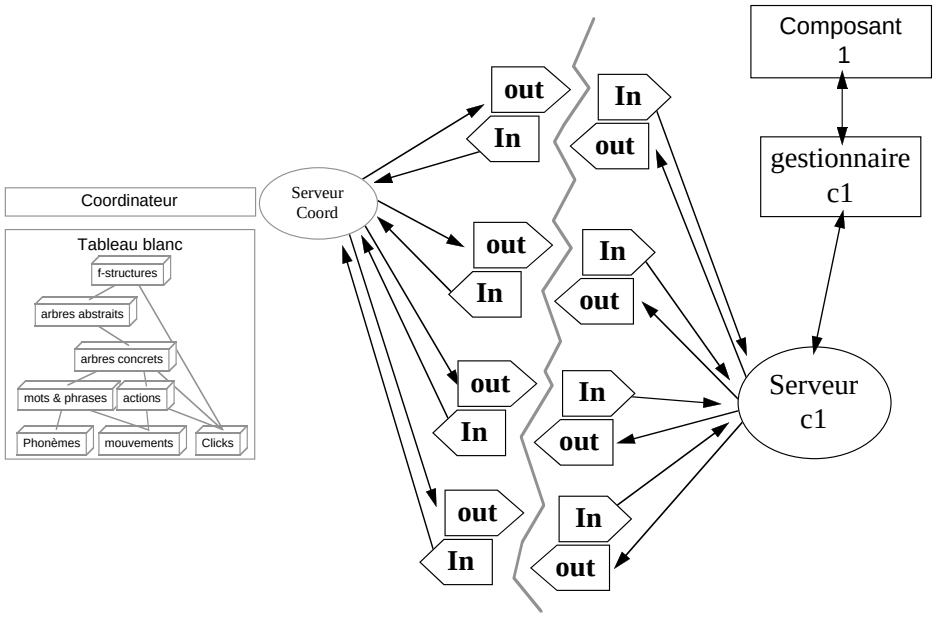


Figure 2.13 : Communication entre composant, gestionnaire, serveur et coordinateur.

Un composant est caché par son serveur. Un gestionnaire fait l'interface entre les structures de données reçues par le serveur depuis le tableau blanc et celles du composant. Les couples de boîtes aux lettres ("In" et "Out") associées à des transactions sont à la base de la communication entre les serveurs.

Les appels directs entre le coordinateur et les composants sont donc interdits. Pour intégrer un nouveau composant, il suffit d'écrire un nouveau gestionnaire.

Un gestionnaire dispose d'une boîte à requêtes où les clients peuvent demander l'ouverture et la fermeture de connexions. Une connexion se fait par l'intermédiaire de deux boîtes aux lettres (entrée et sortie). Le coordinateur donne le travail à effectuer dans les boîtes aux lettres d'entrée et récupère les résultats dans les boîtes aux lettres de sortie.

Processus incrémental, simulation et visualisation

La centralisation des informations dans le tableau blanc constitue une bonne approche pour la visualisation des traitements. Les processus mis en jeu sont complexes et leurs effets sur la structure de treillis difficiles à cerner par le linguiste.

Le tableau blanc est divisé en régions bien séparées qui représentent le "travail" des composants : entrées, résultats, les liens entre eux (les nœuds gris de [Boitet 1988]) et l'histoire de la production des résultats.

2.3.3. Classes d'objets liées à l'architecture

Pour réaliser cette architecture, il est nécessaire de définir des classes d'objets. Ces classes sortent du cadre de LEAF, car elles ne sont pas considérées comme pertinentes du point de vue des fonctionnalités du linguiciel. Par contre, elles sont incontournables du point de vue logiciel.

Une bonne part de ce travail est générique. En effet, certaines classes seront instanciées un grand nombre de fois. De plus les différentes classes de serveurs ou de coordinateurs seront assez semblables. C'est pourquoi l'approche par objets (incluant l'héritage) semble particulièrement adaptée au développement de l'ensemble des classes présentées ci-dessous.

Tableau blanc

La tableau blanc encapsule une structure de données choisie comme représentation centralisée pour le système. Bien que la structure de treillis ait notre préférence, aucune structure n'est imposée par l'architecture.

Coordinateur

Une des originalités de l'architecture de tableau blanc est de s'affranchir d'une description directe des interactions entre les composants. L'interaction se fait implicitement via le coordinateur. La description d'un système peut se faire en deux étapes :

- énumération des composants ;
- description du contrôle exercé par le coordinateur.

Ce deuxième point peut être abordé de multiples manières ; l'architecture de tableau blanc n'est pas explicite sur ce point. Cependant il est intéressant de débroussailler le terrain.

La description sous forme d'automates d'états finis offre une approche relativement intuitive au linguiste. Une telle description est aisément modifiable. Ajouter des

fonctions consiste simplement à rajouter des états ou des transitions. De plus, nous disposons d'outils efficaces et génériques pour son implémenntation[25].

Les coordinateurs sont proches des "médiateurs" de [Nishida & al 1993]. Cependant, l'approche globale est différente puisque la "Knowledgeable Community" est basée sur une architecture à agents.

Gestionnaires

Les gestionnaires ont en charge la traduction des structures de données (dans les deux sens). La définition de filtres permettant d'effectuer des conversions entre les treillis, les grilles et la carte constituent l'essentiel du travail de définition des classes de gestionnaires.

Les gestionnaires sont proches des "facilitateurs" de [Nishida & al. 1993].

Serveurs

Les serveurs ont en charge la gestion des boîtes aux lettres, la constitution des requêtes pour les composants, et la conversion des structures de données échangées entre le (ou les) formats du composants et celui (ou ceux) du coordinateur.

Boîtes au lettres

Il existe deux classes de boîtes aux lettres, celles de réception et celles d'émission. Une boîte aux lettres envoie à une autre boîte aux lettres le message qui contient la requête. La forme par défaut des messages est le ficher texte. Une telle forme est raisonnablement portable, et de plus, sa lisibilité facilite la mise au point. Toutefois, rien n'exclut l'utilisation d'autres formes de message.

2.3.4. Exemple d'application du Tableau Blanc au système LIDIA

Il est intéressant d'essayer d'appliquer cette architecture de tableau blanc à un système déjà existant. Nous avons choisi pour cela le projet LIDIA. En effet, l'architecture de la maquette actuelle n'est pas très éloignée du tableau blanc.

Le serveur LIDIA et le serveur de communication sont presque les équivalents du coordinateur et du serveur du coordinateur. Pour coller au plus près à l'architecture de tableau blanc, il suffirait de transformer l'ensemble des composants en des serveurs vus par le coordinateur.

Application récursive

L'idée du tableau blanc peut et doit être vue de manière récursive[26] afin de décomposer les composants de grande taille pouvant potentiellement intégrer eux-mêmes des sous-composants. Lorsqu'un composant n'est pas, a priori, décomposable, on parlera de composants atomiques. Nous penserons par exemple aux correcteurs ortograhiques qui ne sont généralement pas décomposable en éléments distribués.

Le superviseur des traitements linguistiques de LIDIA est un composant non atomique. Ses composants sont les différentes machines ARIANE. On constate que, dans cette architecture, le serveur de communication est caché. Cela a pour conséquence de

---

25 Le troisième chapitre présente un langage basé sur un gestionnaire d'automates. Les automates envisagés peuvent être déterministes ou non, augmentés de variables, etc.

26 L'application récursive de ce genre d'architecture est inspirée des travaux fait, par exemple, sur PAC [Coutaz 1988].

clarifier la décomposition de la maquette LIDIA en séparant les éléments de nature linguistique de ceux dédiés à la gestion de la communication.

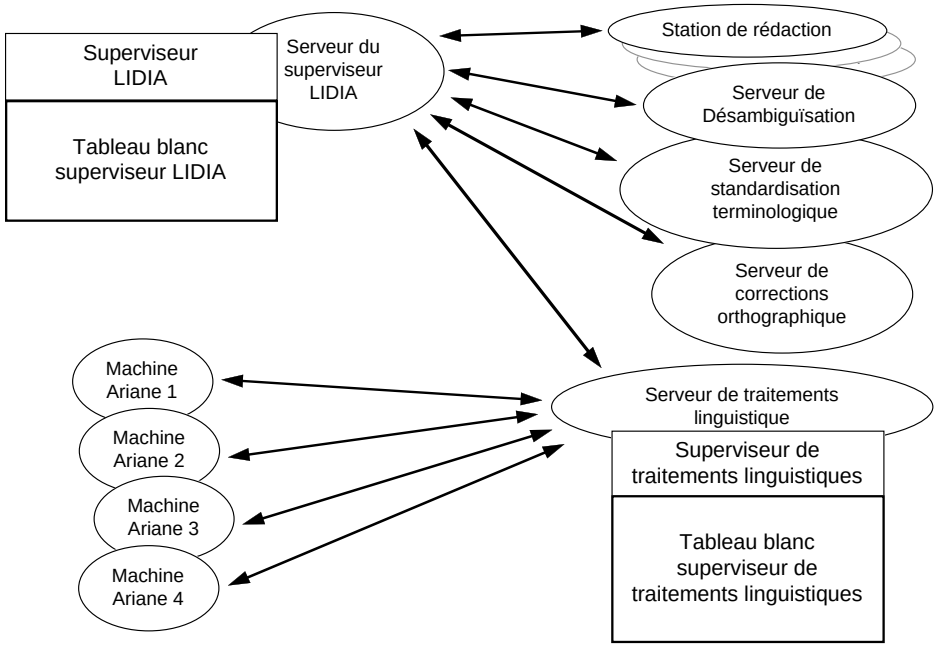


Figure 2.14 : Proposition d'application du tableau blanc à LIDIA.

Mis à part le superviseur de traitements linguistiques, les composants sont atomiques. Le serveur de traitements linguistiques permet de décomposer de manière récursive les machines nécessaires. Il est donc possible d'intégrer des traitements linguistiques hétérogènes.

Les machines ARIANE elles-mêmes pourraient être réinterprétés en termes de tableau blanc. Une telle approche permettrait de s'affranchir des contraintes liées à l'organisation des phases proposée par le moniteur d'ARIANE. Les phases d'articulation seraient à la charge de chaque gestionnaire, et dépendraient de la représentation linguistique utilisée par le composant et de celle de la "région" correspondante du tableau blanc.

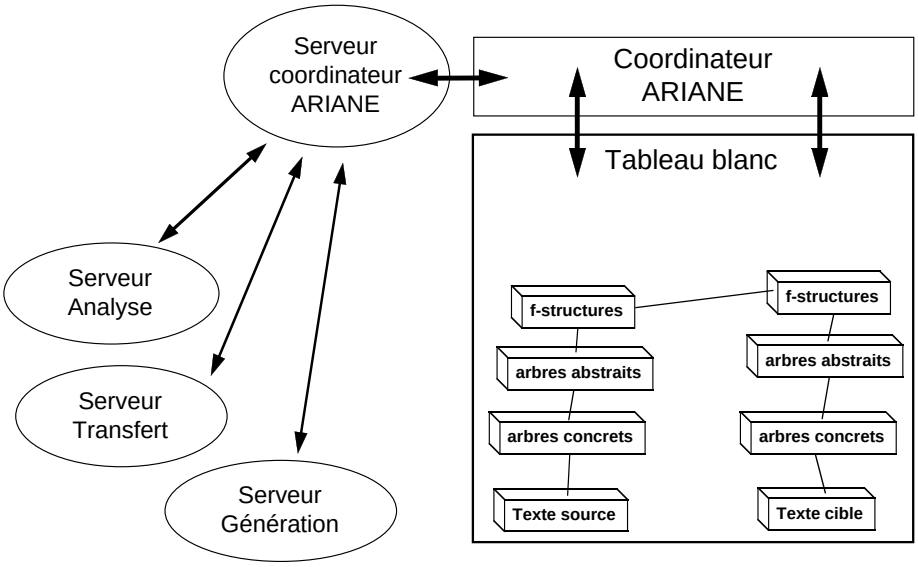


Figure 2.15 : Une application ARIANE sous forme de tableau blanc.

À la demande du coordinateur ARIANE, le serveur d'analyse produit une structure mmc. Il existe deux serveurs de transfert par couple de langues (un pour chaque sens de traduction). Les phases d'articulation (en TRACOMPL) sont distribuées dans des serveurs d'analyse, de transfert et de génération, mais aussi au niveau des gestionnaires de chacun de ces serveurs.

Chapitre 3

Problèmes de généricité et d'extensibilité

Object-oriented programming has been praised for many virtues, of which we believe code reuse to be the most important. […] The claim is that object-oriented techniques make it possible to build "class libraries" that are reusable in the sense that, when we later build systems that require this sort of functionality, we can reuse the library rather than having to code again from scratch. There are two important properties that cause class libraries to be reusable in this way: generality and extensibility.

G. Kiczales and J. Lamping in [Kiczales & Lamping 1992]

La généricité et l'extensibilité définies respectivement aux §1.2.2. et §1.2.3. débouchent sur un certain nombre de problèmes conceptuels et pratiques. Une solution a été proposée pour l'intégrabilité avec l'architecture de tableau blanc.

Une première expérience, qui met en œuvre des mécanismes permettant d'obtenir une bonne extensibilité dans un LSPL, a été réalisée avec la réingéniérie de LT, un langage d'écriture de transcripteurs. Le modèle de ce langage a d'autre part été scrupuleusement suivi dans une implémentation basée sur un gestionnaire d'automates génériques.

Nous explicitons comment l'utilisation de protocoles nous semble consituer une bonne approche pour obtenir des outils génériques et extensibles. Cependant, comment spécifier correctement un protocole ?

L'approche empirique consiste à partir d'une première spécification qui offre autant que possible à l'utilisateur une puissance d'expression et des possibilités d'extensions satisfaisantes. Cette première spécification peut s'avèrer relativement contraignante pour le développeur. Ce premier protocole est ensuite élargi par la définition de classes et de méthodes qui généralisent celles implémentées. La spécification d'origine est modifiée de façon à constituer une spécialisation (des classes et des méthodes) de la seconde spécification .

Certaines difficultés apparaissent lors de ce processus itératif. Par exemple, faire la distinction entre détails d'implémentation et parties cruciales de la spécification n'est pas toujours facile. De plus, il n'est pas aisé de déterminer quand la spécification a suffisamment gagné en généricité (et donc quand arrêter le processus itératif de spécification).

3.1. LT, une première approche

Le Langage de Transcription (LT) est un LSPL dont la première version a été définie et réalisée en PROLOG I par Ch. Boitet et un de ses étudiants au GETA durant l'hiver 82-83 dans le cadre d'un projet de DEA. Yves Lepage [Lepage 1986] a ensuite étendu la spécification et réalisé une seconde version opérationnelle mais beaucoup trop lente. Il a ensuite réécrit le moteur en Pascal/VS et le compilateur en PROLOG-CRISS, ce qui a donné une troisième version encore très lente mais utilisable et utilisée.

Le but initial de ce langage était d'offrir aux linguistes un formalisme agréable permettant d'écrire des transcripteurs entre divers systèmes de codage[27].

La version de LT (LT4) présentée ici est donc la quatrième. L'implémentation se base sur CLOS. Un gestionnaire d'automates génériques sert de fondation au langage. L'aspect le plus étudié dans cette version est l'extensibilité du langage [Lafourcade 1993b].

3.1.1. Description

Pour l'étude formelle du modèle de transducteur de LT, et des automates en général, nous renvoyons à [Lepage 1985], [Lepage 1986] et [Harrison 1978].

LT est un langage qui permet l'écriture de transducteurs caractérisés par :

- leur comportement déterministe ;
- leurs états décrits par les étiquettes et des variables ;
- une bande d'entrée ;
- une bande de sortie;
- deux têtes de lecture (une tête standard et une tête de lecture en avant) ;
- une tête d'écriture ;
- la lecture d'une chaîne non vide en entrée ;
- l'écriture d'une chaîne potentiellement vide en sortie.

Certains de ces points méritent un commentaire.

Un comportement déterministe fait aller l'automate toujours de l'avant, aucun retour arrière n'est possible. Si l'automate se trouve dans un état non final à partir duquel il ne peut aller vers aucun autre état, une erreur à l'exécution est signalée. L'impossibilité de pouvoir lire une chaîne vide en entrée évite les boucles infinies. Si la chaîne d'entrée est finie, l'automate s'arrêtera.

---

27 entre deux alphabets, par exemple.

La transduction peut être vue comme une opération simultanée de lecture et d'écriture (cette dernière faite en fonction de ce qui est lu). Le transducteur écrit des symboles sur la bande de sortie à mesure qu'est lue la bande d'entrée. Le transducteur peut mémoriser son travail en changeant d'état (et en modifiant les valeurs des variables).

Séquence d'événements

La séquence des événements durant l'exécution d'un transducteur LT est la suivante :

- le transducteur démarre au début de la bande d'entrée. La bande de sortie est vide. Les variables ont comme valeur leur valeur initiale. L'état courant est l'état initial ;
- La première transition applicable est choisie. Une transition est applicable si l'expression de chaîne d'entrée peut être lue et si les conditions de chaîne d'entrée peuvent être appliquées ;
- la variable chaîne-lue est liée à ce qui a été lu dans le bande d'entrée ;
- l'action liée à la transition est exécutée. Cette action peut modifier les variables du transducteur ;
- l'automate passe dans l'état d'arrivée de la transition ;
- si l'exécution peut continuer, on retourne à la seconde étape, sinon on arrête.

Conditions d'arrêt

Plus précisément, le transducteur s'arrête si l'une des conditions suivantes est réalisée .

- arrêt normal : tête de lecture standard en fin de bande d'entrée et état final ; action d'arrêt ;
- arrêt anormal : état non final et aucune transition applicable.

(on peut distinguer deux cas selon que l'on est ou non en fin de bande d'entrée).

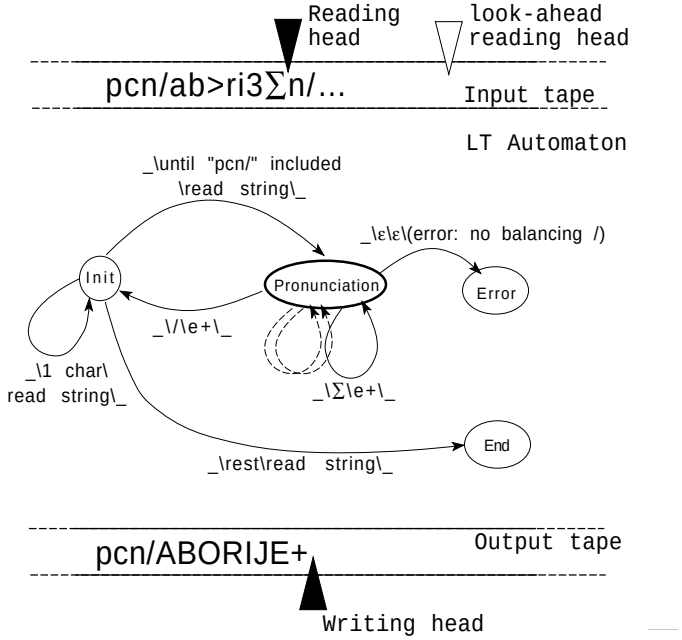


Figure 3.1 : Exemple d'automate LT pour la transcription de formes phonétiques.

Cet exemple est tiré d'une application de LT4 pour la transcription d'entrées de dictionnaires français-anglais-malais [Gaschler & Lafourcade 1994].

À propos de la tête de lecture avant

La tête de lecture avant est toujours en avant ou à la position de la tête de lecture standard. Si la tête de lecture standard avance et rencontre la tête de lecture alors elle l'entraînera avec elle — à la fin du mouvement les deux têtes de lecture seront donc à la même position.

La tête de lecture avant peut être "rétractée" à la position de la tête de lecture standard.

Primitives de LT4

Le langage de LT4 est composé d'un noyau fonctionnel et d'une couche syntaxique. Il est d'usage de parler respectivement de *langage noyau* et de *langage utilisateur.* Nous présentons ici certains aspects du langage noyau (d'autres détails sont donnés en annexes). Pour une définition exhaustive, nous renvoyons à [Lafourcade 1993b]. Quelques considérations liées au langage utilisateur sont exposées à la fin de ce chapitre.

Le langage noyau consiste en des fonctions Common Lisp permettant la définition et l'exploitation d'un transducteur.

Les fonctions de définition d'un transducteur sont les suivantes[28] :

`transcriptor (name [variable-list])` `[macro]`

Cette fonction associe à la variable *name* un transcripteur vide (sans état ni transition). Le transcripteur dispose, éventuellement, d'une liste de variables avec des valeurs par défaut.

```
(transcriptor lt4-example ())
```

`initial-state (name transcriptor)` `[macro]`

Cette fonction associe un état initial, *name*, au transcripteur *transcriptor*. Un seul état initial doit être défini.

```
(initial-state init lt4-example)
```

`final-state (name+ transcriptor)`

cette fonction définit une suite *name*+ d'états comme finals. La définition d'un ou de plusieurs états finals n'est pas obligatoire car le transducteur s'arrête quand la bande d'entrée est complètement lue et que aucune transition n'est applicable. Cependant, l'arrêt est considéré comme anormal et un message d'alerte est produit. Si l'état courant est final, le transducteur s'arrête si la bande d'entrée est vide.

```
(final-states end1 end2 lt4-example)
```

`transitions ((begin-state end-state transcriptor))`

Cette fonction permet de définir un faisceau de transitions entre deux états du transcripteur. Une transition doit avoir la forme suivante :

---

28 Les définitions proposées ici sont des simplifications tirées de [Lafourcade 1993b]. Certaines options ou mot-clés ne sont pas explicités.

```
(input-string-expression
condition action
output-string-expression)
```

Une expression de chaîne d'entrée (on parlera de "ISE" pour *input-string-expression*) spécifie ce qui peut être lu sur la bande d'entrée par la tête de lecture standard. Si cette expression est précédée du symbole "?", la tête de lecture avant est concerné. Les conditions et les actions sont des expressions LISP pouvant faire référence aux variables de l'automate. Une expression de chaîne de sortie (on parlera de "OSE" pour *output-string-expression*) spécifie ce qui doit être écrit sur la bande de sortie par la tête d'écriture.

Par exemple :

```
(transitions (init path lt4-exemple)
   ((lg 1)
    () ()
    (string-upcase read-string))
   (? (lg 2)
    () ()
    read-string)
 )
(transitions (init path lt4-exemple)
   ("A!3"
    () ()
    "à")
 )
```

Le premier faisceau définit deux transitions entre les états *init* et *path*. La première transition permet la lecture d'une chaîne réduite à caractère quelconque et la réécrit en majuscules. La seconde transition permet à la tête de lecture avant de lire une chaîne réduite à deux caractères quelconques et de réécrire telle quelle. Le deuxième faisceau définit une transition transcodant la chaîne "A!3" en "à".

3.1.2. Compilation vers un automate

Le noyau LT4 se compile directement sous forme d'automates d'états finis déterministe. Des développements intéressants sur cette question peuvent être trouvés dans [Gazdar & Mellish 1989].

Modèle abstrait comme base

L'automate est un modèle facile à comprendre, facile à implémenter et surtout efficace. De plus, ce modèle est connu, et il a donné lieu à de nombreuses expériences. Nous pensons, par exemple à [Koskenniemmi 1984] avec KIMMO ou encore à [Karttunnen 1993]. Les domaines d'application ont été la phonologie et la morphologie.

Les fonctionnalités de notre machine abstraite GAM (Generic Automaton Manager) se divisent selon deux niveaux. D'une part, un protocole de définition pour toutes les classes de GAM, et d'autre part des protocoles particuliers liés à des spécialisations de certaines classes.

Le protocole général permet de définir des automates, des états et des transitions ainsi que de lancer un automate :

```
defautomata (automaton-name variables &key class)            [macro]
defstate ((state-name automaton-name) status &key action     [macro]
```

```
destransition (  (transition-name automaton-name)
                 (start-state end-state)
                 &key condition action)                          [macro]
run-automaton (automaton)                          [fonction générique]
```

Un exemple de code GAM généré à partir de LT4 est présenté plus bas.

Les sous-protocoles permettent un contrôle fin sur le fonctionnement de l'automate. Par exemple, l'ordre de considération des transitions, l'application des actions et des conditions, la gestion des variables, etc. Il n'est pas nécessaire de descendre à ces niveaux pour définir correctement un automate.

On dispose donc d'une hiérarchie d'implémentations à plusieurs niveaux (figure 3.2). L'utilisateur peut définir son transcripteur dans un formalisme externe (que nous ne détaillons pas ici) qui est traduit en LT4. LT4 lui-même s'exprime directement en fonction du GAM. Il aurait été possible de définir des classes et des méthodes dérivées du GAM afin de réaliser le noyau LT4. Nous n'avons pas choisi cette approche, car elle n'est pas stratifiée, et ne correspond plus à la définition d'un langage de programmation.

L'utilisation du GAM revient à exprimer LT4 en fonction des primitives d'une machine abstraite. Cette machine abstraite est d'un niveau inférieur à ce qu'elle permet d'exprimer, on diminue ainsi le niveau de complexité de l'implémentation.

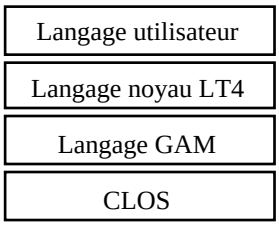


Figure 3.2 : Hiérarchie de langages de LT4.

Un programme LT4 est compilé en langage GAM. Le langage GAM qui permet de définir des automates généraux peut être également être vu comme une boîte à outils. Ce langage est implémenté à base d'objets CLOS pouvant être dérivés ou spécialisés. L'utilisateur non informaticien peut programmer un transducteur LT à partir d'une syntaxe plus abstraite que celle proposée par le langage noyau. Ce "langage utilisateur" est compilé en langage noyau.

Par exemple, nous avons dans le tableau ci-dessous un extrait d'un transcripteur d'une chaîne de caractères (contenant des accents) en un codage utilisé dans le système ARIANE (ISO-8809). La syntaxe externe est à gauche et le code LT4 généré à droite :

```
transcripteur mac-to-ariane
l'état initial est init

de init à init via
  lire 1 caractère si (chiff-p chaîne lue)
    puis écrire chaîne lue
  lire 1 caractère si (ponct-p chaîne lue)
    puis écrire chaîne lue
  lire #\Newline puis écrire chaîne lue

de init à test via
  lire ? 1 caractère si (maj-p chaîne lue)
    puis écrire ""
  lire ? 1 caractère si (maj-diacr-p chaîne lue)
    puis écrire ""

de test à mot-cap via
  lire ? 1 caractère si (maj-p chaîne lue)
    puis écrire "" et faire (retract self)

de mot-cap à mot-min via
  lire "Á" puis écrire "A!1"
  lire "À" puis écrire "A!2"
  lire "Â" puis écrire "A!3"
  lire "Ä" puis écrire "A!4"
  lire "Ã" puis écrire "A!5"
…
```

```
(in-package :lt)

(transcriptor mac-to-ariane nil)

(initial-state (init mac-to-ariane))

(transitions (init init mac-to-ariane)
 ((lg 1) (chiff-p read-string)
         nil read-string)
 ((lg 1) (ponct-p read-string)
         nil read-string)
 (#\newline nil nil read-string))

(transitions (init test mac-to-ariane)
 (? (lg 1) (maj-p read-string) nil "")
 (? (lg 1) (maj-diacr-p read-string) nil ""))

(transitions (test mot-cap mac-to-ariane)
 (? (lg 1) (maj-p read-string)
           (retract self) ""))

(transitions (mot-cap mot-min mac-to-ariane)
 ("Á" nil nil "A!1")
 ("À" nil nil "A!2")
 ("Â" nil nil "A!3")
 ("Ä" nil nil "A!4")
 ("Ã" nil nil "A!5"))
…
```

La taille du code GAM généré à partir de LT4 est important et nous n'en donnons que les premières lignes (celles avec une bordure en pointillée) :

```
(progn
  (defvar mac-to-ariane ())
  (setf mac-to-ariane
        (defautomata mac-to-ariane ((read-string "")
                                    (lethal-string ""))
          :class lt-transcriptor-class))
  (put-prop (states-list mac-to-ariane)
            (gam-name (internal-error-state mac-to-ariane)) nil)
  (setf (internal-error-state mac-to-ariane) nil)
  (defstate (internal-error-state mac-to-ariane) :error
    :action
    (nil
     (when (string-not-equal
             (buffer-substring (buffer (input-tape mac-to-ariane)) t)
             "")
       (print "error state : no applicable transition")))))

(defstate (init mac-to-ariane) :initial)

(deftransition (|transition798| mac-to-ariane) (init init)
  :condition
  ((lethal-string read-string self)
   (and (or t lethal-string read-string self)
        (let* ((|tape799| (input-tape self))
               (|test798|
                (>= (length-rest-string |tape799| 'standard-pointer) 1)))
          (when |test798|
```

```
(setf (read-string self)
              (buffer-substring
                (standard-pointer |tape799|)
                (+ (buffer-position (standard-pointer |tape799|)) 1)))
            (setf (offset self) 1))
          |test798|)
        (setf read-string (read-string self))
        (chiff-p read-string)
        (move-mark (standard-pointer (input-tape self)) (offset self))
        t))
   :action
   ((lethal-string read-string self)
    (let ((|tape800| (input-tape self)))
      (or t lethal-string read-string self)
      (setf read-string (read-string self))
      (when (< (buffer-position (forward-pointer |tape800|))
               (buffer-position (standard-pointer |tape800|)))
        (set-mark (forward-pointer |tape800|)
                  (buffer-position (standard-pointer |tape800|))))
      (setf lethal-string
            (buffer-substring
              (buffer |tape800|)
              (buffer-position (standard-pointer |tape800|))
              (buffer-position (forward-pointer |tape800|))))
      (buffer-insert (buffer (output-tape self)) read-string)
      nil)))
…
```

Automate = Classes + Protocole

Les classes liées au GAM sont très génériques. Il est facile de les utiliser telles quelles ou de les dériver. Quelques expériences (dont celle de ROBRA présentée au chapitre 8) ont montré qu'il est relativement aisé de définir toute une famille d'automates à partir

des classes de base du GAM (ces dernières définissent essentiellement des automates d'états finis déterministes).

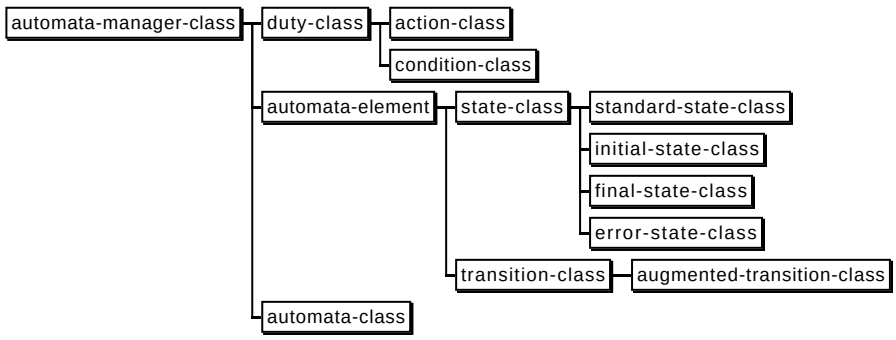


Figure 3.3 : Hiérarchie des classes de base du GAM.

Par exemple, les classes du gestionnaire GAM peuvent être dérivées de façon à disposer :

- d'automates à pile ;
- d'automates récursifs ;
- du non-déterminisme.

La dérivation de ces classes conduit d'une part à définir des sous-classes d'automates ayant des comportements différents, et d'autre part à également modifier le protocole. Il est possible (mais non certain) que le protocole externe de GAM voit un certain nombre de messages se rajouter. Il est plus probable que les sous-protocoles acceptent de nouveaux messages ou soient complètement modifiés afin de permettre un contrôle fin des nouveaux comportements (par exemple, le déterminisme peut être unaire ou n-aire). Enfin, de nouveaux protocoles doivent être créés (par exemple, un protocole pour la gestion des piles).

Pour le GAM, nous avons les définitions[29] suivantes:

```
(defclass automata-class (automata-manager-class)
 ( (current-state   :initarg :current-state
                    :accessor current-state
                    :documentation …)
   (next-state      …)
   (initial-state   …)
   (internal-error-state  …)
   (state-list …)
   (trans-list …)
   (var-list …) ) )

(defmethod automata-run ((self automata-class))
   (setf (next-state self) (initial-state self))
   (do () ((not (nest-state self)))
      (state-run (next-state self)))
   ;; when it's over, returns the variable list of the automaton
   (variable-list self))
```

29 Les définitions sont simplifiées par rapport au code original. Ce dernier est plus important car l'implémentation est "défensive" (vérification du type de certains arguments, etc.) et"outillée" (possibilité de trace, etc.). Seule une sélection des définitions est présentée ici.

```
(defclass state-class (automata-element)
 ( (in-transition …)
   (out-transition …)
   (status …) ) )
(defmethod state-run ((self state-class))
   …)
(defclass transition-class (automata-element)
 ( (start-state …)
   (end-state …) ) )
(defmethod transition-run ((self transition-class))
   …)
…
```

3.1.3. Généricité et Extensibilité dans LT

Cette première expérience permet de dégager quelques points où la généricité et l'extensibilité ont pu être exploitées.

Composition

La compilation en termes d'automates est déjà une expression de la généricité réalisée par le GAM. Il est donc possible de définir des classes d'application intéressantes en fonction de ce gestionnaire.

LT4 définit des classes de têtes de lecture. Comment remplacer les classes composants par des instances d'autres classes ?

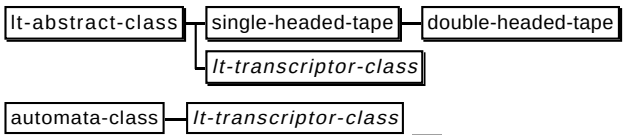


Figure 3.4 : Hiérarchie de classes de LT.

Deux classes de tête de lecture ont été définies. On a volontairement décidé de ne pas contraindre la lecture ou l'écriture, mais plutôt de faire la distinction sur le nombre de têtes. Si un nombre quelconque de têtes devait être envisagé, cette hiérarchie devrait être remise en question.

Le principe de base de la composabilité est de ne pas verrouiller la classe des objets qui composent une instance. Les classes des composants doivent pouvoir être modifiées lors de la création d'une instance ou par dérivation. Par exemple, nous pouvons définir la classe de transcripteurs LT par :

```
(defclass LT-transcriptor-class (automata-class lt-abstract-class)
   ((input-tape  :initform nil
                 :accessor input-tape
                 :documentation "input-tape to be transcribed.")

    (output-tape :initform nil
                 :accessor output-tape
                 :documentation "output-tape (transcribed).")
   …
   ))
```

Cette classe hérite d'une classe abstraite *lt-abstract-class* et d'une classe (très concrète, celle-là) d'automates, automata-class. Les deux attributs explicités correspondent aux bandes de lecture et d'écriture.

Considérons la fonction d'initialisation suivante :

```
(defmethod initialize-instance ((self LT-transcriptor-class) &key)
   (when (next-method-p) (call-next-method))
   (setf (input-tape self) (make-instance 'double-headed-tape))
   (setf (output-tape self) (make-instance 'single-headed-tape))
   )
```

Cette fonction d'initialisation passe la main à la superclasse, puis crée des instances de bandes d'entrée et de sortie. Cependant cette approche ne permet pas de définir des sous-classes de *LT-transcriptor-class* disposant de classes de bande différentes.

L'approche que nous adoptons pour la composabilité est la suivante :

```
(defmethod initialize-instance ((self LT-transcriptor-class) &key)
   (when (next-method-p) (call-next-method))
   (setf    (input-tape self)
            (make-instance (choose-input-tape-class self)))
   (setf    (output-tape self)
            (make-instance (choose-output-tape-class self)))
   )
```

et:

```
(defmethod choose-input-tape-class ((self LT-transcriptor-class))
   'double-headed-tape)

(defmethod choose-output-tape-class ((self LT-transcriptor-class))
   'single-headed-tape)
```

Si on spécialise *choose-input-tape-class* avec la valeur *'single-headed-tape*, on obtient la définition d'un transducteur classique (à une seule tête de lecture).

Un protocole très simple offre ainsi une bonne composabilité. Il est possible de définir une sous-classe de *LT-transcriptor-class* qui disposerait de bandes de classes différentes (pour notre exemple, il s'agirait d'une hypothétique bande de sortie à deux têtes de lecture) :

```
(defclass another-LT-transcriptor-class (LT-transcriptor-class)
   ())

(defmethod choose-output-tape-class
                        ((self another-LT-transcriptor-class))
   'double-headed-tape)
```

Les classes de base de LT4 peuvent être dérivées et composées afin de définir des classes ayant des comportement différents.

La composition se fait donc par le développement d'API. Cependant, les fonctions définies à cet usage sont polymorphes (elles acceptent plusieurs types d'arguments comme paramètres). C'est pourquoi nous parlons plutôt de protocoles (cf §3.3.2.).

Extensibilité

L'extensibilité dans LT4 se caractérise par la possibilité d'ajouter des expressions de chaînes d'entrée et de sortie. Un mécanisme permet en effet au développeur informaticien de définir lui- même de telles expressions sans modifier (ni connaître) l'implémentation originale.

Une expression de chaîne d'entrée *(ISE)* peut être définie avec l'action *reader-action* :

```
reader-action (action expression-de-chaîne tête-de-lecture) [méthode]
```

Comme l'ISE est composée d'un mot-clé ou d'une liste dont le premier élément est un mot clé, la partie action doit contenir ce mot-clé. L'expression de chaîne contient l'ensemble de l'ISE (le mot-clé plus l'argument). Le dernier argument tête de lecture permet éventuellement de discriminer l'action selon la catégorie de la tête de lecture (standard ou avant).

Plusieurs étapes doivent être suivies pour une définition correcte du corps de l'action :

- définir un mot-clé suivi de un ou de plusieurs arguments ;
- définir une fonction qui retourne un résultat qui peut être interprété comme un succès (une chaîne de caractères dans le cas général) ou comme un échec (NIL dans le cas général) ;
- en cas de succès, mettre à jour les positions des têtes de lecture ;
- ne pas oublier de retourner le résultat qui déterminera le succès ou l'échec de la lecture.

L'exemple qui suit permet de lire la bande d'entrée si elle commence par une chaîne membre d'une liste de chaînes. La liste est ordonnée (ce n'est pas un ensemble).

```
(defmethod reader-action ((action (eql 'string-list)) string-list ptr)
   (let   (   (list-symbol (gentemp "list"))
              (string-symbol (gentemp "string"))
              (list (cdr string-list)) )

      (let*  (  (,list-symbol ',list)
                (,string-symbol (find (get-string self (,ptr self)) ,list-symbol
                                      :test #'(lambda (x y) (prefixe y x))))
             )

         (when ,string-symbol
            ;; maj de la variable chaîne-lue (read-string)
            (setf (read-string self) ,string-symbol)
            ;; maj de la tête de lecture
            (setf (,ptr self) (+ (,ptr self) (length ,string-symbol)) )
          )

         ;; on retourne la chaîne trouvée comme résultat
         ,string-symbol

   )))
```

Une expression de chaîne de sortie *(OSE)* peut être définie avec l'action *writer-action* :

`writer-action (action expression-de-chaîne)` [méthode]

Les arguments sont identiques à la définition du ISE, à ceci près que le dernier argument du ISE, le type de la tête de lecture, a été supprimé car il est sans signification ici.

Les étapes à respecter pour définir correctement le corps de l'action sont les suivantes :

- définir la chaîne résultat de la transcription. Ce résultat est généralement fonction de la chaîne lue, contenue dans la variable chaîne-lue de l'automate ;
- écrire la chaîne résultat sur la bande de sortie.

Comme exemple, on peut donner le OSE *read-string~* qui écrit sur la bande de sortie l'image miroir de la chaîne-lue.

```
(defmethod writer-action ((action (eql 'read-string~)) string-expr)
   (declare (ignore string-expr))
   `(setf (output-string self)
          (concatenate 'string (output-string self)
                               (reverse read-string)))
   )
```

Le processus d'extension des expressions de chaînes est efficace mais relativement difficile à comprendre. Il s'agit bien d'un protocole. On pourrait certainement l'améliorer mais il suffit à illustrer notre méthode.

3.2. Analyse du problème : classement des difficultés

Avec LT4, nous avons trouvé un certain nombre de réponses aux différents problèmes posés par la réalisation d'un LSPL. Cependant, en essayant de généraliser notre approche, nous avons vu un certain nombre de problèmes.

3.2.1. Difficultés liées à l'ouverture sur un langage algorithmique

Le noyau LT4 est ouvert sur LISP. Cela peut être considéré comme un avantage et être exploité par des informaticiens. En effet, nous disposons ici d'un outil doté d'une structure de contrôle particulière et qui présente un intérêt pour des applications très éloignées du TALN. Cependant, nous avons vu (§1.1.2.) que pour les linguistes, une syntaxe ouverte sur un langage de programmation général constituait une très mauvaise extension.

La stratification peut constituer une solution. Elle consiste à définir plusieurs niveaux de contrôle sur les objets. Typiquement, pour le GAM, le premier niveau correspond à la déclaration d'un automate, de ses états et de ses transitions. Un niveau plus fin, par exemple, consiste (nous l'avons vu) à exercer un contrôle sur l'ordre de parcours des transitions. L'analyse des différents niveaux est difficile. De plus, en programmation par objets, une fonctionnalité fine peut souvent être vue à la fois comme un paramètre lié au comportement d'une classe générale et comme une dérivation impliquant la création d'une sous-classe. Le choix entre ces deux possibilités est une des difficultés de l'analyse en objets.

La syntaxe externe d'un LSPL doit être figée afin d'offrir aux linguistes une base de travail solide. De plus, figer la syntaxe permet de garantir de bonnes possibilités de ré-ingénierie. Le langage peut alors être réimplémenté ailleurs sur la base de sa syntaxe externe (et de sa sémantique). Si la syntaxe est ouverte sur LISP, réimplémanter le langage exige aussi de réimplémenter le langage LISP, ce qui non seulement représente un travail bien trop lourd pour être réalisable mais est également en complète contradiction avec l'idée de langage spécialisé.

On peut concevoir qu'une syntaxe externe puisse être ouverte sur l'extérieur, mais seulement à partir d'un protocole permettant d'intégrer des fonctions externes. Dans le §3.3.4. nous présenterons une méthode générique pour offrir des syntaxes externes aux linguistes à partir de langages noyaux.

3.2.2. Difficultés liées à l'extensibilité du noyau fonctionnel

Là aussi, l'expérience de LT4 nous a montré un certain nombre de difficultés.

Définition délicate des points d'entrées

Nous pensons qu'un outillage serait nécessaire pour rendre plus facile d'accès la définition de nouvelles expressions de chaînes d'entrée et de sortie[30]. Cet outillage passe des fonctions simplifiant les possibilités d'extension mais les contraignant également selon des schémas pré-établis.

Par exemple, on pourrait considérer que la chaîne de sortie va toujours être concaténée à la bande de sortie. Il serait donc possible de limiter l'extension à un protocole où le résultat de la méthode *writer-action* (ou de son équivalent) serait toujours une chaîne. Par exemple, on peut définir la fonction générique :

```
define-writer-action (action-keyword data-symbol &rest body)
                                                    [fonction générique]
```

qui génére une nouvelle méthode*write-action* spécialisée sur *action-keyword* et qui applique sur *data-symbol* la définition liée à *body* !

```
(define-writer-action 'read-string~ data
   (declare (ignore data))
   (reverse read-string)
 )
```

Le protocole exige donc que le corps de *define-writer-action* produise une chaîne de caractères. C'est au développeur de s'assurer que son code respecte cette contrainte, mais on peut adopter une programmation "défensive" en convertissant dans tous les cas le résultat à une chaîne de caractères[31]. La définition ci-dessus génére le code suivant :

```
(defmethod writer-action ((action 'read-string~)) string-expr)
   `(setf   (output-string self)
            (concatenate 'string
               (compute-output-string ,action ,string-expr)))
 )
(defmethod compute-output-string (   (action 'read-string~))
                                        data)
   (declare (ignore data))
   (reverse read-string)
 )
```

Quelle que soit la spécialisation de *writer-action* sur l'argument *action,* nous constatons que le code généré pour cette méthode est constant. On peut donc générer seulement la méthode *compute-output-string* et de ne pas spécialiser *writer-action :*

```
(defmethod writer-action (action string-expr)
   `(setf   (output-string self)
            (concatenate 'string
               (compute-output-string ,action ,string-expr)))
 )
```

On obtient de cette manière un protocole qui facilite la définition d'une nouvelle expression de chaîne. On a caché la concaténation de la chaîne de sortie qui ne doit

---

30 On retrouve ici l'idée qui motive, au moins en partie, la définition de syntaxes externes. Dans le cas de LT, une syntaxe de type Algol avec des mots clés tirés d'une langue naturelle est plus facile à apprendre pour les linguistes qu'une syntaxe lispienne .

31 à l'aide de la fonction *format.*

plus être gérée explicitement. De plus, le développeur n'est pas contraint d'utiliser ce mécanisme car il peut toujours spécialiser *writer-action*.

Vers une API culture ?

La définition d'interfaces logicielles permettant l'extension est un exercice difficile. Il est d'abord nécessaire de déterminer où il est intéressant de pouvoir permettre l'extensibilité. Une mauvaise analyse des besoins peut aboutir à des définitions trop générales (et donc inutiles car trop vagues pour être exploitées) ou trop spécifiques (et donc trop pointues pour justifier un développement alourdi). De plus, une spécification trop précise risque fort d'être liée à une implémentation particulière et de ne plus offrir aux développeurs de marge de manœuvre.

Afin de favoriser l'extensibilité, nous pensons qu'il ne faut donc pas définir des interfaces logicielles liées à des types de données particuliers mais plutôt s'orienter vers des protocoles. Plusieurs types de données différents peuvent répondre à un protocole donné. Ces questions sont explicitées aux §3.3.1. et §3.3.2.

3.2.3. Difficultés liées à la généricité des objets de base

La généricité permet à l'outil d'être rapidement adapté à de multiples usages. C'est parce que le GAM présente une bonne généricité que le LSPL LT4 peut être compilé en ses termes.

De plus, la généricité bien conçue permet de définir et de réaliser de nouveaux outils à moindre coût. C'est l'idée du MOP (Metaobject Protocol de [Kiczales, Rivières & Bobrow 1991][32]) où CLOS peut être dérivé vers de nouveaux espaces de comportement. Les démarches classiques adoptent, en général, une approche d'encapsulation (ou de boîte noire) où il est difficile de modifier l'existant.

Des composants génériques ont une durée de vie étendue, car ils servent de base à la réalisation de multiples outils.

3.3. Boîtes à outils et protocoles

C'est à la fois les outils et la boîte à outils elle-même qui doivent être génériques. De nouveaux outils doivent pouvoir être rajoutés, la généralisation et la spécialisation n'étant pas des processus que l'on peut figer [Kiczales & al. 1992]. Le modèle à objets semble constituer une bonne base de départ pour la réalisation de composants linguistiques. Nous définissons quelques principes généraux et proposons encore une approche par protocoles.

3.3.1. Types et Classes

Les méthodologies liées à la programmation à objets (voir par exemple [Massini, Napoli, Colnet & al. 1991]) offrent des perspectives intéressantes pour le TALN. Les quatre éléments majeurs du modèle à objets sont l'abstraction, l'encapsulation, la modularité et la hiérarchisation [Booch 1992].

---

32 à ne pas confondre avec les "Memory Organization Packets" de Yale (Schank)

Applications au TALN

Un certain nombre de travaux ont déjà appliqué au TALN le modèle à objets. Les approches sont toutefois très diverses.

Les travaux de [Akasaka & al. 1989] visent à construire une boîte à outils pour le TALN. Les outils sont basés sur deux langages spécialisés, ESP et CIL. ESP est un PROLOG à objets. CIL est un PROLOG augmenté d'une structure d'enregistrement appelée PST (Partially Specified Term), et d'un mécanisme de gel issu de PROLOG 2. Un PST est un ensemble de paires attribut-valeur équivalentes à une catégorie en Grammaire Syntagmatique Généralisée (GPSG) ou à une structure fonctionnelle en LFG. L'ensemble des modules est intégré dans une "coquille" (LTB-shell).

La technique mise en œuvre par [Habert 1992] est assez élaborée. Elle consiste à créer dynamiquement les classes nécessaires pour l'exécution de grammaires à unification. Une classe spécifique est créée pour chaque élément de la grammaire. L'analyseur, OLMES, a été développé en CLOS. Cette approche par création dynamique de classes parait intéressante mais elle semble poser des difficultés de mise au point.

Sous-typage et sous-classage

L'approche par objets distingue l'héritage de type et l'héritage de classe. D'autres types d'héritage peuvent être distingués [Girod 1991]. Certaines approches introduisent dans la sémantique du langage de programmation la distinction entre ces deux héritages [Porter 1992]. Bien que ce double héritage soit nécessaire, c'est habituellement au seul niveau de l'analyse qu'il est visible — les langages ne l'imposent généralement pas.

Un type distingue les catégories d'objets de manière à contrôler quelles valeurs peuvent être utilisées. Les opérations polymorphiques acceptent des objets de types différents pourvu qu'il héritent d'un même type de base.

Une classe implémente un type, c'est-à-dire qu'elle définit le comportement d'un objet de ce type. Une classe réutilise le code de la classe parent (la superclasse). L'héritage de classes ne concerne que le partage du code et n'implique pas que les types correspondant aux classes soit sous-types l'un de l'autre.

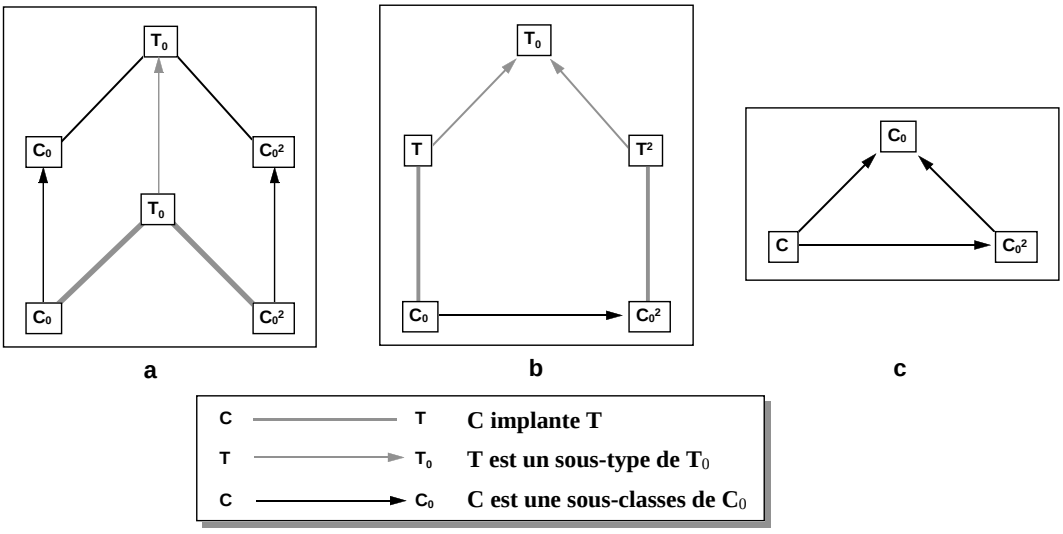


Figure 3.5 : Relations entre types et classes.

Sous-typage et sous-classage (a), sous-classage dynamique par délégation (b), approche par classes abstraites (c).

Il est désirable qu'un type puisse hériter du code sans qu'on ait à redéfinir un sous-type du type de base. S'il y a confusion entre types et classes, il est facile d'introduire des relations de types mal construites. Par exemple, on pourrait définir une classe ensemble comme une sous-classe de liste, et ainsi hériter la plupart du code puisque leur implémentation n'est pas très éloignée. Cependant, d'un point de vue mathématique, un ensemble n'est pas une sorte de liste ; les deux classes se comportent différemment pour l'opération de mise à jour. Une fonction attendant une liste comme paramètre peut être appelée avec un ensemble et échouera. Il est théoriquement indécidable de vérifier la conformité d'une classe par rapport à son type.

Cependant, ce problème peut être contourné, si on dispose de l'héritage multiple. Les types peuvent alors être remplacés par des classes abstraites. L'héritage du code peut ainsi être segmenté. De plus, la délégation introduit un sous-classage dynamique[33] (figure 3.5 b).

Démarche

En plus de la contrainte ne pas dévoyer les quatre éléments majeurs du modèle à objets, il nous semble important de définir quelques principes spécifiques permettant d'implanter efficacement et simplement une boîte à outils génériques et extensibles.

- CLOS

Sauf mention contraire, l'ensemble des travaux et des exemples présentés dans cette thèse a été réalisé en CLOS. Ce langage présente de nombreuses fonctionnalités intéressantes pour la réalisation de protocoles (voir Annexe A, pour plus de détails à ce sujet). Cependant, il faut souligner que d'autres langages de programmation généraux auraient pu être utilisés, malgré certains aspects techniquement plus délicats à mettre en œuvre.

De façon générale, nous refusons de nous focaliser sur un seul langage de programmation, un seul formalisme linguistique, ou un seul LSPL pour la réalisation d'une boîte à outils pour le TALN. Nous avons montré combien l'intégration d'outils différents est nécessaire.

- modélisation naïve

Lors de l'analyse de LT4, nous avons essayé de réaliser concrètement les objets prévus par le modèle de LT. Souvent, certains objets du modèle ne sont pas explicitement implémentés et créent ainsi un décalage entre le modèle théorique et le modèle d'analyse. Ce décalage complique la compréhension de la réalisation.

L'introduction de classes abstraites permet de confondre directement les notions de type et de classe. Deux options sont alors possibles pour le partage du code. Soit ne tenter de factoriser la réalisation par une classe mère commune seulement lorsqu'il y a relation de types (et donc de classes). Soit essayer d'optimiser l'implémentation à l'aide de l'héritage multiple et la délégation. La seconde solution ne semble pas compliquer particulièrement les protocoles et offre une bonne modularité.

---

33 La délégation est implicite dans CLOS (avec *call-next-method*). Par contre dans des langages comme C++ ou Oberon ce n'est pas le cas. Pour une comparaison entre CLOS et SMALLTALK, on peut se référer à [Cointe 1993].

- usage restreint de la création dynamique de classes

Cette approche nous semble trop peu au point pour que l'on puisse la systématiser pour des développements importants. La mise au point devient difficile pour le développeur si l'ensemble des classes définies varie durant l'exécution d'un programme. Dans la même perspective, la modification dynamique de classes est à éviter. Par contre, on pourra considérer la création dynamique de classes à partir de classes fusionnantes[34]. Pour cela, il est nécessaire de disposer de l'héritage multiple.. En effet, cette technique évite de définir a priori une combinatoire lourde de classes dont seulement certaines pourraient être instanciées. Elle a été utilisée pour l'élaboration du gestionnaire d'objets de l'interface utilisateur PICASSO [Konstan & Rowe 1991].

3.3.2. Protocoles et Taxons

La définition de protocoles offre une approche satisfaisante à la réalisation de boîtes à outils génériques. Après avoir présenté les grands principes que doit respecter un protocole bien défini, nous introduisons la notion de stratification qui permet de segmenter un protocole en niveaux d'utilisation.

Interface logicielle et protocoles

Les expériences précédantes aidant, nous pouvons maintenant proposer une définition de ce qu'est un protocole :

Un protocole est la description abstraite des opérations qui peuvent être effectuées sur un ensemble d'objets.

Il s'agit de décrire aussi bien ce qu'un objet peut faire que les opérations qui peuvent lui être appliquées. Par contre, un protocole ne dit rien sur la façon dont ces opérations peuvent être réalisées. Plusieurs exemples de protocoles sont donnés dans [Keene 1989].

Les classes *collection* définies dans [Booch 1992] ou dans [Apple Computer Inc. 1992] comprennent les ensembles, les tableaux, les séquences, les tables, les listes, etc. qui répondent toutes au même protocole. L'approche adoptée pour le stockage des éléments, et donc les performances, peuvent varier d'une classe à l'autre, mais les opérations définies par le protocole (par exemple l'itération) sont possibles sur tous les types de *collection*.

Contrairement aux interfaces logicielles — les API — un protocole est indépendant des types des paramètres. Le type lié à un protocole est abstrait (le type *collection* par exemple). Tous les types qui héritent d'un type abstrait répondant à un protocole répondent eux-aussi à ce protocole.

Stratification

Afin de satisfaire à l'extensibilité, plusieurs niveaux d'utilisation peuvent être définis dans un protocole. Cette approche a été adoptée pour le MOP [Kiczales & al. 1991]. Le niveau "le plus externe" correspond à CLOS et est le plus accessible. Les niveaux "méta" augmentent en complexité mais également en expressivité. Il est, par exemple, possible de définir un nouveau type de métaclasse ou de métaméthode.

---

34 "Mixin Class"

Contrairement à une API, un protocole n'est pas nécessairement fixe. Il est possible et même souhaitable de dériver et spécialiser de nouveaux protocoles à partir d'un même protocole. Les classes sont les briques de base implémentant un protocole.

Taxon

Nous appellerons taxon[35] un ensemble d'objets (typiquement des classes) répondant à un protocole. Inversement, la définition d'un protocole entraîne la définition d'un taxon. Les objets qui se conforment à ce protocole sont les instances de ce taxon. Un taxon est analogue à une classe en ce qu'il fait référence à un ensemble d'objets qui sont des instances spécifiques d'une catégorie abstraite.

Exemple

L'exemple de protocole présenté ici est tiré de [Lafourcade & Sérasset 1993a].

Le Dictionnary Object Protocol (DOP) a été défini pour produire des outils dictionnaires basés sur des paradigmes de stockage de données différents, et facilement interchangeables.

Donnons un exemple :

```
(defclass person () (
   (lastname  :accessor lastname
              :initarg :lastname
              :initform nil)
   (firstname :accessor firstname
              :initarg :firstname
              :initform nil)) )

(defclass researcher (person) (
   (lab       :accessor lab
              :initarg :lab
              :initform nil)) )

(setf base (create-base "base" (  (lastname #'lastname)
                                  (lab #'lab)) ))

(insert-item base (make-instance 'researcher      :lastname "Sérasset"
                                                  :firstname "Gilles"
                                                  :lab "GETA"))
(insert-item base (make-instance 'researcher      :lastname "Blanchon"
                                                  :firstname "Hervé"
                                                  :lab "GETA"))
(insert-item base (make-instance 'researcher      :lastname "Gaschler"
                                                  :firstname "Jean"
                                                  :lab "XEROX"))
(insert-item base (make-instance 'researcher      :lastname "Lafourcade"
                                                  :firstname "Mathieu"
                                                  :lab "GETA"))
```

```
(mapcar #'firstname (find-items base 'lastname "Blanchon"))
> ("Hervé")

(mapcar #'lastname (find-items base 'lab "GETA"))
> ("Sérasset" "Blanchon" "Lafourcade")
```

```
(insert-item base (make-instance 'person :lastname "Dupont")
```

35 Ce terme est dû, semble-t-il, à Mikel Evins [Evins 1994].

```
(mapcar #'firstname (list-cluster base 'firstname))
> ("Sérasset" "Blanchon" "Gaschler" "Lafourcade" "Dupont"))
```

Un dictionnaire (on parle aussi de base) est un ensemble d'objets de différentes classes auxquels on peut accéder par différentes références. Pour chaque dictionnaire, on définit un ensemble de références (*firstname* et *lab*). Une référence est composée d'un nom et d'un fournisseur de référence (*'lastname* et la fonction #*'lastname*). Le fournisseur est une fonction qui, quand elle est appliquée à un élément quelconque, retourne une valeur (dans notre exemple, il s'agit simplement des accesseurs aux champs). Cette valeur est la clé d'accès associée à une référence d'un élément d'un dictionnaire.

Tous les éléments doivent retourner une valeur pour l'ensemble des fournissseurs du dictionnaire. Si un élément ne doit pas être stocké sous une référence particulière, le fournisseur de cette référence doit retourner la valeur NIL.

Pour utiliser le DOP, on crée un dictionnaire en indiquant l'ensemble des références. Un fournisseur (qui est défini lors de la création du dictionnaire) doit accepter une instance d'élément comme paramètre et retourner une chaîne de caractères ou NIL comme valeur.

Les messages définis par le protocole de DOP sont les suivants[36] :

Certains des messages se réalisent selon des fonctions ou des macro, car ils ne peuvent pas être associés à une instance (en particulier quand cette dernière n'existe pas encore).

`create-base (nom fournisseur* &key class)` `[macro]`

Cette macro permet de créer une nouvelle instance de base DOP nommée *nom* et disposant d'une liste de références. Une référence est de la forme (nom-de-clé fournisseur). Le mot clé *class* permet de préciser la classe de l'instance. La valeur retournée est l'instance de la base créée.

```
(setf base (create-base "base" (  (lastname #'lastname)
                                  (lab #'lab)) ))
```

`open-base (nom-de-fichier)` `[fonction]`

Cette fonction permet d'ouvrir une base dont la définition est contenue dans un fichier. La valeur retournée est l'instance de la base ouverte.

```
(open-base "ccl:base.dop")
```

Les messages qui suivent peuvent être réalisés avec des méthodes et donc être spécialisés sur différentes classes.

`delete-base (base)` `[méthode]`

Ferme la base si elle est ouverte, puis la détruit (avec ses fichiers associés s'il y en a).

```
(delete-base base)
```

36 Seuls les messages les plus représentatifs sont décrits ici.

`insert-item (base objet)` [méthode]

Insère un élément dans la base. Pour toutes les références de la base, applique le fournisseur correspondant sur l'objet et le stocke avec la clé correspondante.

```
(insert-item base (make-instance 'researcher      :lastname "Gaschler"
                                                  :firstname "Jean"
                                                  :lab "XEROX"))
```

`find-items (base reference clé)` [méthode]

Retourne les éléments sur lesquels le fournisseur de *référence* vaut *clé*.

```
(find-items base 'lastname "Blanchon")
```

Toutes les classes répondant à ce protocole sont des instances du taxon DOP. Les classes sont libres de répondre aux messages selon des implémentations spécifiques. De plus, il n'est pas nécessaire que les classes appartiennent à la même hiérarchie de classes.

Le DOP a été défini pour des bases organisées selon deux types de stockage. D'une part, une classe de bases persistantes (définie à partir d'un mécanisme d'objets persistants, WOOD [St Clair 1991]), d'autre part, une classe de bases définies à partir de tables de hachage. Le premier type de base est relativement économique en mémoire, mais peu efficace en temps d'accès. C'est l'inverse pour le second type de base.

En pratique, on remarque deux choses. D'abord, bien que pouvant implémenter le protocole de manière très différente, les deux classes peuvent partager la plus grande partie du code. C'est seulement certaines méthodes très spécifiques qui dépendent de leurs structures de données. Prenons la méthode *find-items* comme exemple :

```
(defmethod find-items (base keyform reference)
   (let ((table (get-item-key-table base reference)))
      (get-item base table keyform))
 )
```

La recherche des éléments se fait en deux étapes. On sélectionne d'abord la *table* correspondant à la référence, puis on recherche dans cette table l'élément dont la clé est *keyform*.

On voit que cette fonction est tout à fait générale et définit un sous-protocole composé des messages *get-item-key-table* et *get-item*. Ces deux messages sont réalisés différemment selon les deux types de base. En regroupant ces deux classes sous une même superclasse *dop-base* (figure 3.6), on factorise le code commun.

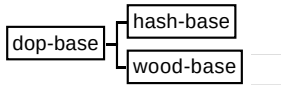


Figure 3.6 : La classe abstraite dop-base.

Cette classe a deux sous-classes implémentant de manières différentes une base DOP.

Une classe peut être une instance du taxon défini par le protocole DOP sans pour autant être une instance des sous-protocoles de DOP.

3.3.3. Apports du génie linguistique au génie logiciel

Le génie linguistique profite des avancées du génie logiciel, mais cet échange peut également se faire dans l'autre sens. En effet, d'une part, le génie logiciel peut tirer parti des technologies du TALN et d'autre part les techniques de génie logiciel définies pour le TALN peuvent être appliquées à d'autres domaines. C'est le cas des éditeurs multidialectes et de l'architecture de tableau blanc.

Editeur multi-dialectes et Traduction Automatique par pivot

L'expérience de la ré-ingéniérie de LT nous a amené à définir un noyau fonctionnel extensible pour le langage que nous avons présenté. Ce noyau, basé sur CLOS, est syntaxiquement inacceptable par des développeurs linguistes. De plus, le noyau est directement ouvert sur LISP, ce qui est satisfaisant pour des informaticiens mais problématique pour des linguistes (voir §1.1.2.). Des syntaxes fermées plus "conviviales" doivent et peuvent être fournies.

Dans [Lafourcade 1994a], nous avons présenté une technologie de Traduction Automatique appliquée à un éditeur de langage de programmation. Cette application a été développée pour le LSPL LT4[37] (présenté au §3.1.).

L'interface de l'environnement de développement de LT4 dispose d'un éditeur implémentant une syntaxe externe qui répond mieux aux attentes des linguistes. La partie inférieure de l'éditeur offre un menu dont l'élément actif est étiqueté "English" (voir figure 3.7 a). L'utilisateur peut sélectionner dans ce menu un dialecte parmi une liste de dialectes (actuellement : English, French, German, Arabic, Malay, …). La sélection d'un élément de menu entraîne la traduction immédiate du programme vers le dialecte choisi (par exemple, le français, voir figure 3.7 b).

[37] Ce développement était motivé d'une part pour valider cette idée et d'autre part pour fournir aux linguistes un outil lié à un environnement utilisable.

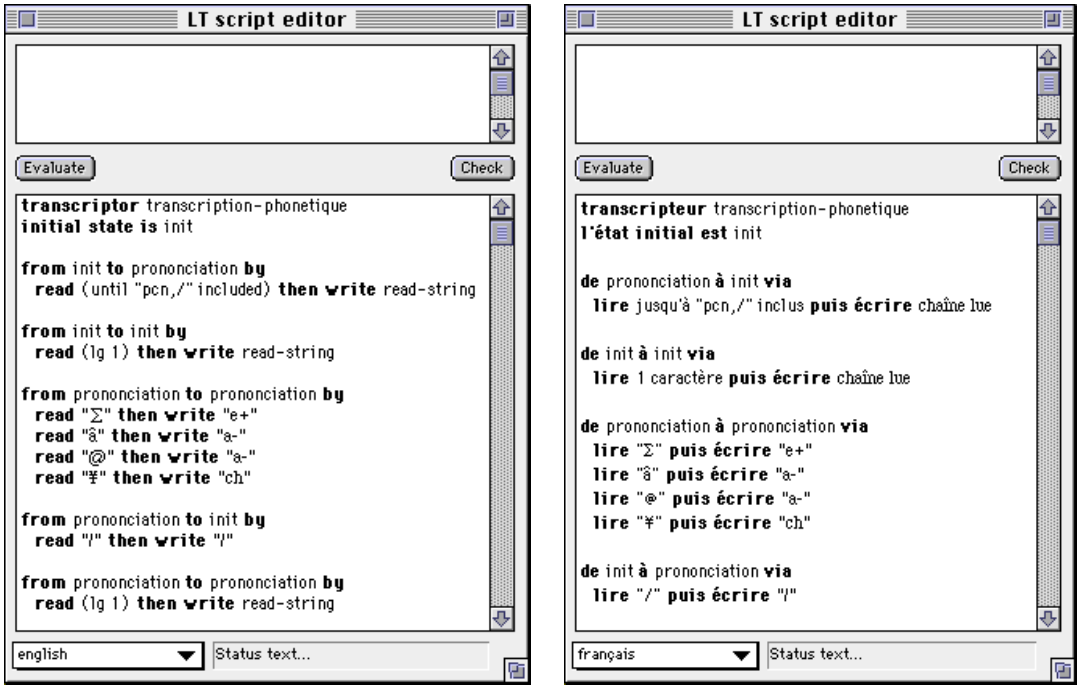


Figure 3.7 : Exemples de dialectes LT.

Le même programme LT peut être édité dans plusieurs dialectes différents, ici anglais et français. L'éditeur multi-dialectes permet de passer directement d'un dialecte à l'autre.

Les dialectes ne sont pas nécessairement liés à une langue naturelle. Ils peuvent présenter des syntaxes plus concises ou abstraites (on écrit plus vite, mais on perd en lisibilité). Les expériences menées avec ce type d'éditeur et le LSPL LT auprès de linguistes nous ont confirmé le bien-fondé de cette approche du point de vue de l'utilisabilité [Gaschler & al. 1994].

La traduction effectuée va bien au-delà du simple remplacement de mot-clés, qui se révèle rapidement insuffisant lorsque l'on cherche à élaborer des dialectes proches de langues naturelles (variation considérable de l'ordre des mots). Le nombre de mots-clés (certains peuvent être optionnels) est variable.

Nous nous sommes inspiré de la traduction par pivot (figure 3.8). Le pivot choisi est la forme noyau de LT4. Les dialectes que nous avons développés correspondent strictement à la forme noyau et ne permettent pas l'accès à LISP. C'est seulement à cette condition que le langage LT4 peut être figé et réimplémenté dans d'autres environnements avec d'autres langages de programmation. L'ouverture sur LISP, limiterait ipso facto la portabilité au langage LISP lui-même.

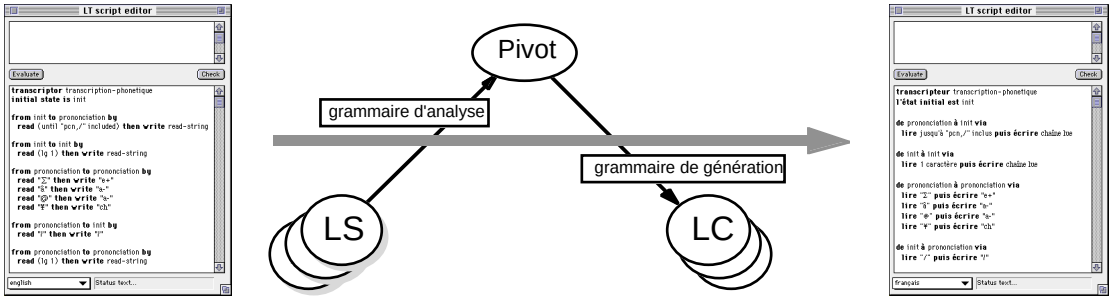


Figure 3.8 : Traduction par pivot entre deux dialectes.

L'introduction de la notion de dialecte perturbe le shéma de hiérarchie de langages présenté au §3.1.2. En effet, nous ne disposons plus d'un seul langage utilisateur mais de plusieurs[38].

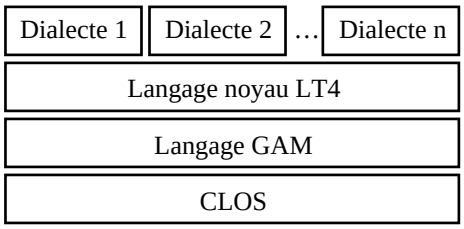


Figure 3.9 : Introduction des dialectes dans la hiérarchie de langages de LT4.

Application du tableau blanc à d'autres domaines

L'architecture de tableau noir reste confinée à de petites parties de l'Intelligence Artificielle. Par contre, l'approche par règles, initialement créée pour le TALN, s'est étendue à l'Intelligence Artificielle (avec les systèmes experts) et au génie logiciel (avec l'écriture de compilateurs).

L'architecture de tableau blanc pourrait trouver des applications dans d'autres domaines que le TALN. Cette architecture pourrait constituer une solution pour des problèmes où :

- des logiciels de taille importante doivent être intégrés ;
- plusieurs modes de communications sont envisagés (directe, par réseau hétérogène, par transfert de fichers) ;
- le temps réél n'est pas crucial.

Protocoles

Le génie logiciel se penche activement sur la conception de protocoles, que se soit pour la réalisation d'interfaces Homme-Machine ou pour la définition de langages de programmation généraux [Kiczales & al. 1991]. Les protocoles sont un moyen efficace d'obtenir la recherche de la généricité et l'extensibilité d'une bibliothèque de classes [Meyer 1990]. Cependant, leur définition reste délicate. Nous reviendrons sur cette question dans les deux parties suivantes.

---

38 Il n'y a pas un langage utilisateur particulier. Toutes les syntaxes externes sont des dialectes du langage noyau.

Conclusion de la première partie

La réalisation concrète d'un système basé sur l'architecture de tableau blanc implique la réalisation préalable d'une bibliothèque de classes appropriée. Il est intéressant de doter ces classes des bonnes caractéristiques d'intégrabilité, de généricité et d'extensibilité. Bien qu'adaptée à des composants de grande taille, cette architecture a avantage à être appliquée récursivement. En guise d'illustration, nous avons donné un première analyse en ces termes de la maquette LIDIA-1.

La généricité et l'extensibilité peuvent s'exprimer à l'aide de protocoles. Il ne s'agit plus d'articuler une analyse autour de classes, mais autour de comportements. Les classes appartenant à un protocole doivent répondre à un certain nombre de messages. Ces messages sont implémentés spécifiquement par chaque classe à l'aide de méthodes. La notion de taxon permet de regrouper les classes répondant à un protocole. Cette relation est indépendante de la relation d'héritage.

Un LSPL générique doit être fondé sur une stratification des niveaux de définition. Il est ainsi possible d'utiliser ce LSPL comme langage cible d'une compilation. LT est compilé dans notre langage d'automates (GAM), offrant ainsi des facilités de mise au point. La spécialisation permet également d'ajuster les comportements de ce langage. Un LSPL extensible doit fournir des protocoles permettant de rajouter des fonctionnalités. L'expérience de LT en a été un première illustration.

Bien que largement illustrées, la généricité et l'extensibilité requièrent une approche logicielle générale, qui définit les limites de ces bonnes propriétés. Il est de plus nécessaire d'avoir une approche qui permet une application généralisée à tous les LSPL ou aux langages de programmation généraux.

Seconde partie

Généricité et systèmes de décorations

Introduction de la seconde partie

Dans le domaine du TALN, des théories linguistiques définissent à la fois des structures de données (graphes, arbres, chaînes, structures attribués…) et des formalismes ou représentations linguistiques (LFG, GPSG, HPSG,…). Pour implémenter des langages qui permettent d'intégrer ces représentations linguistiques dans le traitement automatique, on utilise soit des systèmes très spécifiques, soit des langages généraux de représentation de la connaissance. Certains langages généraux se basent sur des approches hybrides comme LIFE [Aït-Kaci & Lincoln 1989] qui intègre les paradigmes de la programmation par objets, logique et fonctionnelle.

Les systèmes spécifiques sont coûteux à développer et pourraient tirer avantage à être produits à partir de systèmes plus généraux. Les approches hybrides ne permettent généralement pas d'ajouter, après coup, de nouveaux paradigmes. Il nous semble qu'une approche "tolérante"[39], non seulement ouverte à de nombreux formalismes existants, mais également extensible à de nouveaux formalismes, présente l'intérêt de donner aux développeurs linguistes une plus grande marge de manœuvre. Cette même idée a été à l'origine de travaux sur les bases lexicales [Sérasset & Blanc 1993].

Le modèle LEAF nous invite à découpler, dans un LSPL, la partie moteur (le contrôle) de la partie décoration (les structures de données manipulées)[40]. Cette perspective nous amène à concevoir un langage de représentation linguistique utilisable pour diverses applications, et surtout intégrable à des moteurs variés.

Nous avons défini un langage de représentation, DÉCOR, basé sur un nouveau type d'objet : la structure de décoration. DÉCOR est un langage noyau ouvert et

39 Les termes d'"œcuménique" et "théorie accueillante" ont été proposés dans [Sérasset 1994a].

40 Une idée similaire présentée par [Barnett, Knight, Mani & al. 1990] consiste aussi à séparer différents types de connaissances (connaissance du monde et connaissances linguistiques). Le morcellement des différentes connaissances, tout comme la séparation nette dans un LSPL entre le contrôle et les données, cherche à concrétiser une même idée : la modularité.

générique, destiné à servir de base à l'implémentation d'autres langages plus spécifiques. Il est possible de l'utiliser lui-même, mais le développeur linguiste souhaitera une syntaxe externe cachant la notation LISP. La réalisation d'une telle syntaxe peut, selon nous, bénéficier d'une technologie similaire à celle des éditeurs multi-dialectes (présentée à la fin du chapitre 3).

Nous montrons comment nous avons essayé d'introduire la généricité dans l'implémentation de DÉCOR. Elle se concrétise par un protocole externe, et par des protocoles internes qui permettent des spécialisations de granularités différentes. La recherche de possibilités d'extension est délicate, car elle débouche sur des protocoles qui amènent souvent à redéfinir certains aspects d'un langage (qui avaient été auparavant supposés figés), et rend souvent l'implémentation plus complexe.

Chapitre 4

Langages de représentation existants

La représentation de la connaissance, et son utilisation pour l'inférence et le raisonnement, est généralement une des parties les plus importantes des applications de l'Intelligence Artificielle. Une des conséquences est qu'un grand nombre de formalismes, d'outils ou d'environnements ont vu le jour, comme par exemple KEE, ART, KL1. Le TALN a évidemment tiré parti de ces avancées et il est intéressant de brosser un tableau des formalismes qui ont été utilisés. Dans la perspective de la programmation par objets, nous nous concentrons sur les formalismes de type attribut-valeur. Conformément à la typologie des langages dressée dans la première partie, nous parlerons plutôt de langages de représentation de la connaissance à base d'objets.

Des structures plates utilisées dans ARIANE ou METAL aux prototypes et à leurs dérivés, comme les psi-termes et structures de traits typés, les approches sont multiples et la terminologie complexe. En général, une variable est un symbole (ou une chaîne de caractères) associé à une liste de valeurs. Ces informations peuvent se présenter sous forme de traits. Les facilités de manipulations des objets varient grandement d'un langage à l'autre. Par exemple, les opérations sur les symboles ne sont en général pas bien définies, et Colmerauer a introduit la notion d'étiquettes pour remédier à ces problèmes. Dans [Johnson 1991], on peut trouver une présentation des structures de traits complexes associée à l'utilisation de clauses logiques afin de séparer les contraintes que les structures doivent vérifier. L'intégration à la logique a également été l'objet des travaux qui ont débouché sur LIFE (après son ancêtre LOGIN [Aït-Kaci & Nasr 1986]) et TFS ([Zajac 1988] et [Emele & Zajac 1990]). Dans [Ahrenberg, Jönsson & Dahlbäck 1991] la connaissance linguistique est exprimée à l'aide d'un format de type PATR-II augmenté de la disjonction et de la négation. De plus, un travail important a été effectué sur le "sucre syntaxique" afin de rendre le système utilisable par des

linguistes. L'utilisation des structures de traits ne se limite pas au TALN ou à la représentation de la connaissance. Ainsi, les Frames de Minsky se situaient, à l'origine, dans le cadre de travaux sur la perception dans les systèmes de vision, mais leur usage a été étendu. Dans [Myers, Giuse & Zanden 1992], les prototypes sont à la base d'un système déclaratif pour la spécification d'interfaces utilisateur.

4.1. Ensemble de traits

La forme la plus simple des formalismes de type attribut-valeur est l'ensemble de traits[41]. Selon les formalismes, la valeur attribuée à un trait peut être de plusieurs natures (symbole, nombre, chaîne de caractères, …). Un trait simple peut être réduit à un symbole (présent ou absent, donc booléen) ou être une valeur atomique (symbole, chaînes, entier, booléen). Un trait complexe peut avoir une liste de valeurs atomiques.

4.1.1. Traits simples

Un trait est dit "simple" s'il est réduit à un symbole non valué ou n'a qu'une valeur atomique.

REZO

Les objets manipulables par un programme REZO [Stewart 1978] sont des arbres où chaque nœud comporte une étiquette et un ensemble de traits. La structure de base de REZO a fait l'objet d'une extension par rapport au modèle original que sont les ATN. Les chaînes de mots ont été remplacées par l'utilisation d'un graphe orienté, sans circuit semblable à ce que l'on peut trouver dans [Kay 1973] ou dans [Colmerauer 1970]. L'avantage énorme sur les ATN est la composabilité : un programme REZO est, en effet, un transducteur d'un graphe-Q dans un autre et non plus d'une chaîne de caractères dans une autre.

Six types de valeurs différentes sont manipulés [Stewart 1975] : booléen, nombre, étiquette, ensemble, arbre et liste. Une étiquette peut commencer par un caractère quelconque mais doit ensuite être formée de chiffres et de lettres. Un ensemble est une suite de traits séparés par des virgules.

```
[+HUM, *feminin]
[T1]
```

Un nœud est constitué d'une étiquette et d'un ensemble optionnel de traits. Un arbre est un nœud dominant une liste éventuellement vide d'arbres. Par exemple :

```
PHRASE [AFFIRMATIF]  (SN [NOMPROPRE, HUMAIN] (MAX),
                      SV(V(MANGE), SN[CONCRET, PLURIEL]
                                    (DES, BANANES)))

+(10,*(-30,45))
```

-30 est une étiquette (il ne s'agit pas de - suivi de 30). * est aussi une étiquette (dont le nœud domine la liste d'arbres composée de -30 et 45).

41 La différence que nous faisons ici entre ensemble et liste est qu'une liste autorise la répétition d'un même élément, ce qui n'est pas le cas dans un ensemble. Nous n'entrerons pas dans les considérations liées aux notions d'ordre.

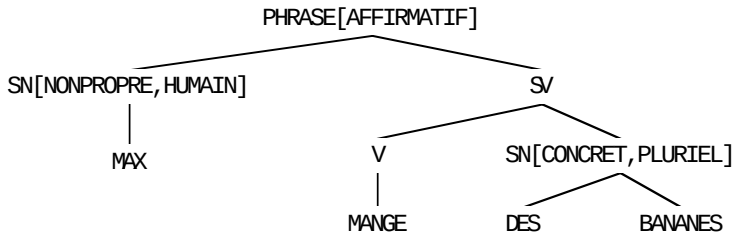


Figure 4.1 : Exemple d'arbre sous forme graphique.

Les attributs sont limités à des étiquettes atomiques. Les seuls opérateurs disponibles pour les étiquettes sont l'égalité et l'inégalité. Un ensemble (une liste de traits) dispose des opérations classiques (intersection, union, inclusion, …).

Phonologie

Les traits utilisés en phonologie sont spécifiés positivement ou négativement. Le phonème [b] peut être caractérisé par les traits :

```
[+ labial] [+ voisé] [- nasal]
```

Il s'agit de traits binaires qui permettent de regrouper des objets linguistiques sur la base de propriétés communes. Comme un attribut peut être positif, négatif ou non spécifié, on obtient une logique à trois valeurs.

GPSG

La théorie de la grammaire syntagmatique généralisée (Generalized Phrase Structure Grammar) a été élaborée durant les années quatre-vingt par [Gazdar, Klein, Pullum & al. 1985]. L'objectif visait à définir un modèle syntaxique non-tranformationel[42]. Ce modèle est syntagmatique en ce qu'il est basé sur des règles de réécriture de grammaires hors-contexte. Il est généralisé car il permet la définition de règles sous-spécifiées. Par exemple, l'ordre des mots est séparé et peut être appliqué pour toutes les configurations de la grammaire.

GPSG considère les catégories syntaxiques comme des ensembles de traits (en LFG, il s'agit de structures de traits)[43]. Les catégories peuvent donc contenir un nombre variable de traits. Plus le nombre de traits est élevé, plus la catégorie est spécifiée (et inversement). L'ensemble vide est la catégorie la moins spécifiée qui unifie toutes les autres. L'écriture des règles s'en trouve simplifiée, car il suffit de mentionner seulement les traits pertinents. Plus une règle est générale, moins les catégories qui la composent sont spécifiées.

Les traits se distinguent par leur type de valeurs (atomique ou de catégorie) et par le mode de propagation lié au type de trait (trait de tête, de pied ou de contrôle/accord). Le mode de propagation est également appelé "principe d'instanciation".

- Traits utilisés

Le système de traits présenté ici est issu de [Gazdar & al. 1985]. D'autres systèmes basés sur GPSG (comme ALVEY de [Grover, Caroll & Briscoe 1993]), ne définissent pas

---

42 Un second objectif consistait à montrer qu'une grammaire hors-contexte suffisait comme description d'une langue naturelle. Depuis, certains phénomènes linguistiques dépassant la puissance de ces types de grammaires ont été donnés comme exemples [Abeillé 1993].

43 Structure et ensemble sont en fait équivalents. Une structure est nommée contrairement à un ensemble qui, a priori, ne l'est pas. Il est seulement désigné par sa valeur.

strictement les mêmes traits ou les mêmes catégories. Les principe de bonne formation peuvent aussi être légèrement différents.

La plupart des traits ont des valeurs atomiques. Les principaux attributs sont résumés dans le tableau suivant :

Attribut	**Valeurs**	**Commentaires**
N	+ ou - (booléen)	Trait lexical (barre = 0)
V	booléen	Trait lexical
Barre	0, 1 ou 2	Niveau de barre (théorie X-barre) 0 = mot, 1 = syntagme intermédiaire, 2 = syntagme maximal
Suj	booléen	Sujet
Sous-cat	numérique	Valence des prédicats locaux
Prepform	lexicale	Préposition
Comform	lexical	Complémenteur
Mode	lexical	Infinitif, subjonctif, etc.

Une valeur catégorielle est un ensemble prédéfini de traits. Par exemple, nous avons Verbe = {N = -, V = +}. À la différence des structures de traits, les catégories ne sont pas des objets informatiques qui associent un symbole (par exemple verbe) à un ensemble de traits. Il s'agit seulement de l'ensemble de traits dont l'interprétation est la catégorie verbe. Dans ALVEY ([Caroll & al. 1991] et [Grover & al. 1993]), la notion d'*alias* a été introduite comme abréviation pour désigner des catégories.

Des contraintes peuvent être appliquées sur les traits de façon à rendre incompatibles entre eux certains traits ou, inversement, à les lier. Par exemple, un adverbe ne peut pas être une information sur le nombre (singulier ou pluriel) ({Num} ⇒ {N = +, V = -}). En français, le complémenteur d'un groupe peut être "de" ou "à" si et seulement si ce groupe est à l'infinitif ({Compform = de/à} ⇒ {Mode = inf}). La présence d'un trait ou d'un attribut peut être interdite par l'utilisation de la négation (notée ~). Par exemple, la sous-catégorisation n'est pas autorisée sur les catégories non lexicales ({Barre 0} ⇒ ~{Sous-cat}).

Les catégories majeures sont celles définies en termes des attributs N, V et Barre.

Catégorie	**Traits**	**Contraintes**
Verbes	{V = +, N = -, Barre = 0}	Mode
Groupes verbaux	{V = +, N = -, Barre = 2, Suj = -}	Mode, ~Sous-cat
Phrase	{V = +, N = -, Barre = 2, Suj = +}	Mode, Compform, ~Sous-cat
Noms	{V = -, N = +, Barre = 0}	~Mode
Pronoms, GN	{V = -, N = +, Barre = 2}	~Sous-cat
Préposition	{V = -, N = -, Barre = 0}	Prepform
Adverbes, Groupes prépositionels	{V = -, N = -, Barre = 2}	Prepform, ~Sous-cat
Ajectifs	{V = +, N = +, Barre = 0}	~Mode
Groupes adjectivaux	{V = +, N = +, Barre = 2}	~Sous-cat

Complémenteurs	Compform	~Barre, ~Sous-cat, ~V, ~N
Déterminants	{Det = +}	~Barre, ~Sous-cat, ~V, ~N

Nous rappelons que les catégories n'existent pas en tant que telles mais seulement comme combinaisons de traits.

Un trait peut être instancié avec une disjonction de valeurs (notée par /). Par exemple, la sous-catégorisation est définie à l'aide du trait numérique Sous-cat. Ce trait permet d'affiner la règle à appliquer sur le syntagme courant. Les verbes intransitifs ont le trait Sous-cat à 1, et les verbes transitifs à 2. Si un verbe correspond aux deux catégories (comme regarder), le trait peut prendre la valeur 1/2. Dans l'implémentation, ALVEY Sous-cat prend ses valeurs dans une liste de catégories.

- Principes d'instanciation

Plusieurs principes d'instanciation permettent de donner des valeurs aux traits ou de vérifier leur cohérence. Une relation de dépendance spécifie que la valeur d'un trait d'une catégorie appartenant à un arbre local peut dépendre de la valeur d'une autre catégorie d'un autre arbre local. La relation de dépendance peut être locale ou globale selon que les catégories appartiennent ou non au même arbre local.

Le principe de *trait de tête* impose une relation d'identité entre certains traits de la catégorie dominante (celle en partie gauche des règles) et les traits correspondant d'une catégorie dominée (en partie droite des règles) privilégiée (on l'appelle TETE).

Le principe du *trait de pied* consiste à partager certains des traits de la catégorie dominée avec les traits correspondants de n'importe quelle catégorie dominée.

Le principe d'*accord /contrôle* permet de régler les phénomènes d'accord entre deux catégories (par exemple verbe et sujet) en forçant le partage d'un trait particulier (ACCORD) entre deux catégories sœurs.

La spécification de *traits par défaut* permet de représenter implicitement une partie des traits.

La théorie GPSG définit des structures de données à partir desquelles on peut travailler mais elle impose une législation linguistique rigoureuse. Les principes d'instanciation catégorisent les traits en mode de propagation (au sens des grammaires attribuées) différents.

4.1.2. Traits complexes

Un attribut complexe peut avoir comme valeur, une valeur atomique ou une liste de valeurs.

ARIANE

Les décorations utilisées par les LSPL d'ARIANE sont des ensembles d'attributs typés (appelés *jeux de variables* dans le jargon local). Ces attributs correspondent à des catégories linguistiques (mode, cas, genre, nombre, degré, …) formalisées. Par exemple :

```
KMS == (VB, NM, ADJ, ADIP, DR, COP, PC, NA)
```

L'interprétation de l'attribut KMS ci-dessus est la suivante. La catégorie morpho-syntaxique (KMS) peut prendre ses valeurs de manière non exclusive parmi les

valeurs : verbe, substantif, adjectif, adjoint invariant, déicteur, lexème d'hypotaxe ou de parataxe, ponctuation ou forme non alphabétique. Par exemple, KMS('pauvre') = (NM, ADJ).

Ou encore, nous avons :

```
SUBCOP == (COOR, CJS, PRP)
```

La classe des lexèmes d'hypotaxe ou de parataxe[44] peut être raffinée de manière non-exclusive, en conjonction de coordination, en conjonction de subordination, et en préposition.

Les attributs qui correspondent respectivement à la catégorie syntaxique, à la classe syntagmatique, au nombre et la fonction syntaxique pourraient être définis comme suit :

```
CAT == (ART, NOM, VB, ADJ)
K == (GN, GP, GV)
NB == (S, P)
FS == (DES, EPIT, GOV, SUJ, OBJ)
```

Un attribut est un ensemble de valeurs élémentaires. Ces valeurs peuvent être exclusives ou non exclusives (mode par défaut). Un attribut exclusif ne peut prendre qu'une seule valeur à la fois parmi les valeurs possibles. Un attribut non-exclusif peut prendre comme valeur un sous ensemble des valeurs possibles. Les attributs sont regroupés selon leur mode.

Un *format* est une décoration constante et nommée. Par exemple :

```
FSCOOR := KMS(COOP), SUBCOP(COOR)
```

Le format FSCOOR indique que l'attribut KMS a la valeur COOP et que l'attribut SUBCOP a la valeur COOR.

Une *unité lexicale* est une valeur de l'attribut spécial UL (prédéfini et exclusif). Les unités lexicales sont introduites dans les dictionnaires et les grammaires. Cet attribut est de type chaîne et a quelques valeurs prédéfinies. Il n'a pas d'autres opérateurs que l'égalité et la différence (notés = &) et également de l'ouverture.

```
UL == (UL0 alias '', 'ULTXT', 'ULFRA', … pot 3200)
```

Une *structure arborescente étiquetée* est un arbre où chaque nœud porte des attributs (une décoration). Par exemple :

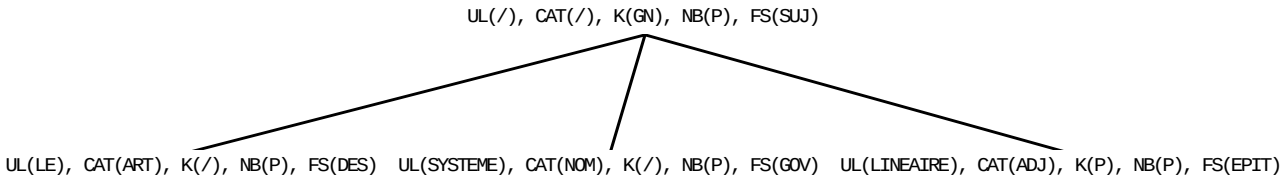


Figure 4.2 : Exemple d'arborescence étiquetée dans ARIANE.

La forme de sortie qui suit correspond à l'analyse ambiguë de la phrase «le capitaine a rapporté un vase de Chine»déjà présentée au §2.2.4.

---

44 L'hypotaxe correspond à la subordination. La parataxe correspond à la coordination.

```
1: 'ULTXT' (2: 'ULFRA' (3: 'ULSOL' (4: 'PHVB' (5: 'DGN' (6: 'LE', 7: 'CAPITAINE'), 8: 'NV' (9:
'AVOIR', 10: 'RAPPORTER'), 11: 'DGN' (12: 'UN', 13: 'VASE/LE', 14: 'GP' (15: 'DE', 16: '*CHINE')),
17: '.')), 18: 'ULSOL' (19: 'PHVB' (20: 'DGN' (21: 'LE', 22: 'CAPITAINE'), 23: 'NV' (24: 'AVOIR',
25: 'RAPPORTER'), 26: 'DGN' (27: 'UN', 28: 'VASE/LE'), 29: 'GP' (30: 'DE', 31: '*CHINE'), 32:
'.'))))
```

```
   1 '': UL('ULTXT').
2 '': UL('ULFRA').
3 '': UL('ULSOL'), MODIF(NON).
4 'A': UL('PHVB'), LINKS(OUI), RECHTS(OUI),
ENONCP(DECL), K(PHVB), PHASE(ACC), SUBV(VF),
VALET(Q), VOIX(ACT), ARGS(A0,A1), CAT(V),
MT(IPR), NBR(SING), PERS(3).
5 'CAPITAINE': UL('DGN'), FS(SUJ), K(GN),
RL(ARG0), SUBN(NC), VALET(N), CAT(N,D),
GNR(MAS), NBR(SING), PERS(3).
6 '*LE': UL('LE'), FS(DES), CAT(D), GNR(MAS),
NBR(SING), TYPOG(PCAP).
7 'CAPITAINE': UL('CAPITAINE'), FS(GOV),
SUBN(NC), CAT(N), GNR(MAS), NBR(SING), PERS(3),
SENS(1,2,3).
8 'A': UL('NV'), LINKS(OUI), RECHTS(OUI),
ENONCP(DECL), K(PHVB), PHASE(ACC), SUBV(VF),
VALET(Q), VOIX(ACT), ARGS(A0,A1), CAT(V),
MT(IPR), NBR(SING), PERS(3).
9 'A': UL('AVOIR'), FS(AUX), SUBV(VF), CAT(V),
MT(IPR), NBR(SING), PERS(3), SENS(1).
10 'RAPPORTE!1': UL('RAPPORTER'), FS(GOV),
SUBV(PPA), CAT(V), GNR(MAS), NBR(SING), PERS(3),
SENS(1,2,3).
11 'VASE': UL('DGN'), RECHTS(OUI), FS(OBJ),
K(GN), RL(ARG1), SUBN(NC), VALET(N), CAT(N,D),
GNR(MAS), NBR(SING), PERS(3).
12 'UN': UL('UN'), FS(DES), CAT(D), GNR(MAS),
NBR(SING).
13 'VASE': UL('VASE/LE'), FS(GOV), SUBN(NC),
CAT(N), GNR(MAS), NBR(SING),
SENS(1).
14 '*CHINE': UL('GP'), FS(COMP), K(GN),
RS(UNDE), SUBN(NP), VALET(DEN), CAT(N,S),
GNR(FEM), NBR(SING), PERS(3).
15 'DE': UL('DE'), FS(REG), CAT(S).
16 '*CHINE': UL('*CHINE'), FS(GOV), SUBN(NP),
CAT(N), GNR(FEM), NBR(SING), SENS(1).
17 '.': UL('.'), CAT(P), PONC(DR).
18 '': UL('ULSOL'), MODIF(NON).
19 'A': UL('PHVB'), LINKS(OUI), RECHTS(OUI),
ENONCP(DECL), K(PHVB), PHASE(ACC), SUBV(VF),
VALET(Q), VOIX(ACT), ARGS(A0,A1), CAT(V),
MT(IPR), NBR(SING), PERS(3).
20 'CAPITAINE': UL('DGN'), FS(SUJ), K(GN),
RL(ARG0), SUBN(NC), VALET(N), CAT(N,D),
GNR(MAS), NBR(SING), PERS(3).
21 '*LE': UL('LE'), FS(DES), CAT(D), GNR(MAS),
NBR(SING), TYPOG(PCAP).
22 'CAPITAINE': UL('CAPITAINE'), FS(GOV),
SUBN(NC), CAT(N), GNR(MAS), NBR(SING), PERS(3),
SENS(1,2,3).
23 'A': UL('NV'), LINKS(OUI), RECHTS(OUI),
ENONCP(DECL), K(PHVB), PHASE(ACC), SUBV(VF),
VALET(Q), VOIX(ACT), ARGS(A0,A1), CAT(V),
MT(IPR), NBR(SING), PERS(3).
24 'A': UL('AVOIR'), FS(AUX), SUBV(VF), CAT(V),
MT(IPR), NBR(SING), PERS(3), SENS(1).
25 'RAPPORTE!1': UL('RAPPORTER'), FS(GOV),
SUBV(PPA), CAT(V), GNR(MAS), NBR(SING), PERS(3),
SENS(1,2,3).
26 'VASE': UL('DGN'), FS(OBJ), K(GN), RL(ARG1),
SUBN(NC), VALET(N), CAT(N,D), GNR(MAS),
NBR(SING), PERS(3).
27 'UN': UL('UN'), FS(DES), CAT(D), GNR(MAS),
NBR(SING).
28 'VASE': UL('VASE/LE'), FS(GOV), SUBN(NC),
CAT(N), GNR(MAS), NBR(SING), SENS(1).
29 '*CHINE': UL('GP'), FS(CIRC), K(GN),
RS(UNDE), SUBN(NP), VALET(DEN), CAT(N,S),
GNR(FEM), NBR(SING), PERS(3).
30 'DE': UL('DE'), FS(REG), CAT(S).
31 '*CHINE': UL('*CHINE'), FS(GOV), SUBN(NP),
CAT(N), GNR(FEM), NBR(SING), SENS(1).
32 '.': UL('.'), CAT(P), PONC(DR).
```

Figure 4.3 : Forme de sortie d'un arbre d'analyse ambiguë dans ARIANE.

La notion d'*attribut général* (ou hiérarchique) permet d'étendre les structures précédentes (qui sont considérées comme élémentaires). Un attribut général définit non plus une liste de noms mais une liste d'arbres. La racine peut dominer une liste de feuilles (des noms) ou une liste d'arbres mais pas une liste mixte. On peut avoir par exemple :

```
VEG1 == (VEX1A(1, 2), VEX1B(VEX1C(A, B) VEX1D(C, D))))
```

les attributs généraux sont VEX1 et VEX1B et les attributs élémentaires sont VEX1A, VEX1C et VEX1D.

```
VEX1 == (1, VEX1A(A, B))
```

n'est pas une déclaration acceptable (la liste est mixte).

Ce type de description est général, mais certains LSPL n'acceptent qu'une restriction de ce système de décorations. Par exemple, ATEF n'inclut pas les attributs arithmétiques et généraux. De plus certains attributs sont obligatoires (par exemple, l'attribut DICT qui reflète les dictionnaires courants ouverts, mais qui ne fait pas partie du jeu de décorations).

Les opérations sur les variables incluent entre autres l'intersection (-I-), l'union (-U-), l'égalité (-E-), la non égalité (-NE-). On peut avoir par exemple :

Déclaration des attributs :

```
V5 == (A, B, C)
V1 == (V2(1, 2, 3), V3(A, B, C, D,), V4(1, 2, 3))
```

Une expression d'attributs

```
V1(X) -I- ((V2(Y) -I- (I -U- 3)) -U- (V3(Z) -I- (A -U- D))) -NE- V10
```

dont l'interprétation est :

```
si     V1(X) = V2 (1, 2), V3(A, B, C), V4(4,3)
  et   V2(Y) = V2(1, 2, 3)
  et   V3(Z) = V3 (A, B, C)
alors
       V2(Y) -I- (1 -U- 3) = V2(1, 3)
  et   V3(Z) -I- (A -U- D) = V3(A)
  et   (V2(Y) -I- (I -U- 3)) -U- (V3(Z) -I- (A -U- D))
    = V2(1 ,3), V3(A)
```

le résultat est donc :

```
(V2(1), V3(A), V4(1, 3) ≠ V10
```

Le système de décorations utilisé dans ARIANE permet aux linguistes de représenter facilement des objets linguistiques. Cependant, l'implémentation actuelle limite le nombre d'attributs à 256, ce qui oblige les linguistes à de nombreuses acrobaties. Par ailleurs, le besoin en traits plus structurés (avec des sous-traits, par exemple) se fait cruellement sentir, surtout pour des représentations de phénomènes comme la sous-catégorisation. D'autres systèmes de décorations rencontrent des problèmes analogues.

METAL

Nous avons déjà présenté deux aspects du système METAL dans les §1.1.2. et §2.2.2.

Le système de décorations linguistiques utilisés dans METAL (dans les dictionnaires monolingues et bilingues) définit les types de valeurs suivants :

- entier un nombre entre $-2^{28}+1$ et $2^{28}-1$;
- chaîne une séquence de caractères entre guillemets ;
- symbole un symbole LISP ;
- booléen les symboles T ou NIL ;
- ensemble une séquence, non ordonnée et sans répétition d'entiers, de chaînes et/ou de symboles ;
- liste une séquence ordonnée d'entiers, de symboles, de chaînes ou de listes.

La forme *defeat* (pour "define feature") permet de définir les différents attributs. Un attribut est composé d'un nom, d'un type d'attribut et d'un ensemble de valeurs possibles (son extension).

```
(defeat CA  :set
            :value (N A D G)
            :documentation "Case: Nominative, Accusative, Dative,
                            Genitive.")
```

Les mots-clés de la fonction *defeat* permettent de définir des options qui dépendent du type. Les types d'attribut possibles sont : *:boolean, :string, :integer, :list, :set* et *:member.* Une liste correspond à une liste arbitraire LISP. L'ensemble peut contenir n'importe quel sous-ensemble des valeurs définies par le mot-clé :values. *:member* n'autorise comme valeur qu'un des éléments défini par le mot-clé *:values*.

D'autres mots-clés sont utilisés :

:range
cette option peut être utilisée en conjonction avec le type *:integer* afin de définir la borne minimum et maximun d'une valeur numérique ;

:type
cette option peut être utilisée en conjonction avec le type *:list*. L'unique argument est une liste pouvant contenir plusieurs symboles parmi *:symbol, :string, :integer, :list ou :any.* Chaque élément de l'attribut doit être compatible avec un des types listés ;

:allocate
permet d'indiquer la taille d'une liste en vue d'optimiser la mémoire de l'application.

La définition d'une catégorie se fait par la macro *defcat*. Par exemple :

```
(defcat PNCT  :optional cmt gen prf)
```

La catégorie PNCT n'a pas d'attribut obligatoire mais peut potentiellement avoir les attributs CMT, GEN et PRF (avec les interprétations respectives : *commentaire*, *uniquement génération* et *préférences*). Ces attributs doivent avoir été définis par ailleurs. Il est a noter que toutes les catégories ont par défaut les attributs CAT, CAN et ALO qu'il n'est pas nécessaire de déclarer.

Certains des mot-clés de la macro *defcat* sont les suivants :

:required
permet de définir un attribut obligatoire pour toutes les entrées de cette catégorie. L'argument doit être soit un attribut précédemment déclaré, soit une liste dont le premier élément est le nom d'un attribut précédement défini et les éléments suivants des instances de valeurs de l'attributs. Ces valeurs doivent être compatibles avec le type de l'attribut ;

:optional
permet de définir des attributs optionnels ;

:constraint
permet de définir des contraintes sur les co-occurrences d'attributs.

Dans METAL, il n'est pas possible d'avoir comme valeur d'un attribut un autre attribut. Il s'agit d'un système relativement plat. De plus l'ensemble des attributs doit être déclaré globalemment et il n'est évidement pas possible d'avoir deux attributs différents ayant le même nom.

4.2. Structure de traits complexes

Une structure de traits complexes permet à ses attributs non seulement d'avoir comme valeurs des valeurs atomiques et des listes de valeurs mais également d'autres structures de traits.

Les structures de traits permettent une représentation détaillée des objets linguistiques mais aussi d'inclure certaines caractéristiques (morphologiques, syntaxiques et sémantiques) de façon structurée (ce que n'autorisent pas les listes de traits).

Les structures de traits complexes ont vu leur usage généralisés avec les grammaires d'unification [Abeillé 1993]. L'utilisation et la nature de ces objets sont bien théorisés (par exemple avec [Backofen & Smolka 1992] et [Carpenter 1992]).

4.2.1. Traits non typés

Une structure de traits est un ensemble ouvert et non-ordonné de traits. Un trait est un couple attribut-valeur. Les valeurs peuvent être des symboles atomiques ou des traits. Par exemple :

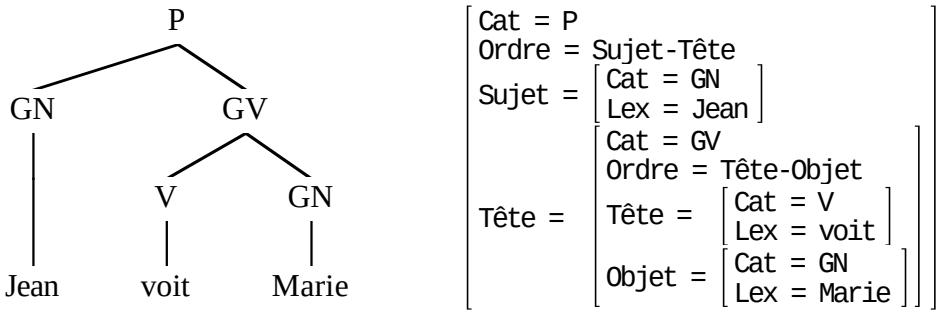


Figure 4.4 : Structure de traits complexes.

Cet exemple, tiré de [Abeillé 1993], montre comment utiliser des traits (ici Ordre) pour indiquer l'ordre des constituants.

Il est habituel d'associer une structure de traits complexes à un graphe orienté sans cycle (DAG). Le structure de la figure 4.4 prendrait ainsi la forme suivante :

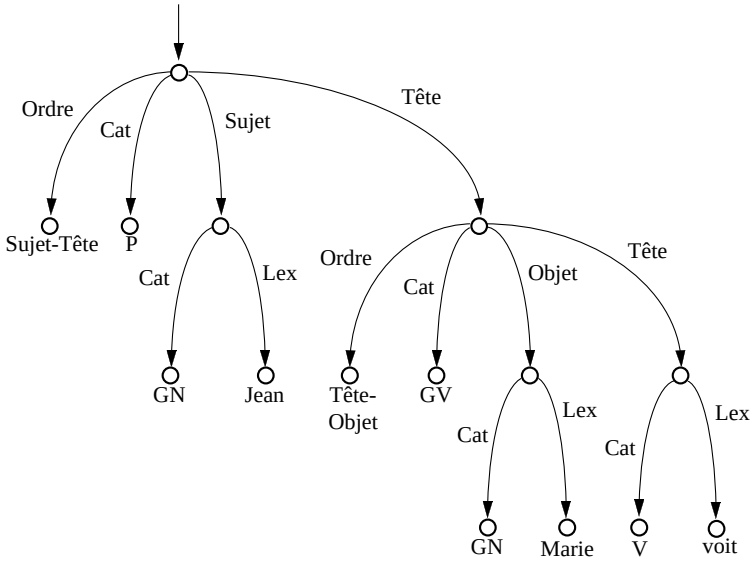


Figure 4.5 : Structure de traits complexes sous forme de DAG.

Dans les deux notations (crochet ou graphe), il n'y a pas d'ordre "horizontal" (entre frères). C'est pourquoi la structure donnée dans l'exemple 4.4, explicite l'ordre avec un

trait approprié. L'information contenue dans cet exemple peut donc être catégorisée entre des informations de décorations et des informations de structures[45].

Unification et réentrance

Il est nécessaire de faire la distinction entre structures identiques et structures à traits partagés. Une structure réentrante partage sa valeur avec une autre structure. Leurs valeurs seront toujours identiques quelles que soient les modifications ultérieures.

$$\begin{bmatrix} \texttt{Det} = [\texttt{Accord} = [\texttt{Num} = \texttt{Sing}]] \\ \texttt{Nom} = [\texttt{Accord} = [\texttt{Num} = \texttt{Sing}]] \end{bmatrix} \qquad \begin{bmatrix} \texttt{Det} = [\texttt{Accord} = \langle 1 \rangle [\texttt{Num} = \texttt{Sing}]] \\ \texttt{Nom} = [\texttt{Accord} = \langle 1 \rangle] \end{bmatrix}$$

Figure 4.6 : Structures sans et avec réentrance.

L'unification de ces structures avec la structure $[\texttt{Det} = [\texttt{Accord} = [\texttt{Genre} = \texttt{masc}]]]$ ne donne pas le même résultat (a et b) :

$$\begin{bmatrix} \texttt{Det} = [\texttt{Accord} = \begin{bmatrix} \texttt{Num} = \texttt{Sing} \\ \texttt{Genre} = \texttt{Masc} \end{bmatrix}] \\ \texttt{Nom} = [\texttt{Accord} = [\texttt{Num} = \texttt{Sing}]] \end{bmatrix} \qquad \begin{bmatrix} \texttt{Det} = [\texttt{Accord} = \langle 1 \rangle \begin{bmatrix} \texttt{Num} = \texttt{Sing} \\ \texttt{Genre} = \texttt{Masc} \end{bmatrix}] \\ \texttt{Nom} = [\texttt{Accord} = \langle 1 \rangle] \end{bmatrix}$$

Figure 4.7 : Résultats (a et b) de l'unification sans et avec réentrance.

Le trait *Accord* enchassé sous l'attribut *Nom* de la structure 4.7 b est égal au trait *Accord* enchâssé sous l'attribut *Det*. Ce n'est pas le cas de la structure présentée dans la figure 4.7 a.

La relation d'*extension* entre deux structures A et B est définie de la manière suivante :

A est une extension de B ($A \supset B$) si:

- tous les traits à valeur atomique de B existent dans A avec la même valeur ;
- tous les trais à valeur non atomique de A ont une valeur qui est une extension de la valeur du trait correspondant dans B.

Nous avons par exemple :

$$\begin{bmatrix} \texttt{Det} = [\texttt{Accord} = \begin{bmatrix} \texttt{Num} = \texttt{Sing} \\ \texttt{Genre} = \texttt{masc} \end{bmatrix}] \\ \texttt{Nom} = [\texttt{Accord} = [\texttt{Num} = \texttt{Sing}]] \end{bmatrix} \supset [\texttt{Det} = [\texttt{Accord} = [\texttt{Genre} = \texttt{masc}]]]$$

Le relation inverse de l'extension est la subsomption (dans l'exemple ci-dessus B subsume A). Du point de vue graphique A "subsume" B si A est contenu dans B.

Cette relation d'ordre permet de définir une hiérarchie. Le treillis qui en résulte possède une borne supérieure et une borne inférieure. La structure vide (appelée "top" et notée $\top$) subsume toutes les autres structures. La structure mal formée (appelée "bottom" et notée $\bot$) est subsumée par toutes les autres structures.

Il est maintenant possible de définir l'*unification* de deux structures A et B comme la plus petite structure extension de A et extension de B. Si cette structure n'existe pas alors l'unification échoue et rend "bottom".

---

45 On parle souvent d'informations algébriques et d'informations géométriques.

L'*anti-unification* (ou encore la généralisation) est l'opération (inverse de l'unification) qui associe à deux structures de traits la structure minimale qui subsume les deux.

D'autres opérations comme par exemple différents types d'unions ou de différences peuvent également être définies.

LFG

C'est avec les grammaires lexicales fonctionnelles que les structures de traits ont commencé à être utilisées comme alternatives aux représentations arborescentes. Les valeurs des traits sont non atomiques et peuvent être enchâssées dans des descriptions fonctionnelles (M. Kay). Les descriptions linguistiques gagnent en pouvoir expressif. Par exemple, la notion de fonction grammaticale peut ainsi être plus aisément réprésentée qu'avec des formalismes définis en termes d'arbres [Abeillé 1993].

LFG fait la distinction entre structures fonctionnelles (structure f) et structures de constituants (structure c). La structure c est un arbre de dérivation produit à partir de la grammaire. Aux nœuds de l'arbre sont associées des structures f (qui sont des structures de traits).

Une grammaire LFG est semblable à une grammaire attribuée :

```
P → SN                V
    ↑Suj =↓           ↑ =↓

V → dort
    ↑Num = sing, ↑Mode = indicatif, ↑Pers = 3, ↑Pred = 'dormir<Suj>'

V → Jean
    ↑Num = sing, ↑Genre = masc, ↑Pred = 'Jean'
```

Par exemple , la structure fonctionnelle (structure f) de «Jean dort»est :

$$\begin{bmatrix} \text{Suj} = \begin{bmatrix} \text{Pred} = \text{'Jean'} \\ \text{Genre} = \text{masc} \\ \text{Num} = \text{sing} \end{bmatrix} \\ \text{Pred} = \text{'dormir<Suj>'} \\ \text{Mode} = \text{indicatif} \\ \text{Num} = \text{sing} \\ \text{Pers} = 3 \end{bmatrix}$$

- Attributs à valeur atomique

Les traits morphologiques (Nombre, Genre, Mode, …) ont des valeurs atomiques. L'attribut Pred prend pour valeur la forme sémantique de la structure considérée. Par exemple, la valeur de Pred pour manger est "manger<Suj, Obj>". Pred n'est jamais unifiable.

Les valeurs possibles sont booléennes (noté - ou +), numériques (par exemple, l'attribut Pers) ou symboliques (par exemple Mode avec la valeur indicatif).

- Attributs à valeur non-atomique

Les attributs ayant une structure f comme valeur sont par exemple Suj (sujet) et Obj (objet).

- Attributs à liste de valeurs non-atomiques

L'attribut ajout a pour valeur toutes les structures f correspondant aux modifieurs d'un prédicat.

- Principes de bonne formation

Pour être bien formées, les structures fonctionnelles doivent répondre à trois contraintes. Le principe d'unicité exige qu'un attribut ne puisse apparaître plus d'une fois dans une structure. Le principe de cohérence exige que toutes les fonctions de sous-catégorisation soient gouvernées par un prédicat local. Le principe de complétude exige que toutes les sous-structures comportent toutes les fonctions gouvernées par un prédicat local.

4.2.2. Traits typés

Plusieurs systèmes disposant de structures de traits typés ont été développés soit pour le TALN, soit à des fins plus générales. La notion de type permet de profiter de l'approche à objet : l'abstraction et la généralisation via l'utilisation de l'héritage. Les types permettent :

- de factoriser des structures ;
- d'introduire une seconde hiérarchie (explicite cette fois) qui ajoute des contraintes sur l'opération d'unification.

L'unification entre deux structures de traits typées A (de type t1) et B (de type t2) est la plus petite structure C qui est une extension de A et une extension de B et dont le type t est l'unification de t1 et t2. Si t1 et t2 ne sont pas unifiables (t vaut ⊥), l'unification de A et B échoue même si les traits de A et B sont compatibles.

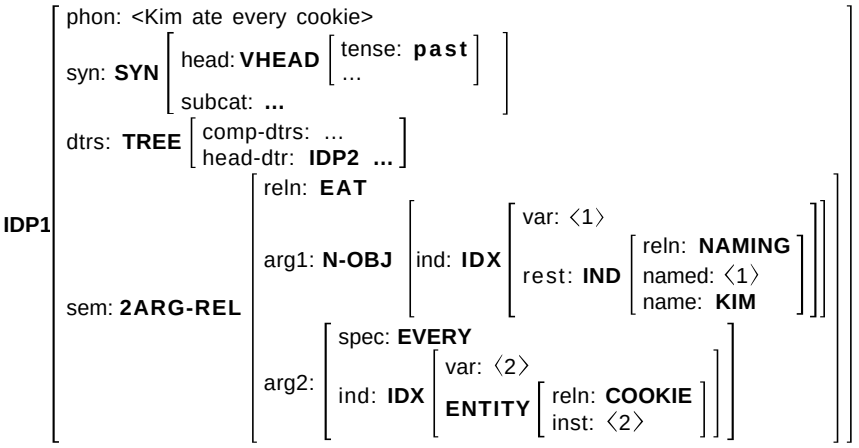


Figure 4.8 : Exemple de structure de traits typés.
(tiré de [Emele & al. 1990]).

HPSG

Les grammaires syntagmatiques dirigées par les têtes (HPSG — Head-driven Phrase Structure Grammars) s'inspire fortement de LFG et est considéré comme une variante de GPSG.

Tout objet linguistique doit appartenir à un type prédéfini. Les attributs sont typés car ils prennent une valeur qui est elle-même typée. La relation de subsomption définit une hiérarchie de types. La théorie HPSG définit une hiérarchie des type des objets linguistiques qu'elle prévoit. Cette hiérarchie est la suivante :

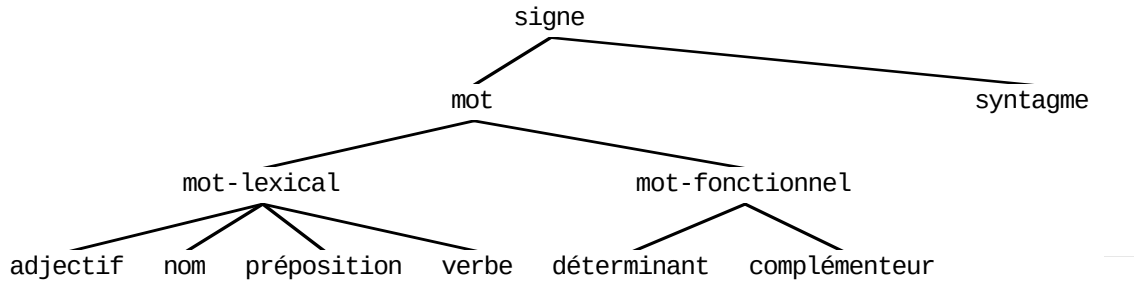


Figure 4.9 : Organisation hiérarchique des signes linguistiques de HPSG.

Les signes linguistes ont les définitions suivantes :

$$\text{signe} = \begin{bmatrix} \text{Phon} \\ \text{Synsem} \end{bmatrix}$$

$$\text{mot} = \text{signe} \cup \left[\text{Synsem} = \left[\text{local} = \left[\text{Cat} = [\text{Lex} = +]\right]\right]\right] = \begin{bmatrix} \text{Phon} \\ \text{Synsem} = \left[\text{local} = \left[\text{Cat} = [\text{Lex} = +]\right]\right] \end{bmatrix}$$

$$\text{syntagme} = \text{signe} \cup [\text{Branches}] = \begin{bmatrix} \text{Phon} \\ \text{Synsem} \\ \text{Branches} \end{bmatrix}$$

$$\text{mot-fonctionnel} = \text{mot} \cup \left[\text{Cat} = \left[\text{Tête} = [\text{Spec}]\right]\right] = \begin{bmatrix} \text{Phon} \\ \text{Synsem} = \left[\text{local} = \left[\text{Cat} = \begin{bmatrix} \text{Lex} = + \\ \text{Tête} = [\text{Spec}] \end{bmatrix}\right]\right] \end{bmatrix}$$

$$\text{préposition} = \text{mot} \cup \left[\text{Cat} = \left[\text{Tête} = [\text{Prepform}]\right]\right] = \begin{bmatrix} \text{Phon} \\ \text{Synsem} = \left[\text{local} = \left[\text{Cat} = \begin{bmatrix} \text{Lex} = + \\ \text{Sous-cat} \\ \text{Tête} = [\text{Prepform}] \end{bmatrix}\right]\right] \end{bmatrix}$$

4.3. Prototypes

Les langages à objets sont souvent divisés en deux grandes catégories, les langages de classes et les langages de prototypes.

Les langages de classes distinguent les classes et les instances. Les classes disposent d'attributs et fournissent une réponse à des messages par l'activation de méthodes. Les instances sont des réalisations particulières de classes. On peut faire le parallèle avec les types et les variables des langages classiques.

Les langages basés sur les prototypes ne mettent en jeu qu'une seule catégorie d'entité. L'approche consiste à ne définir que des objets concrets décrivant les entités pertinentes pour la réalisation d'une solution à un problème. Les prototypes ne sont pas des abstractions (comme le sont les classes) et ne sont pas (directement) liés à d'autres objets. Le prototype est une entité "auto-suffisante" capable de répondre à des messages. Les objets sont créés par clonage (clonage profond ou de surface) ou ex nihilo.

Une taxonomie originale des manipulations liées aux prototypes est donnée par [Dony, Malenfant & Cointe 1992].

4.3.1. Frames

Les frames, introduites par M. Minsky, sont destinées à représenter des connaissances générales. Leur conception repose généralement[46] sur une structure à trois niveaux, frame, attribut, facette. Chaque frame dispose d'un ensemble ouvert de champs (*slots*) équivalents aux attributs de structures de traits. Un champ possède à son tour un certain nombre de facettes, elles-mêmes possédant des valeurs.

De nombreux systèmes se basent sur les frames, par exemple le langage KR à la base du système Garnet ([Guise 1993] et [Myers & al. 1992]) et le langage K ([Cavazza & Zweigenbaum 1992]).

GFP

Le système de frames présenté ici est tiré de [Lafourcade 1994b]. Le système GFP (Geta-Frame Protocol) est un système de gestion de frames basé sur des protocoles stratifiés. La définition de ce gestionnaire a été à l'origine inspiré du FrameKit de [Nirenburg 1989a]. Nous y avons ajouté, à titre expérimental, la possibilité de définir des protocoles. Ce gestionnaire a été réalisé en CLOS.

Un niveau supplémentaire, celui des vues, a été rajouté aux trois niveaux classiques (frame, champ, facette). On a par exemple :

```
<:frame geta  (is-a (value (common laboratory)))
              (member-of  (value (common imag cnrs)))
              (location   (city (common grenoble)
                          (country (common france)))
```

Le symbole geta est associé à une frame. Celle-ci spécifie deux champs prédéfinis (*is-a* et *member-of*). *value* est la facette par défaut. *common* est la vue par défaut.

Les frames sont liées à des démons[47], procédures locales déclenchées par des actions spécifiques. Les démons sont contenus dans des facettes particulières qui peuvent être héritées d'une frame sur l'autre. Un démon (qui est une valeur de facette) peut être n'importe quelle expression LISP. Cette expression est évaluée si le démon est déclenché. Cette expression, peut avoir 5 variables spéciales :

- !frame correspondant au nom de la frame courante ;
- !slot correspondant au nom du champ courant ;
- !facet correspondant au nom de la facette courante ;
- !view correspondant au nom de la vue courante ;
- !filler correspondant au nom de la valeur courante (l'expression courante).

Les démons suivants sont définis :

- :if-added ;
- :if-needed ;
- :if-accessed ;
- :if-erased.

---

46 Certains systèmes sont plus complexes, par exemple les frames utilisées dans [Goodman & al. 1991] définissent un niveau suplémentaire, les vues.

47 On parle parfois de réflexes.

Quand une valeur est ajoutée à un attribut et que ce dernier spécifie un démon *if-added*, ce démon est alors déclenché. Concrètement, si un attribut dispose d'une facette *if-added* toutes ses valeurs sont alors évaluées. Les autres démons fonctionnent de façon similaire dans les cas où :

on accède une valeur inexistante ;

on accède une valeur qui existe ;

on efface une valeur.

Un nouveau type de démon, :restrictions, permet de n'accepter l'écriture que de valeurs vérifiant un prédicat. Par exemple :

```
(frame-add-filler 'geta 'member-of :restrictions '(institution-p !filler) :view
'common)
> t

(frame-add-value 'geta 'member-of 22)
> nil
```

La valeur 22 est refusée car elle ne correspond pas à une institution.

Une relation est une représentation formée de deux liens. Un lien est une facette qui contient un symbole qui correspond à une frame. Les deux liens ont des noms opposés (par exemple : père/père-de, classes/instances). Voici un exemple inspiré de [Goodman & al. 1991]:

```
<:frame father-of  (instance-of (value (common relation)))
                   (inverse (value (common father)))>
```

Créons les frames ken et rudy :

```
(frame-add-value 'ken 'father-of 'rudy)
>T

(frame-get-values 'ken 'father-of)
>(RUDY)

(frame-get-values 'rudy 'father-of)
>(KEN)
```

Certaines relations sont prédéfinies : *is-a/subclasses, instance-of/instances, part-of/parts.*

```
(frame-get-values 'relation 'instances)
> (FATHER FATHER-OF PARTS PART-OF SUBCLASSES IS-A INSTANCES INSTANCE-OF)
```

Les frames peuvent partager de l'information via un mécanisme d'héritage. Nous pouvons prendre comme exemple les frames voiture et Twingo :

```
<:frame car (wheels (value (common)))>

<:frame twingo (instance-of (common car)))>

(frame-get-values 'twingo 'wheels :inherit t)
> (4)

<:frame ma-voiture   (instance-of (value toyota)))
                     (color (value yellow)))>

(frame-get-values 'ma-voiture 'color :inherit t)
> (yellow)
```

```
(frame-get-values 'ma-voiture 'wheels :inherit t)
> (4)
```

Plusieurs représentations d'une même frame peuvent être définies simultanément grâce aux vues.

La notion d'environnement permet de regrouper des frames. Dans un environnement donné toutes les frames doivent être associées à des symboles distincts. Si aucun environnement n'est spécifié, alors un environnement par défaut est utilisé.

```
<:frame my-env (instance-of (value (common environment)))>
<:frame ken   (environment (value (common my-env)))
              (age (value (common 16)))>
```

Nous introduisons, ici, une nouvelle relation : *environment/environment-of*

```
(frame-get-values 'my-env!ken age)
>(16)
(frame-get-values 'ken age)
>error: KEN is not a definied frame.
```

4.3.2. Systèmes hybrides

Les systèmes hybrides se basent sur plusieurs paradigmes.

LIFE

LIFE [Aït-Kaci & al. 1989] est une synthèse de la programmation logique, la programmation fonctionelle et la programmation à objets. Ces travaux se situent dans la continuité de LOGIN ([Aït-Kaci & al. 1989], [Aït-Kaci, Meyer & Roy 1992] et [Aït-Kaci & al. 1986]). La seule implémentation connue de ce langage est Wild-Life [Aït-Kaci & al. 1992].

La structure de base de LIFE est le ψ-terme, dont voici quelques exemples :

```
42,  -5,66
int, hello, abc
"une chaîne de caractères"
freddy(nails ⇒ long, face ⇒ ugly, films ⇒ 5)
```

Il peut donc s'agir de nombres (entiers, réels, etc), de symboles (semblables à ceux de LISP), de chaînes de catactères ou de symboles avec des attributs symboliques (des structures de traits).

On peut voir un ψ-terme comme la généralisation d'un terme PROLOG dans lequel les positions des sous-termes sont spécifiées par des mots-clés (plutôt qu'implicitement notées par des positions numériques). Afin d'être compatible avec PROLOG, les positions peuvent être marquées implicitement :

```
thing(a, b, c)
```

qui est équivalent à :

```
thing (1 ⇒ a, 2 ⇒ b, 3 ⇒ c)
```

Dans ce système, le bouclage arrière est possible.

Une des caractéristiques qui distingue LIFE de PROLOG est l'approche fonctionelle. une définition de fonction consiste en une ou plusieurs règles composées d'une tête et d'un corps. Ces deux parties sont des ψ-termes. Une expression fonctionnelle est un ψ-terme dont le symbole de racine est un symbole de fonction :

```
fact(n) ⇒ si n = 0 alors 1 sinon n*fact(n-1)
```

Chapitre 5

DÉCOR – implémentation d'une approche générique

L'étude de l'existant nous amène à la définition d'un langage de représentation de structures adaptées aux traitements linguistiques. Nous avons ici un double objectif : pouvoir définir les formalismes présentés dans le chapitre précédent, soit comme des sur-couches de ce noyau, soit comme des sous-ensembles, et ouvrir le langage à des applications non exclusivement linguistiques.

Les aspects les plus intéressants sont les suivants. D'abord, la définition explicite de types le rapproche des langages de classes. DÉCOR contient aussi la notion de prototype. Ensuite, la généralisation du co-indexage nous amène à introduire la notion d'interprétation des valeurs des attributs. En quatrième lieu, la notion d'héritage a été étendue. Enfin, DÉCOR est dynamique : tous les attributs d'une structure peuvent être modifiés durant l'exécution, et on peut changer dynamiquement le type d'une structure.

DÉCOR doit beaucoup aux travaux présentés dans [Boitet, Hue & Collomb Réd. 1982] tout en gardant une approche aussi générale que possible. Ce langage n'est pas l'union de tous les formalismes possibles, mais un formalisme autonome et économe, dont les objets et les primitives permettent la création d'un grand spectre d'applications.

La définition de DÉCOR a été faite dans la perspective du TALN, mais indépendamment de toute théorie linguistique particulière.

5.1. Besoins

Un certain nombre d'aspects des structures de traits complexes ont été généralisés dans le langage DÉCOR. Les décorations sont vues comme des prototypes particuliers qui peuvent être contraints à l'aide de types. Les valeurs peuvent être considérées de multiples manières, grâce à des "interprétations". L'ensemble du langage est fortement dynamique, ce qui permet par exemple aux linguistes de redéfinir certains types à l'exécution.

5.1.1. Décorations, Types et Prototypes

L'objet de base manipulé dans DÉCOR est la *décoration*. Une décoration est une forme particulière de frame. Dans cette perspective, il s'agit donc bien d'un langage de prototypes. Cependant, la décoration est un objet qui peut être contraint par l'adjonction de *types* qui forment une seconde catégorie de frames.

Valeurs

Il est nécessaire de définir clairement quelles sont les valeurs que manipulent d'une part le système et d'autre part le linguiste. Il est nécessaire de réserver certaines valeurs au système.

L'ensemble des valeurs de DÉCOR est constitué des symboles, des nombres, des chaînes et des frames (on parlera indistinctement de prototypes). Les décorations et les types sont des types particuliers de frames. Un certain nombre de symboles sont prédéfinis. De manière générale, tous les symboles commençant par le caractère ":" sont réservés.

Une valeur peut être qualifiée d'atomique (nombre, chaîne, booléen, symbole), de complexe (différents types de listes définies en intention ou en extension) ou d'agrégée (un prototype).

Nous pouvons, par exemple, avoir les différentes valeurs de décoration suivantes :

```
5
gn
:null, :indef
"Jean"
(1, 2, 3)
(5, "Jean", vrai, (1, 2, 3))
```

La structure suivante est une frame :

```
⎡ Phon = '("j" "&")          ⎤
⎢ Syntaxe = ⎡ Catégorie = N ⎤ ⎥
⎢           ⎣ Sous-cat      ⎦ ⎥
⎣ Sémantique = "Jean"         ⎦
```

On la notera :

```
<:frame  nil  (phon '("j" "&"))
           (Syntaxe nil <:frame
                          (Catégorie N)
                          (Sous-cat)>
           (Sémantique "Jean")>
```

Le symbole réservé *:frame* permet de préciser qu'il s'agit bien d'un prototype. Le symbole réservé NIL indique que la frame n'est pas nommée.

La notation entre chevrons ('<' et '>') est utilisée pour écrire de façon lisible une frame. Une autre notation est utilisée pour définir une frame.

:null désigne la valeur nulle et est la valeur de tout attribut présent sans valeur (par exemple, l'attribut Sous-cat ci-dessus). *:indef* est la valeur d'un attribut absent, et est aussi produite par certaines fonctions et opérateurs.

Variables

Une variable est un symbole auquel est associée une valeur. Les variables sont définies dans un environnement. Un environnment par défaut, unique, est prédéfini et est utilisé si aucun autre environnement n'est spécifié.

Le fonctionnement des environnements liés au GFP (Geta Frame Protocol) a été présenté au §4.3.1.

Décorations

Intuitivement, une décoration agrégée peut être représentée par un arbre dont les nœuds internes portent des valeurs. Ces valeurs ne peuvent pas être agrégées. Les branches ne portent pas d'information, contrairement aux structures de traits complexes qui portent comme symboles les noms des attributs. Il n'y a pas d'ordre entre les fils d'un nœud. La structure représentée par la figure 4.4 peut ainsi être représenté par le graphe de la figure 5.1. La représentation sous forme de crochets reste toujours utilisable.

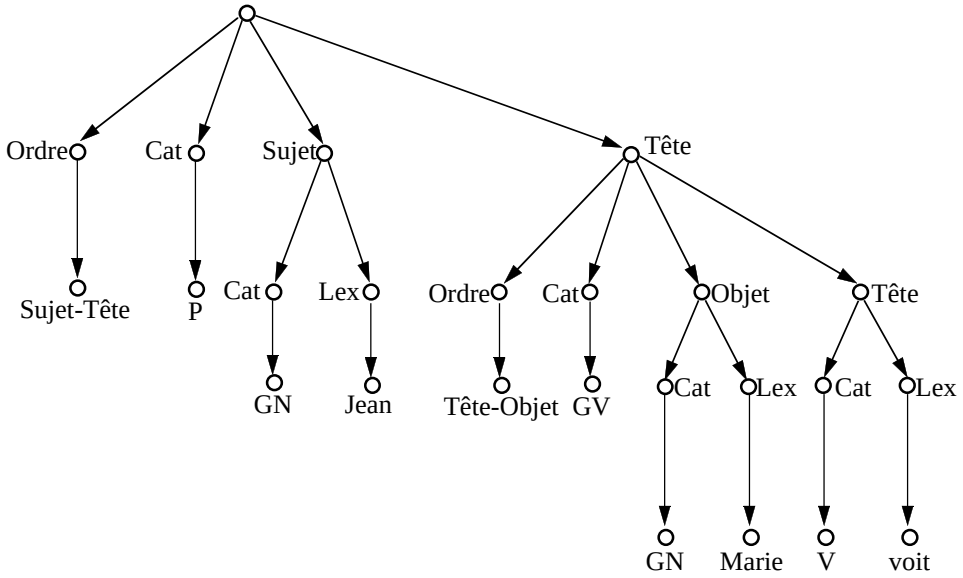


Figure 5.1 : «Jean voit Marie» sous forme de décoration.

La notion d'attribut est relative à un nœud. Le nom de l'attribut est n'importe quelle valeur atomique. Par définition, tous les nœuds frères doivent avoir des noms différents. La valeur d'un attribut est la forêt dominée par le nœud qui représente l'attribut. Une structure qui dispose de deux attributs frères de même nom est considérée comme mal formée.

Le nom complet d'un attribut est la séquence de valeurs de la racine de la décoration au nœud contenant l'attribut. Par exemple, dans la décoration de la figure 5.1, nous avons les chemins suivants :

```
Cat.P
Sujet.Cat
Tête.Tête.lex
```

La valeur d'un chemin est la forêt dominée par le dernier nœud du chemin. On peut considérer les valeurs immédiates ou les valeurs totales. Une valeur immédiate est la liste des racines des forêts en bout de chemin. Une valeur totale est la forêt complète.

Il est parfois désirable d'abréger la notation d'un long chemin en omettant les nœuds intermédiaires (entre la racine et le nœud d'arrivée). On utilise alors la notation du double point :

```
Tête..GV
```

Une telle notation est potentiellement ambiguë. Pratiquement, en cas d'ambiguïté on considérera que cette notation correspond au chemin minimal. Un chemin est plus court qu'un autre s'il contient moins d'éléments. Si leurs longeurs sont égales, on dispose d'un ordre total sur les valeurs atomiques (symbole < chaîne < nombre < booléen)[48]. Ce choix est uniquement pratique et est, en fait, contradictoire avec l'absence d'ordre latéral. DÉCOR fournit des fonctions pour modifier ces préférences.

Types

La notion de type utilisée dans les formalismes de structures de traits typés permet de définir la spécification minimale qu'une structure doit avoir pour être considérée comme appartenant à ce type. Il s'agit donc toujours d'un ensemble infini dont l'élément le moins spécifié est le type considéré. La notion de type que nous adoptons est radicalement différente en ce qu'elle permet de définir (en intention ou en extension) l'ensemble des valeurs que peut prendre une décoration ou un trait.

Un type est un ensemble de contraintes portant sur une décoration ou sur les traits. Toute décoration typée doit se conformer à son type, c'est-à-dire que la structure de l'arbre doit respecter le schéma prévu par les contraintes formulées par le type.

Les types sur les décorations permettent d'énumérer l'ensemble des attributs permis et ceux-là seulement. Une décoration d'un tel type est "fermée", et ne peut pas être davantage spécifiée à l'aide d'autres attributs. Des types "ouverts" permettent de contraindre certains attributs s'ils sont présents, mais n'interdisent pas l'ajout d'autres attributs.

Les types peuvent être organisés en une hiérarchie de types. Les nouveaux types peuvent être crées par dérivation de types déjà existants. La dérivation peut prendre la forme de compositions, de surcharges et de spécialisations.

Le résultat de certaines opérations sur les décorations est influencé par les types. Par exemple, dans le cas de l'unification, les types contraignent davantage le résultat et peuvent faire échouer l'opération si les types sont incompatibles.

Différence entre structure de traits et décoration

Une structure de traits définit un ensemble infini d'objets. La structure est d'autant plus spécifiée (ou contrainte) qu'elle a d'attributs définis.

Une décoration définit un ensemble de valeurs. Plus une décoration est spécifiée, plus l'ensemble de valeurs est important (en d'autres termes, plus l'arbre la représentant est grand, en largeur ou en profondeur). Les valeurs ne sont pas des contraintes (ce rôle est tenu par les types).

---

48 Lors d'une comparaison, les symboles sont transformés en chaînes et sont comparés avec l'ordre lexicographique. vrai est inférieur à faux. Par exemple : Cat.P < Ordre.Sujet-Tête < Sujet.Cat.GN.

5.1.2. Héritage multiple

Certains systèmes offrent des mécanismes particuliers pour différents types d'héritage [Ducournau & Habib 1989]. Dans [Genthial 1991], l'accent est mis sur l'intérêt de l'héritage multiple dans le cas de langages de représentation linguistique basés sur des structures de traits typés. Il permet essentiellement des définitions factorisées des propriétés linguistiques.

Toute décoration peut hériter d'une ou de plusieurs autres décorations. La sémantique adoptée pour l'héritage multiple est celle définie dans CLOS [Steele 1990]. L'héritage définit un ordre partiel sur les décorations. Une nouvelle décoration peut donc être définie à partir de l'application de l'opération d'unification sur une liste de décorations nommées ou anonymes. Par exemple :

```
<:decor   a
     (fa1 "hello")
     (fa2 "hello")>

<:decor   b
     (fb  666)>

<:decor   a-prime
     (fa2 "world")>
```

```
(define! d  :decor
     (:is-a '(a b a-prime))
> <:decor d
     (:is-a '(a b a-prime))>

(get-value 'd.fa1)
> "hello"

(get-value 'd.fa2)
> "world"
```

5.1.3. Multiples interprétations de valeurs

Habituellement, un attribut est constitué d'un trait (en général un symbole) suivi d'une valeur. Les systèmes de frames ont introduit la notion de facette permettant à un trait de disposer d'un ensemble de valeurs référencées elles-mêmes par des symboles. De plus, dans le langage KR, un trait peut porter une formule dont l'interprétation fournira dynamiquement la valeur du trait. La traitement effectué sur l'information liée au trait dépend donc de la nature de cette information.

La notion d'interprétation de valeurs permet de généraliser cette idée. Cependant, au lieu de déduire le traitement approprié en fonction du type de la valeur portée par le trait, il peut être intéressant de noter explicitement cette information (l'interprétation). D'une part, tout type de valeur est potentiellement acceptable et son traitement sera fonction non seulement de son type mais aussi de son interprétation. De plus, cette approche générique permet de rajouter au langage de nouvelles interprétations si le besoin s'en fait sentir.

Les interprétations les plus courantes sont :

- la valeur directe qui consiste à rendre strictement la valeur portée par le trait. Il s'agit de l'interprétation par défaut ;
- la formule qui est équivalente aux formules de KR ;
- la référence qui est une "adresse" vers un autre objet du langage.

Un accès à un attribut rendra une valeur dépendante de la valeur effectivement portée par l'attribut mais aussi de son interprétation. Il est possible de changer l'interprétation d'une valeur pour un attribut sans toucher à cette valeur. Cela n'est évidement pas possible quand l'interprétation de la valeur est directement fonction de son type.

5.1.4. Dynamisme

Bien que la notion de langage dynamique ne soit pas très récente, la possibilité de redéfinir durant l'exécution (ou au moins sans recompilation) les types des objets est encore peu exploitée. Des travaux sur des langages dynamiques émergent petit à petit, on pense par exemple à [Apple Computer Inc. 1992][49].

Le "dynamisme" permet donc au développeur de modifier les définitions liées à ces types (ou à ces classes) et de reprendre l'exécution sans recompilation. La méthode la plus directe est de redéfinir complètement une variable contenant un type par exemple. Une approche complémentaire consiste à fournir des primitives qui permettent de modifier certains attributs des types. Le processus de développement s'en trouve facilité.

Il semble intéressant de fournir ce type de fonctionnalités dans les langages de représentations linguistiques. Cependant, le risque de rendre incohérent l'environnement d'exécution d'un programme n'est pas nul. Par exemple, [Bergstein 1991] pose clairement certains problèmes liés à la transformation de classes. Le système doit être capable d'avertir le développeur des conséquences d'une modification et lui donner la possibilité de revenir en arrière.

5.2. Définition externe

La définition externe de DÉCOR est l'ensemble des structures et des fonctions accessibles à l'utilisateur. Cependant, le langage présenté ici n'est qu'un langage noyau dont la syntaxe reste subordonnée à LISP. Les dialectes de DÉCOR (voir §3.3.3.) doivent fournir des syntaxes plus accessibles aux linguistes. Un exemple d'un tel dialecte est donné en Annexe D.

5.2.0. Présentation générale

La terminologie employée dans ce contexte est celle qu'emploient couramment les linguistes informaticiens. Les prototypes sont appellés "décorations". Certaines décorations particulières sont appelés "types".

La représentation d'un objet se fait au moyen d'une liste imbriquée dont le premier élément est un symbole dénotant la catégorie de l'objet. Les arguments suivants sont des listes de longueur et de contenu variables. Par exemple, le type *petit-nombre* est noté :

```
<:type petit-nombre
       (:is-a :number)
       (:range 0 5)>
```

[49] Il semblerait que les possibilités de rédéfinition incrémentale à l'exécution des classes aient finalement été abandonnées dans le langage Dylan. Ce choix a été essentiellement motivé par un nécessaire compromis entre efficacité et dynamicité. Nous pensons que, dans le cadre d'un langage offrant aux linguistes une démarche exploratoire (ce qui n'est certes pas l'objectif de Dylan), de telles fonctionnalités peuvent s'avérer intéressantes.

```
<:decor   mon-nombre
       (:type petit-nombre)>
```

Un objet anonyme est noté de la même façon mais sans être nommé (le nom est alors fait le symbole NIL). Un nom est forcément un symbole. Si un nom valide est fourni, alors une variable dont le symbole est ce nom est automatiquement créée dans l'environnement courant et liée à l'objet.

```
<:type (:is-a :number)
       (:range 0 5)>
```

Il est inutile de définir un type anonyme tout seul puisqu'il ne peut être référencé mais il peut être défini à l'intérieur d'une autre définition.

Voici, un exemple de décoration dont le type est anonyme :

```
<:decor mon-autre-nombre
       (:type <:type  (:is-a :number)
                      (:range -10 10)>
        )>
```

Dans DÉCOR, les objets sont créés à l'aide de la macro *define!* dont les arguments dépendent du type d'objet à créer.

```
define! nom type argument*                                    [macro]
define! nil type argument*                                    [macro]
```

Le *nom*, s'il est présent, est un symbole non réservé. Le *type* est un symbole réservé (on ne considérera dans un premier temps que les symboles *:type* et *:decor*). Zéro, un ou plusieurs arguments peuvent suivre. L'*argument* est une liste dont le premier élément doit être un trait.

```
(define!   petit-nombre  :type
             (:is-a :number)
             (:range 0 5))
> <:type   petit-nombre
        (:is-a :number)
        (:range 0 5)>
(define! mon-nombre :decor
           (:is-a :decor)
           (:type 'petit-nombre)
           (:valeur 8))
> <:decor  mon-nombre
           (:is-a :decor)
           (:type 'petit-nombre)>
(define!   nil :type
           (:is-a :number)
           (:range 0 5))
> <:type   (:is-a :number)
           (:range 0 5)>
```

On peut avoir comme définition imbriquée :

```
(define! mon-autre-nombre :decor
        (:is-a :decor)
        (:type (define! nil :type
                     (:is-a :number)
                     (:range 10 50)) )
        (:valeur 12))
```

ou encore, une définition nommée :

```
(define! mon-autre-nombre :decor
        (:is-a :decor)
        (:type (define! autre-nombre :type
                    (:is-a :number)
                    (:range 10 50)) )
        (:valeur 12))
```

Nous avons vu qu'une valeur pouvait avoir plusieurs interprétations. L'interprétation par défaut est l'interprétation directe. Ce type d'information n'est noté que si la valeur est différente de la valeur par défaut. Par exemple :

```
<:decor ma-decoration
       (:trait1  (:valeur 'mon-nombre)
                 (:interpretation 'reference)
        )>
```

Le *trait1* a pour valeur le symbole *'mon-nombre.* Cependant cette valeur est considérée comme une référence et la lecture standard du trait rendra la valeur pointée par la référence. La référence est un lien dynamique vers d'autres objets.

```
(get-value 'ma-decoration.trait1)
> <:decor  mon-nombre
           (:type 'petit-nombre)>
(get-value (get-value 'ma-decoration.trait1))
> 8
(set-value 'mon-nombre.valeur 9)[50]
> t
(get-value (get-value 'ma-decoration.trait1))
> 9
```

L'attribut *trait1* de *ma-decoration* à pour valeur le symbole *'mon-nombre.* Cependant, l'interprétation indique qu'il s'agit d'une référence. La valeur rendue est donc la décoration mon-nombre.

Pour simplifier l'écriture, nous omettrons dans les décorations l'attribut *:is-a* s'il porte la valeur par défaut *:decor.* De façon équivalente, nous omettrons dans les types l'attributs *:is-a* s'il porte la valeur par défaut *:top.*

5.2.1. Types

Un type permet de spécifier un domaine de valeurs. On appelle *extension* d'un type l'ensemble des valeurs acceptables que prévoit ce type. Les types peuvent définir leur extension en intention (par exemple les réels) ou en extension par énumération directe de valeurs.

Les types permettent de contraindre le domaine de valeur des attributs des décorations. Pour les décorations agrégées, la présence ou l'absence des attributs peuvent aussi être spécifiées à l'aide des types. Un type peut être vu simplement comme une contrainte sur les valeurs que peut prendre un objet.

---

50 Une notation plus directe est : (set-value 'mon-nombre 9)

Les types peuvent être construits par dérivation ou composition de types prédéfinis. La dérivation consiste à leur rajouter des contraintes qui peuvent éventuellement demander des arguments. La composition permet d'en définir un nouveau à partir du calcul d'une formule appliquée sur plusieurs types. Ces deux constructions sont resteintes par la hiérarchie de types.

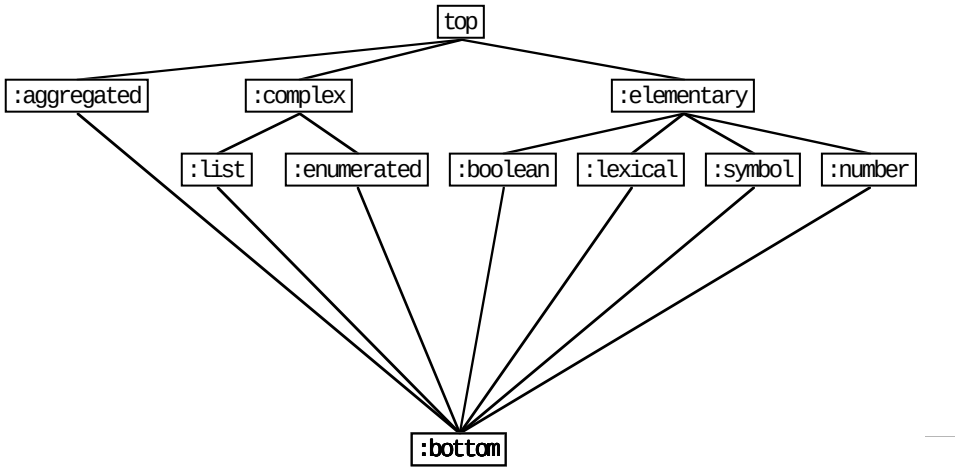


Figure 5.2 : Hiérarchie des types prédéfinis de DÉCOR.

Une décoration peut spécifier son type par un attribut prédéfini *:type*. Si une décoration n'est pas typée, aucune contrainte spécifique ne s'applique alors.

Seules certaines contraintes sont décrites ici. Quelques contraintes générales ne sont pas explicitées comme par exemple, *:default-value* qui permet pour un attribut la définition d'une valeur par défaut.

Types élémentaires

La création de nouveaux types peut se faire par dérivation de types prédéfinis. La dérivation prend ici la forme d'ajout de contraintes. Les types élémentaires sont lexicaux, numériques, booléens et symboliques.

- `:lexical` `[type de base]`

Toute décoration de type lexical doit associer une chaîne de caractères au trait valeur.

```
<:decor decoration-lexicale
        (:type lexical)
        (:value "Hello World!")>
```

Les opérations autorisées par ce type sont les opérations classiques sur les chaînes de caractères. Les contraintes possibles sont la longueur (minimale et maximale) et une liste de valeurs interdites.

```
<:type prefixe
        (:is-a :lexical)
        (:range 0 5)>

<:type mot-sauf-connecteur
        (:is-a :lexical)
        (:forbidden-values '("et" "ou" "donc" "ainsi"
                             "alors" "sinon"))>
```

Le type préfixe autorise des chaînes de cinq caractères au plus. Le type *mot-sauf-connecteur* autorise toutes les chaînes sauf celles qui sont contenues dans la liste.

- `:number` `[type de base]`

Toute décoration de type numérique doit associer un nombre au trait valeur. Il est possible de particulariser le type numérique avec plusieurs contraintes spécifiques. La contrainte *:range* définit les bornes supérieures et inférieures acceptables. Par exemple, le type *note* n'accepte que des valeurs numériques entre 0 et 20.

```
<:type note
         (:is-a :number)
         (:range 0 20)>
```

Les deux arguments de la contrainte *range* sont soit des réels, soit une des deux valeurs symboliques *'negative-infinite* et *'positive-inifinite*.

Le type numérique accepte implicitement n'importe quel réel. La contrainte *integer* permet de n'accepter que des entiers. La contrainte *:forbidden-values* est également applicable avec une liste de nombres comme arguments.

- `:boolean` `[type de base]`

Un trait de type booléen peut avoir la valeur *:vrai* ou *:faux* (équivalents aux deux symboles de LISP T ou NIL).

```
<:decor vole-p
         (:type :boolean)
         (:value :vrai)>
```

Il n'existe pas de contraintes prédéfinies pour le type booléen.

- `:symbol` `[type de base]`

Un symbole est un élément atomique qui ne peut être que désigné. Il ne peut pas être l'objet d'opérations comme la concaténation ou le tri.

```
<:decor simple-symbol
           (:type :symbol)
           (:value 'ex1)>
```

La seule contrainte applicable au type symbole est l'interdiction de valeurs. L'argument de la contrainte *:forbidden-values* accepte comme paramètre une liste de symboles.

```
<:type nom-de-type-utilisateur
         (:is-a :symbol)
         (:fordidden-values '(:number :string :boolean :symbol))>
```

Types de listes

Une décoration de type liste peut contenir plusieurs valeurs. Les contraintes peuvent porter sur le nombre et les types de valeurs. Dépourvue de contraintes, une décoration de type liste peut avoir un nombre ouvert de valeurs de types quelconques.

- `:list` `[type de base]`

```
<:type ma-liste
         (:type :list)
         (:value '(2 5 1 "hello" 'gn :faux))>
```

La contrainte *range*, similaire à celle liée au type numérique, permet de spécifier le nombre minimum et maximum d'éléments. La borne inférieure ne peut évidement être

inférieure à zéro. La contrainte *:forbidden-values* est également applicable avec une liste de valeurs de types autorisés comme arguments. Les contraintes *:allowed-types* et *:forbidden-types* pemettent de n'autoriser ou de n'interdire que des valeurs de certains types.

```
<:type liste-note
          (:type :liste)
          (:range 0 30)
          (:allowed-types '(:number))>
```

Le type *liste-note* autorise comme valeur des listes d'au plus 30 éléments et exclusivement du type *:number*.

Il est à noter qu'une valeur ne dispose pas en elle-même d'un type mais qu'elle est compatible avec un type si elle est acceptable pour celui-ci. Les types autorisés définissent donc les valeurs admissibles et non pas des décorations dont le type serait autorisé. Il faut donc distinguer type de valeur et type de décoration.

Nous notons le type qui accepte comme valeurs uniquement des valeurs de type *t:*

`(t :value)`

par exemple `(:number :value)`

et nous notons le type qui accepte comme valeurs uniquement des décorations de type t:

`(t :decor)`

par exemple `(:number :decor)`

Nous considérons, dorénavant, la notation simple (par exemple *:number*) comme une disjonction entre les types de valeur et les types de décoration. Une valeur de type `:number` peut porter aussi bien un nombre qu'une décoration dont la valeur est un nombre.

Types énumérés

Un type énuméré donne l'ensemble des valeurs acceptables. Il s'agit donc d'une définition en extension et non pas en intention (comme le type numérique, par exemple). Deux catégories de types sont distinguées selon la contrainte d'exclusivité.

- `type exclusif` `[type de base]`

Une valeur de type exclusif peut prendre une seule valeur parmi les valeurs énumérées.

```
<:type t-drv
          (:type :enumerated)
          (:exclusive :vrai)
          (:allowed-values '(VN VA1 VA2 AN AV NA NV))>
```

Le type énuméré exclusif *t-drv* correspond à la dérivation dans l'analyse morphologique de l'allemand (d'après [Guilbaud 1980]).

- `type non-exclusif` `[type de base]`

Soit t1 un type exclusif d'extension {:null, v_1, … v_n}, le t2 est non-exclusif si son extension est l'ensemble des parties de l'extension de t1 moins :null ({v_1, …v_n}) ou :null.

```
<:type t-kms
          (:type :enumerated)
          (:exclusive 'faux)
          (:allowed-values '(VB NM ADJ ADIP DR COP PC NA))>
```

Les valeurs autorisées peuvent être hétérogènes (de types différents).

Types agrégés

Un type aggrégé permet d'énumérer et de contraindre les traits que peut avoir une décoration. De plus, il est possible de spécifier si une décoration consitue un domaine fermé (touts les traits possibles ont été énumérés) ou ouvert (de nouveaux traits peuvent être rajoutés). Les traits spécifiés par le type peuvent être instanciés de manière exclusive (un seule à la fois) ou non-exclusive (n'importe quel sous-ensemble des traits). Les traits peuvent également être déclarés obligatoires.

Un type agrégé sera noté sous la forme :

```
<:type [nom-du-type]
     (:is-a super-type)
     (trait    (:type valeur)
               (facette valeur)*)*>
```

Une facette[51] permet de fournir pour un attribut des informations complémentaires à la valeur. Les contraintes portant sur un attribut seront exprimées par l'utilisation de facettes.

Par exemple :

```
<:type mon-type
     (:is-a :aggregated)
     (a   (f1 2)
          (:type 'number))
     (b   (:type 'boolean))>
```

Toutefois à des fins pratiques, on peut omettre de noter la facette *:type*. Ce raccourci d'écriture n'est valable que pour les types. Par exemple :

```
<:type mon-type
     (:is-a :aggregated)
     (a   (f1 2)
          'number)
     (b   'boolean)>
```

Un type agrégé peut spécifier des contraintes localement sur chaque attribut ou globalement sur l'ensemble des attributs.

- `open/closed` `[contrainte]`

La contrainte *:closed* spécifie s'il est possible ou non d'ajouter de nouveaux attributs à la décoration. Par exemple le type *rectangle* est une fermeture sur les attributs top, left, bottom et right. Touts les attributs doivent avoir des valeurs entières.

```
<:type rectangle
           (:is-a :aggregated)
           (:closed t)
           (top integer)
           (left integer)
```

51 Nous avons présenté la notion de facette appliquée aux frames au § 4.3.1.

```
            (bottom integer)
            (right integer)>
```

Une décoration de type rectangle ne peut pas rajouter un nouvel attribut à sa définition. En outre, les valeurs de ses attributs doivent être entières.

```
<:decor rectangle1
     (:type 'rectangle)>
```

```
(get-value 'rectangle1.top)
> :indef

(add-value 'rectangle1 'top)
> t

(get-value 'rectangle1.top)
> :null

(set-value 'rectangle.top "hello")
> nil

(set-value 'rectangle.top 20)
> t

(get-value 'rectangle1.top)
> 20
```

Après la définition de la décoration, l'attribut *top* n'existe pas et une tentative de lecture rend le symbole *:indef*. Définir l'attribut *:top* pour *rectangle1* consiste à rajouter comme valeur le symbole *:top*. La lecture de cet attribut est maintenant possible mais comme aucune valeur particulière n'est portée par cet attribut, le symbole *:null* est rendu. Après une tentative infructueuse d'affectation d'une valeur de type lexical, on affecte à l'attribut une valeur de type entier.

- exclusifs/non-exclusif

Comme pour les types énumérés, il est possible d'exiger de ne pouvoir instancier qu'un seul attribut à la fois. Si cette contrainte n'est pas définie, le type est non-exclusif.

```
<:type t-gn
          (:is-a :aggregated)
          (:exclusive t)
          (masc  integer))
          (fem  integer))
          (neut  integer))>
```

- obligation locale et générale

Il est possible d'exiger la présence de tous les attributs définis ou seulement de certains d'entre eux. Cette contrainte (qu'elle soit générale ou locale) est contradictoire avec la contrainte générale d'exclusivité et éliminera cette dernière en cas de conflit.

```
<:type rectangle-2
            (:is-a aggregated)
            (:obl t)
            (top  integer)
            (left integer)
            (bottom integer)
            (right integer)>
```

Le type *rectangle-2* n'est plus fermé mais rend obligatoire l'instanciation des quatre attributs. Une définition équivalente consistera à rendre obligatoire chaque attribut.

```
<:type rectangle-3
            (:is-a :aggregated)
            (top   (:type integer)
                   (:obl t))
            (left  (:type integer)
                   (:obl t))
            (bottom  (:type  integer)
                     (:obl t))
            (right (:type integer)
                   (:obl t))>
```

- contraintes libres

Il est possible de définir des contraintes plus complexes à partir de formules. Les contraintes peuvent porter sur les co-occurences de traits ou bien sur les valeurs des traits.

Par exemple, il est possible d'exiger que deux attributs ne puissent être simultanément instanciés :

```
<:type mon-type
          (:is-a :aggregated)
          (:exclusive :faux)
          (:contraints '(  (lambda (x y)
                           (exlcusive-or
                             (exists-p (make-path !decor x))
                             (exists-p (make-path !decor y))))
                           ('f1 'f2))
          (f1 :top)
          (f2 :top)
          (f3 :top)>
```

mon-type définit trois attributs (*f1, f2, f3*) dont les valeurs acceptables sont quelconques. La contrainte globale spécifie un *ou exclusif* pour l'existence de deux chemins dans la décoration courante (défini par la méta-variable !decor). Les arguments fournis à la formule définissent les chemins !decor.f1 et !decor.f2.

Composition et Manipulation de types

Une alternative à l'ajout de contrainte est la composition de types. Le nouveau type est le résultat de l'évaluation d'une fonction sur plusieurs types. Ce calcul peut être statique ou dynamique. Un type défini dynamiquement verra ses contraintes mises à jour automatiquement si les types à la base de sa définition sont modifiés. Un type défini statiquement est fixé une fois pour toutes (il peut toutefois être modifié par la suite mais aucune mise à jour n'aura lieu automatiquement). Certaines fonctions pour la composition de types sont issues d'ARIANE X [Boitet & al. 1982].

`disjunct (type+)` `[fonction]`

La fonction de base de la composition de types est la disjonction de types. Il est possible de construire un type t qui spécifie qu'il accepte de valeurs de type t1 ou t2 ou t3 ou n'importe quel autre type.

```
(disjunct :number :lexical)
```

```
> <:type
     (:is-a '(:or :number :lexical))>
```

Toute décoration de ce type pourra porter une valeur numérique ou lexicale.

La manipulation de types n'est en général possible que pour des types appartenant au même type de base (booléen, lexical, symbole, nombre, énuméré, liste et agrégé).

- `exclusive (enumerated) → enumerated` `[fonction]`
  `exclusive (aggregated) → aggregated` `[fonction]`

La contrainte *:exclusive* des types agrégés (ou énumérés) est mise à la valeur *:vraie.* Considérons comme base de nos exemple le type *t-gn* suivant :

```
<:type t-gn
          (:is-a :aggregated)
          (:exclusive :faux)
          (masc integer)
          (fem integer)
          (neut integer)>
```

alors :

```
(exclusive 't-gn)
> <:type
          (:is-a :aggregated)
          (:exclusive :vrai)
          (masc (:type integer))
          (fem (:type integer))
          (neut (:type integer))>
```

La fonction inverse *non-exclusive* force l'attribut *:exclusive* à la valeur *:faux*.

- `close (aggregated) → aggregated` `[fonction]`

La contrainte *:closed* des types agrégés est mise à la valeur *:vraie.* La fonction inverse *open* force la contrainte *:close* à la valeur *:faux*.

Par exemple, si on a :

```
<:type t-gn
          (:is-a :aggregated)
          (:closed t)
          (:exclusive :faux)
          (masc integer)
          (fem integer)
          (neut integer)>
```

alors :

```
(exclusive 't-gn)
> <:type   (:is-a :aggregated)
          (:closed t)
          (:exclusive :vrai)
          (masc (:type integer))
          (fem  (:type integer))
          (neut (:type integer))>
```

- `Eventail (aggregated) → aggregated [fonction]`

Soit T un type agrégé. L'éventail de T est un type agrégé non-exclusif T' obtenu en remplaçant tous les types des attributs par leurs équivalents non-exclusifs. Les noms des attributs sont préservés.

Par exemple, considérons le type suivant :

```
<:type subn
          (:is-a :enumerated)
          (:allowed-values '(nc np))>

<:type decam
          (:is-a :aggregated)
          (:exclusive :vrai)
          (k   <:type (:is-a :aggregated)
                      (:exclusive :vrai)
                      (k0 :boolean)
                      (n subn)
                      (v :boolean)> )
          (gnr <:type   (:is-a :aggregated)
                        (:allowed-values '(masc fem neu))> )
          (nbr <:type   (:is-a :aggregated)
                        (:exclusive :faux)
                        (:allowed-values '(masc fem neu))> )
          (ul  (:type :lexical)
               (:obl :vrai))>
```

```
(eventail 'decam)

<:type
     (:is-a :aggregated)
     (:exclusive :faux)
     (k    <:type  (:is-a :aggregated)
                   (:exclusive :faux)
                   (n    (:type (non-exclusive 'subn))
                         (:interpretation 'formula))
                   (v :boolean)> )
     (gnr <:type   (:is-a :aggregated)
                   (:exclusive :faux)
                   (:allowed-values '(masc fem neu))> )
     (nbr <:type   (:is-a :aggregated)
                   (:exclusive :faux)
                   (:allowed-values '(sin plur))> )
     (ul   (:type :lexical)
           (:obl :vrai))>
```

Par défaut, un type est non-exclusif (*:exclusif* vaut *:faux*). Si le type lié à un attribut est un symbole d'un type auquel la contrainte :exclusif pourrait être appliquée (c'est à dire des types énumérés et aggrégés) alors le nouveau type de l'attribut est l'ancien type rendu non-exclusif.

La forme :

```
(n   (:type (non-exclusive 'subn))
     (:interpretation 'formula))
```

spécifie que l'attribut *n* a le type *subn* forcé de manière non-exclusive comme type. Il s'agit d'une formule, car on souhaite une définition dynamique.

Si le type d'un attribut est une valeur de type, cette valeur est remplacée par son éventail.

- `Spectre (aggregated) → aggregated [fonction]`

Il est souvent souhaitable d'éviter le caractère obligatoire d'un attribut. La fonction *spectre* permet de supprimer tous les comportements obligatoires d'une définition de type.

Soit T un type agrégé. Le spectre de T est le type agrégé T' obtenu par la suppression de toutes les contraintes *obl* (globale et locale). Les noms d'attributs sont préservés.

```
(eventail 'decam)

> <:type
     (:is-a :aggregated)
     (:exclusive :faux)
     (k    <:type  (:is-a :aggregated)
                   (:exclusive :faux)
                   (n    (:type (non-exclusive 'subn))
                         (:interpretation 'formula))
                   (v :boolean)> )
     (gnr <:type   (:is-a :aggregated)
                   (:exclusive :faux)
                   (:allowed-values '(masc fem neu))> )
     (nbr <:type   (:is-a :aggregated)
                (:exclusive :faux)
                (:allowed-values '(sin plur))> )
     (ul   :lexical)>
```

- `Enveloppe (aggregated) → aggregated [fonction]`

La fonction enveloppe est la combinaison des deux opérateurs précédents.

Soit T un type agrégé. L'enveloppe de T est le type agrégé T' défini comme le spectre de l'éventail de T (ou comme l'éventail du spectre, les deux opérations étant commutatives).

```
(enveloppe 'decam)

> <:type
     (:is-a :aggregated)
     (:exclusive :faux)
     (k    <:type  (:is-a :aggregated)
                   (:exclusive :faux)
                   (n    (:type (non-exclusive 'subn))
                         (:interpretation 'formula))
                   (v :boolean)> )
     (gnr <:type   (:is-a :aggregated)
                   (:exclusive :faux)
                   (:allowed-values '(masc fem neu))> )
     (nbr <:type   (:is-a :aggregated)
                (:exclusive :faux)
                (:allowed-values '(sin plur))> )
     (ul   (:type :lexical)
           (:obl :vrai))>
```

- `Union (enumerated+) → enumerated [fonction]`
  `Union (aggregated+) → aggregated [fonction]`

Cette fonction calcule l'union discriminée des attributs de plusieurs types agrégés. Cette fonction rend un résultat analogue à la classe définie par héritage multiple dans CLOS.

Par exemple, considérons le type t-gn défini précédemment et le type t-nb défini comme suit :

```
<t-gn
     (:is-a :aggregated)
     (:exclusive t)
     (sing (:type integer))
     (plur (:type integer))>
```

```
(union 't-gn 't-nb)

> <:type   (:is-a (:aggregated)
        (:exclusive t)
        (masc (:type integer))
        (fem  (:type integer))
        (neut (:type integer))
        (sing (:type integer))
        (plur (:type integer))>
```

La définition de ce type est purement statique car elle a été calculée à partir de t-gn et t-nb. Si les deux types t-gn et t-nb sont modifiés par la suite, le type résultat ne change pas.

Par contre, la définition suivante est dynamique :

```
(define! nil :type
     (:is-a  (union 't-gn 't-ng)
             (:interpretation :formula))

> <t-gn
     (:is-a  (union 't-gn 't-ng)
             (:interpretation :formula))>
```

La hiérarchie de type est alors :

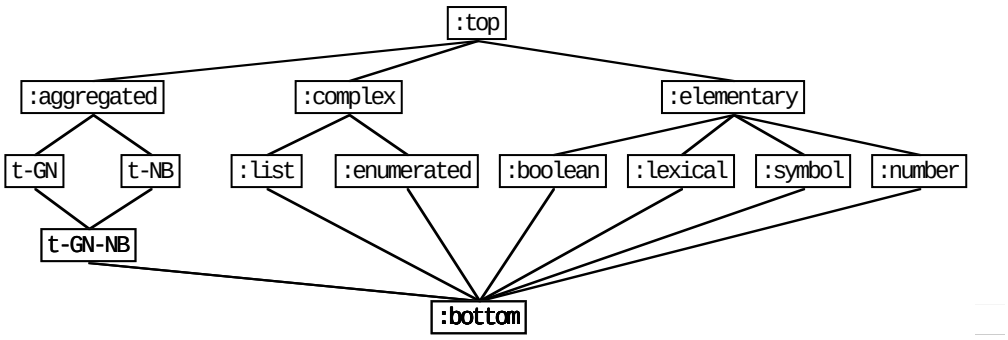


Figure 5.3 : Nouvelle hiérarchie de types après t-GN-NB = union(t-GN, t-NB).

L'héritage standard correspond à la fonction union. De plus, il permet, contrairement aux formules de pouvoir interroger un type sur ses parents. En cas de conflit entre les attributs de plusieurs types, la règle "le premier qui a parlé a raison" s'applique. Pourt l'héritage, nous introduisons un ordre décroissant de la gauche vers la droite (ordre de lecture). Cet ordre est celui de CLOS.

- `Intersection (enumerated+)` → `enumerated` `[fonction]`
  `Intersection (aggregated+)` → `aggregated` `[fonction]`

Cette fonction produit un type dont les attributs sont les attributs communs entre les attributs de types arguments (indépendamment du type des arguments).

```
<:type t-gn
     (:is-a :aggregated)
     (:exclusive t)
     (sing integer)
     (plur integer)>

<:type tgn-nb
       (:is-a (:aggregated)
       (:exclusive t)
       (masc :boolean)
       (fem  :boolean)
       (neut :boolean)
       (sing :boolean)
       (plur :boolean)>
```

```
(intersection 't-gn 't-gn-nb)

> <:type
     (:is-a :aggregated)
     (:exclusive t)
     (sing integer)
     (plur integer)>
```

Les contraintes globales sont celles du premier type argument. Les contraintes locales de chaque attribut sont préservées en cas de conflit.

- `difference (enumerated+) → enumerated` `[fonction]`
  `difference (aggregated+) → aggregated` `[fonction]`

```
(difference 't-gn 't-gn-nb)

> <:type
     (:is-a :aggregated)
     (:exclusive t)
     (masc boolean)
     (fem  boolean)
     (neut boolean)>
```

5.2.2. Décorations

Une décoration est une structure consituée de traits. Un trait est un couple attribut-veleur. Les traits ne sont jamais réentrants, ce qui permet d'assimiler systématiquement une décoration à un arbre. La réentrance (ou la co-indexation) est simulée à l'aide de valeurs particulières, les références (parfois appelées index) qui sont une généralisation des étiquettes.

Valeurs

Une valeur peut être atomique, complexe, ou aggrégée. Une valeur atomique est :

- un des symboles : *:vrai*, *:faux*, *:null* ou *:indef* ;
- un nombre ;
- une chaîne de caractères ;
- un symbole ;
- une formule.

Une formule est soit un couple (lambda-expression/liste de paramètres), soit une simple s-expression. Par exemple, les formules suivantes permettent toutes les deux de calculer la hauteur du *rectangle1 :*

```
'(   (lamdba (x y) (- (get-value x) (get-value y))
     ('rectangle1.bottom 'rectangle1.top) )

'(- (get-value 'rectangle1.bottom) (get-value 'rectangle1.top))
```

Ces deux notations, bien qu'aboutissant au même résultat, ne sont pas complètement équivalentes. En effet, du point de vue de la manipulation dynamique de la définition des formules, la première notation est plus pratique. Des fonctions permettent de changer la valeur des paramètres passés à la lambda expression ou de remplacer la fonction par une autre. Pour manipuler la seconde notation (changer la valeur de ses paramètres ou la fonction), il faut effectuer des transformations relativement lourdes sur la liste qu'est la s-expression.

Une valeur complexe est une liste de valeurs. Une valeur agrégée est une décoration.

Décoration atomique

Une décoration atomique ne dispose que d'un seul trait prédéfini *valeur*. Par exemple :

```
(define! pi   :decor
           (:value 3.14159)
 )
> <:decor   (:is-a :decoration)
           (:value 3.14159)>

(get-value pi)
> 3.14159

(get-value pi.valeur)
> 3.14159
```

Pour une décoration atomique, les deux notations sont équivalentes.

Les opérations de base pour les décorations atomiques consistent à écrire et à lire la valeur.

```
set-value (decor value)                                    [fonction]
get-value (decor)                                          [fonction]
```

Décorations complexes

Une décoration complexe se différencie d'une décoration atomique en ce que son trait valeur peut valoir d'une liste de valeurs. Les décorations dont le type est complexe (liste ou énumération) sont dites décorations complexes.

Les opérations *add-value* et *remove-value* sont disponibles pour ce type de décorations.

```
add-value (decor value)                                    [fonction]
remove-value (decor value)                                 [fonction]
```

Décorations agrégées.

Une décoration complexe peut avoir plusieurs traits quelconques. Les fonctions *set-value* et *add-value* ont des effets légèrement différents selon les paramètres.

Si l'argument *decor* de *set-value* est une décoration aggrégée, un nouvel attribut correspondant à la valeur (*:value*) est alors créé. Les valeurs permises sont les symboles, les chaînes et les nombres. Pour les décorations agrégées, si l'argument *decor* est une décoration, la fonction *add-value* a le même effet que *set-value.*

```
<:decor d   (f1 "hello")>
```

```
(set-value 'd 'f2)
> t
(get-value 'd.f2 )
> null
```

Si l'argument *decor* de *set-value* désigne un attribut, la valeur de cet attribut est changée.

```
<:decor d   (f1 "hello")
          (f2 1)>
```

```
(set-value 'd.f2 "world")
> t
(get-value 'd.f2)
> "world"
```

Typage

Si une décoration est typée, ses valeurs doivent respecter le type. Pour créer une décoration instance d'un type, il suffit de spécifier l'attribut *type*. L'instance est initialisée conformément au type. Les attributs obligatoires sont automatiquement créés avec leur valeur par défaut, l'attribut type est spécifié, etc.

Si une décoration est atomique son attribut *:value* est spécifié à l'instanciation.

```
(define! mon-nombre   :decor
          (:type :number)
 )
> <:decor  mon-nombre
          (:is-a :decoration)
          (:type :number)
          (:value)>
(set-value 'mon-nombre "hello")
> nil
(get-value 'mon-nombre )
> :null
(set-value 'mon-nombre 5)
> t
(get-value 'mon-nombre )
> 5
```

Une décoration agrégée est spécifiée de façon minimale par rapport à son type. Par exemple, considérons un type de rectangle très contraint :

```
<:type colored-rectangle
           (:is-a :aggregated)
           (:exclusive :faux)
           (top   (:type integer)
                  (obl t))
           (left  (:type integer)
                  (obl t))
           (bottom  (:type  integer)
                    (obl t))
```

```
                (right (:type integer)
                       (obl t))
                (color (:type <:type
                                (:is-a :enumerated)
                                  (:exclusive t)
                                  (:allowed-values '(red blue green))>
                       (:default-value 'blue))>
```

```
(define! mon-rectangle :decor
        (:type 'rectangle)

> <:decor mon-rectangle
           (:is-a :decoration)
           (:type 'rectangle)
           (top :null)
           (left :null)
           (bottom :null)
           (right :null)
           (color 'blue)>
```

Les quatre premiers attributs (*top, left, bottom* et *right*) sont obligatoires mais n'ont pas de valeur par défaut. Ils sont donc initialisés à *:null*.

Contraintes locales

Les types permettent de factoriser des contraintes globales ou locales sur les attributs. Il est aussi possible de définir ces mêmes contraintes au niveau des décorations. En cas de conflit avec les contraintes prévue par le type, les contraintes locales sont prioritaires.

Par exemple, la décoration *mon-rectangle* est une décoration ouverte, car elle instancie localement la contrainte *:closed* à faux.

```
<:decor mon-rectangle
       (:is-a :decoration)
       (:type 'rectangle)
       (:closed :faux)>
```

Il est possible d'ajouter une contrainte locale avec la fonction *set-constraint*. La lecture d'une contrainte se fait avec *get-constraint.*

```
set-constraint (decor [attribut] nom valeur)                [fonction]
get-constraint (decor [attribut] nom)                       [fonction]
```

Si l'attribut n'est pas spécifié, la contrainte est globale.

```
(set-constraint mon-rectangle :close :vrai)
> t

mon-rectangle
> <:decor mon-rectangle
        (:is-a :decoration)
        (:type 'rectangle)
        (:closed :vrai)>
```

Héritage

L'attribut spécial *:is-a* (que nous avons déjà rencontré dans les exemples ci-dessus) permet d'établir une hiérarchie d'héritage entre décorations et entre types. Les valeurs valides pour ce traits sont :

- un symbole ;
- une liste de symboles dans le cas de l'héritage multiple ;

Une définition de décoration peut contenir le nom d'une ou de plusieurs autres décorations comme valeur de l'attribut :is-a. Par exemple :

```
<:decor d1
       (:is-a :decoration)
       (f1 5)>

<:decor d2
       (:is-a :decoration)
       (f2 6)>

(define! d :decor
       (:is-a '(d1 d2)))

> <:decor d
       (:is-a '(d1 d2))>

(get-value 'd.f1)
> 5

(set-value 'd1.f1 10)
> t

(get-value 'd.f1)
> 10
```

Il est possible de construire statiquement des décorations à partir d'autres décorations de la même façon qu'on construit des types.

Interprétation des valeurs

Un attribut peut avoir (en plus de *valeur*) une facette *:interprétation* qui peut être instanciée avec trois valeurs :*'direct*, *'reference* et *'formula*. Ces trois valeurs d'interprétation ont les significations suivantes: interprétation de valeurs

- 'direct

La s-expression valeur de la facette valeur n'est pas interprétée.

- 'reference

La s-expression valeur de la facette valeur est considérée comme une référence à un autre objet défini dans DÉCOR. Seules les valeurs de type référence peuvent être interprétées comme une référence.

- 'formula

La s-expression valeur de la facette valeur du trait considéré est évaluée. Les meta-variables acceptables dans les formules sont : *!decor, !type, !slot.*

La fonction *set-interpretation* permet d'affecter une interprétation à un attribut. La fonction *get-interpretation* retourne l'interprétation de l'attribut.

```
set-interpretation (decor [attribut] interpretation)      [fonction]

get-interpretation (decor [attribut])                     [fonction]
```

Si l'attribut n'est pas spécifié, il s'agit de l'interprétation de l'attribut prédéfini *:value* (c'est l'interprétation d'une décoration atomique).

5.3. Recherche de la généricité : les prototypes comme paradigme de base

Avec DÉCOR, la recherche de la généricité passe par une forte stratification. En effet, nous avons défini un langage à partir duquel il est aisé d'implémenter plusieurs systèmes de décorations (parmi ceux présentés au chapitre 4, par exemple). Le système pourrait être utilisé tel quel mais les risques de retomber dans les travers de METAL sont importants, car DÉCOR est un noyau ouvert sur LISP. Il est donc nécessaire de définir une syntaxe externe, destinée aux linguistes qui "ferme" le langage (voir les annexes).

La recherche de la généricité requiert une ouverture partielle de la boîte noire qu'est la définition du langage. Dans [Kiczales & al. 1992], cet aspect essentiel d'une analyse visant à la généricité (et aussi à l'extensibilité) consistait à "rendre la boîte noire (presque) transparente"[52].

5.3.1. Les frames comme noyau

L'implémentation de DÉCOR se base sur le gestionnaire de frames GFP. Quelques détails liés à cette implémentation sont donnés dans le chapitre suivant.

Ce choix a été guidé par un désir de souplesse. On a deux niveaux (prototypes puis types/décorations) avec de moins en moins de liberté et de plus en plus de méthodes et de classes spécialisées pour le génie linguiciel.

Pour simplifier, nous avons choisi de faire correspondre directement la notion de prototype avec celle de frame. Les décorations et les types ne sont que des catégories particulières de frames. Toutes les opérations applicables sur les types et les décorations sont applicables sur les frames.

Par exemple, nous pouvons avoir la frame suivante :

```
<:frame geta
       (:is-a 'laboratory)
       (member-of 'imag 'cnrs)
       (location   (city 'grenoble)
                   (country france))>
```

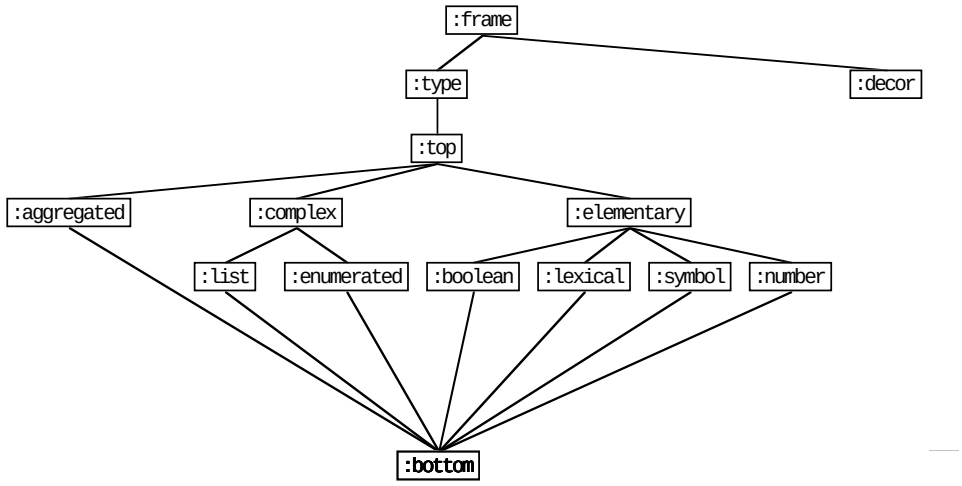


Figure 5.4 : Hiérarchie de frames dans DÉCOR.

52 "make the blackbox (almost) transparent"

5.3.2. Extensibilité et généricité

Nous avons vu qu'il était possible de définir des contraintes générales en spécifiant globalement ou localement une contrainte *:constraint,* qui doit être une formule. L'argument *:constraint* passé à la fonction *define!* permet de définir de nouveaux types de contraintes.

Les contraintes définissables par l'utilisateur sont des tests booléens sur l'état de la frame. Toute modification de la frame entraînera un test qui invalidera la modification si le test échoue. Un protocole permet de définir un certain nombre de fonctions utilitaires pour les frames. Ce protocole est inspiré du MOP, car il manipule aussi des informations sur les frames et non seulement les informations que portent les frames.

Une fonction générique permet de récupérer des informations sur les frames :

```
get-object-data (frame selecteur)                [fonction générique]
```

Ces informations retournées par le sélecteur ne peuvent pas être modifiées directement. Elles sont mises à jour si la frame est modifiée.

Les valeurs possibles pour *selecteur* sont :

- :feature-list — liste de tous les attributs spécifiés ;
- :predefined-feature-list — liste des attributs commençant pas un ":" spécifiés pour cette frame ;
- :user-defined-feature-list — liste des attributs non prédéfinis spécifiés pour cette frame ;
- :date-creation

…

Par exemple, on peut définir une contrainte globale spécifiant que tous les attributs doivent avoir une valeur numérique ou nulle :

```
(define! all-numerics :constraint
   (:arguments (test :boolean))
   (:object-category :decor)
   (:daemons :if-added)
   (:check-constraint-method
         (object object-category constraint-name args)
         ((if (first args)
                (or-list
                  (mapcar
                        #'(lambda (x)
                           (or (eql (get-value (make-path object x)) :null)
                               (number-p (get-value (make-path object x))))
                        (get-data-object object :user-defined-feature-list))
                 )
                t))
> <:constraint all-numerics
         …)
```

La contrainte ne demande qu'un seul argument booléen. Elle s'applique sur des décorations (ce qui veut dire qu'elle ne s'appliquera pas sur les types de décorations, mais qu'elle peut bien sûr être portée par des types). Elle s'applique lors de l'écriture d'une valeur.

La formule associée à la valeur de la contrainte *:all-numerics* applique un *ou logique*[53] à une liste de valeurs booléennes obtenue par application d'un test sur la valeur de chaque attribut de la frame. Il est pour cela nécessaire de récupérer la liste des attributs et pour chacun d'eux de construire le chemin !frame.*attribut.* On ne calcule cette valeur que si l'argument de cette contrainte est vrai (valeur t).

```
(define! ma-frame :frame
     (:is-a :frame)
     (all-numerics t))
> <:frame ma-frame
     …>

(add-value 'ma-frame 'f)
> t

(get-value 'ma-frame.f)
> :null

(add-value 'ma-frame.f "hello")
> nil

(add-value 'ma-frame.f 5)
> t
```

On s'aperçoit que ce mécanisme de création de contraintes est tout à fait généralisable pour créer des modèles d'attributs. De ce point de vue, il est possible de mettre en place un mécanisme permettant de déclarer à l'avance des attributs, comme dans le système MÉTAL[54]. Ce mécanisme permet donc de factoriser de l'information. Les contraintes sont évaluées en séquence afin de vérifier que la frame les respecte toujours bien.

On peut définir des attributs avec la fonction *define!* :

```
(define! color :feature
   (:type color-type)

> <:feature color
        …>
```

Les informations associables à un attribut sont les contraintes locales présentées précédemment.

La définition à partir de frames nous semble donc suffisamment générique pour permettre un certain nombre d'extensions intéressantes. Nous avons défini quatre catégories de frames dont les comportements et les paramètres de définition sont spécifiques.

---

53 La fonction *or-list* est équivalente à la macrofonction Lisp *or*. On peut cependant lui appliquer une valeur évaluable comme paramètre. Cette fonction est définie comme suit :

```
(defun or-list (boolean-list)
     (when boolean-list
       (if (car boolean-list)
         (car boolean-list)
         (or-list (cdr boolean-list)))))
```

54 Nous avions critiqué cette fonctionnalité de MÉTAL qui consiste à déclarer les attributs (à l'aide de defeat). En effet, dans MÉTAL c'est la seule manière de déclarer des attributs et elle pose le problème majeur d'empêcher des structures différentes d'avoir des attributs différents de même nom.

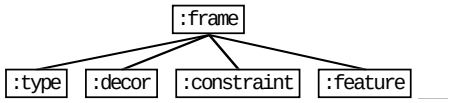


Figure 5.5 : Extension des catégories de frames de DÉCOR.

Nous verrons dans le chapitre suivant qu'il est tentant de définir un méta-protocole permettant de définir génériquement de nouvelles catégories de frames. Il n'est pas irréaliste de définir un tel protocole, mais la tâche n'est pas aisée.

Chapitre 6

Vers des composants génériques

DÉCOR a été l'objet de deux tentatives d'implémentation. La première a consisté en une définition directe des différents objets du langage à partir de CLOS. Nous avons rencontré un certain nombre de problèmes non résolus, et en particulier obtenu une faible généricité. La seconde tentative a consisté à écrire un gestionnaire de prototypes (GFP) et à l'utiliser comme base de l'implémentation. L'introduction de ce niveau intermédiaire a permis d'obtenir une grande généricité, et des possibilités d'extension. On pourrait enfin envisager une troisième approche, avec le MOP [Kiczales & al. 1991].

Les problèmes liés à la gestion des classes ont été l'objet d'études (par exemple avec [Nierstrasz & Pintado 1990] pour une application générale). L'étude de la spécialisation et de la réalisation dans [Girod 1991] a été conduite pour différentes approches à la programmation par objet, dans la perspective du génie logiciel. [Akasaka & al. 1989] propose une boîte à outils pour le TALN. Toutefois, peu a été fait en ce qui concerne l'implémentation d'un langage de programmation générique.

Un langage générique comme DÉCOR peut être utilisé de plusieurs façons. Il se situe en effet à un niveau intermédiaire par rapport aux différents formalismes présentés au chapitre 4. Les extensions et les restrictions sont toutes deux possibles, éventuellement en même temps. Il est toutefois important de ne pas sortir du modèle défini par les prototypes. Il serait par exemple absurde de définir un langage de classes équivalent à CLOS à partir de DÉCOR. Enfin, on peut se demander si la multiplicité des surcharges qui peuvent être introduites ne va pas ruiner l'ensemble du langage.

6.1. Leçons d'une double implémentation

6.1.1. Première tentative à base de classes

La première analyse et implémentation de DÉCOR (que l'on appellera dans la suite DÉCOR$_{clos}$) se basait exclusivement sur CLOS (voir figure 6.2 et 6.3 pour les hiérarchies de classes).

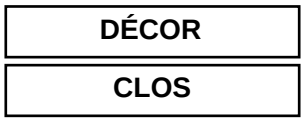


Figure 6.1 : Stratification de DÉCOR$_{clos}$.

Problèmes d'implémentation

De nombreuses classes ne devaient être instanciées qu'une seule fois, comme "top", "bottom" et les classes élémentaires. Les types élémentaires ayant des contraintes supplémentaires, étaient représentés sous forme d'instances de classes élémentaires avec des attributs particuliers. Ces attributs qui définissaient les types de contraintes était tous définis a priori. Une fonction interne *check-value* vérifiait, lors d'une affectation, l'adéquation d'une valeur par rapport à un type.

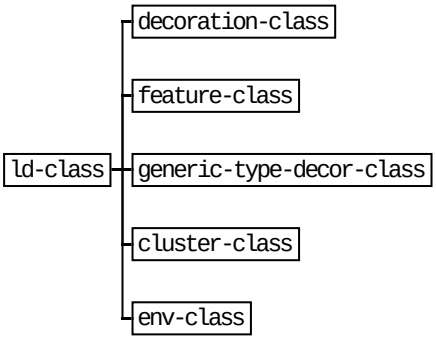


Figure 6.2 : Classes de bases de DÉCOR$_{clos}$.

Une telle architecture compromettait toute extensibilité et donc, dans une large part, la généricité. Nous l'avons vu, la généricité (qui est, nous le rappelons, la possibilité d'utiliser un outil dans des contextes divers et non prévus a priori) tire beaucoup de l'extensibilité (qui est la possibilité de définir de nouveaux objets à partir des objets existants). L'extensibilité n'était possible que par modification du code existant, et pas par surcharge ou spécialisation.

Par exemple :

```
(defclass number-type-class (elementary-type-class)
   (…
   (range   :accessor range
            :initargs '(:negative-infinite :positive-infinite))
    …))
> NUMBER-TYPE-CLASS

(setf number (make-instance 'number-type-class))
> NUMBER
```

D'autres instances de *number-type-class* pouvaient être créées et spécifiées avec une liste de deux valeurs pour *range*. Cependant, il n'était pas possible de créer simplement un sous-type de *number* avec un nouveau type de contrainte. Il fallait pour cela définir dynamiquement une nouvelle classe.

De nouvelles contraintes auraient pu être rajoutées à l'aide d'un sous-classage. Mais les classes doivent être réalisées par un seul et unique développeur qui connaît en permanence la hiérarchie de toutes les classes déjà définies. Dans ce cas, l'utilisation de l'héritage multiple (pour combiner deux contraintes développées séparément) amène rapidement à une hiérarchie de classes peu cohérente et difficile à comprendre.

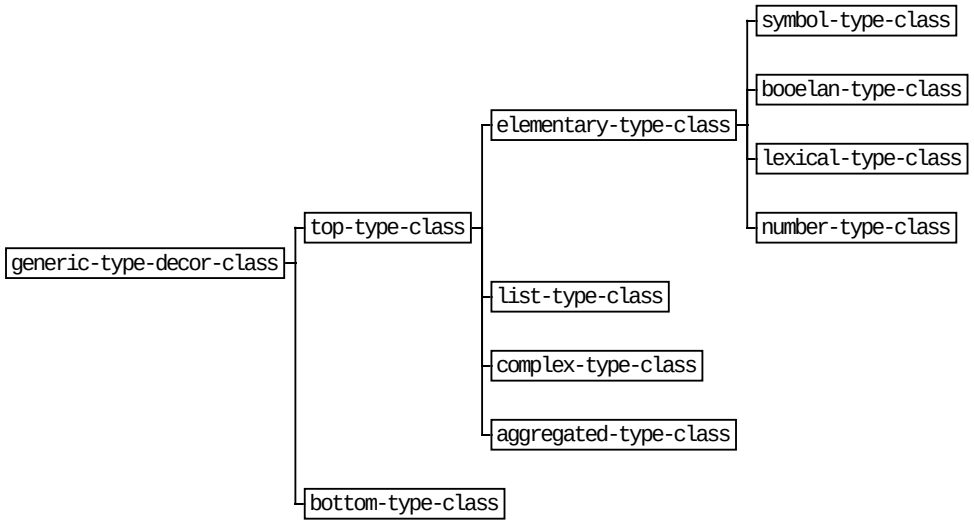


Figure 6.3 : Hiérarchie de classes de types de DÉCOR$_{clos}$.

Avec cette implémentation, nous arrivons non pas à un langage avec une seule entité de base que serait la décoration (ou de façon plus générale le prototype), mais au mieux avec quatre entités : les décorations, les types de décoration, les classes de décorations et les classes de types de décorations.

C'est à l'opposé du but à atteindre.

Problèmes de modèle

Dans DÉCOR$_{clos}$, la notion de prototype n'existait pas dans l'implémentation. L'analyse était légèrement différente de celle présentée au chapitre 5. En effet, l'influence d'ARIANE était importante (trop sans doute), ce qui explique l'introduction des "formats" dans l'implémentation.

L'héritage entre décorations devait être simulé au niveau des instances de la classe de décorations. L'héritage n'était pas unifié en ce sens qu'il était divisé entre celui des classes CLOS et celui des décorations et des types. La définition des types de contraintes était câblée dans les classes. Une autre solution aurait consisté à en faire des instances d'une "constraint-class" mais cela n'aurait pu que repousser davantage le problème.

Le plus gros problème venait d'un manque de généricité et de la difficulté de choisir entre des représentations sous forme de classes ou des représentations sous forme d'instances.

On peut facilement implémenter un langage quelconque à partir d'un langage de classes (nous l'avons vu avec le GFP). Toutefois, il nous semble maintenant clair qu'il est "maladroit" de définir un langage fondamentalement basé sur des frames (ce qu'est DÉCOR) directement à partir d'un langage de classes.

6.1.2. Seconde expérience à base de frames

La seconde implémentation se base sur le modèle des langages à prototypes. Cette correspondance plus fine entre le modèle prévu par DÉCOR et l'implémentation est plus intuitive et simplifie la réalisation.

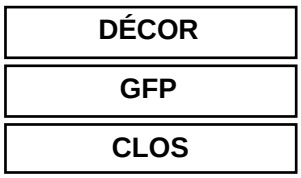


Figure 6.4 : Stratification de $DÉCOR_{frame}$.

Héritage

Nous devons tenir compte d'une double forme d'héritage : l'héritage entre décorations et l'héritage entre types.

Au sein de la même classe d'objets (décoration ou type), l'héritage prend la même forme. Il est nécessaire de calculer pour une frame la liste de précédence, c'est-à-dire la liste des ancêtres de la frame. Cette liste est triée conformément à l'ordre topologique de CLOS.

Pour une frame F héritant directement de $F_1 \ldots F_n$, la précédence locale est l'ensemble $((F\ F_1)\ (F\ F_2)\ \ldots\ (F\ F_n))$. L'interprétation est que F précède F_1 et F_2, etc. Le tri topologique ordonne un nombre arbitraire d'éléments en respectant la précédence locale. Une procédure de choix (tie-breaker) est utilisée quand il est nécessaire de sélectionner un élément parmi plusieurs éléments minimaux. Le tie-breaker se base sur l'ordre d'héritage des frames mères. Il est «nécessaire de sélectionner la frame qui a une sous-frame directe la plus à droite dans la liste de précédence construite jusqu'à maintenant»[55]. Pour cela, il suffit de consulter la liste de précédence partiellement construite de gauche à droite et de sélectionner le premier élément minimal présent parmi les superclasses des éléments de la liste de précédence.

La convention pour l'héritage des valeurs d'une décoration est que, pour une frame donnée et un triplet <champ facette vue> fixé, la valeur retenue est la première qui a été rencontrée dans le parcours du graphe d'héritage. Si, lors du parcours du graphe d'héritage, aucun triplet <champ facette vue> n'est trouvé, la valeur rendue est NIL.

Par exemple :

```
<:frame a
     (f1 1)>

<:frame b
     (f2 2)
     (spec "hello")>

<:frame c
     (:is-a (a b))
     (spec "world")
     (f3 4)>
```

L'accès à *c.f1* ou *c.f2* se fera dynamiquement par héritage. L'attribut *spec* est réécrit, *c.spec* retournera la valeur "world".

---

55 "[…] In this case select the one that has a direct subclass rightmost in the class precedence list computed so far" [Steele 1990].

Définition générique

La fonction *define!* est utilisée pour définir des objets. Il s'agit d'une macro fonction qui chapeaute une méthode dont la spécialisation se fait sur la catégorie d'objets.

```
define! (name categorie &key &allow-other-keys &rest)        [macro]
```

Si la valeur de l'argument *name* est NIL, un objet anonyme est créé. Cette macro se contente de rajouter des quotes devant les paires d'attributs/valeurs et de transformer en mots-clés les arguments dont les attributs sont connus.

```
(defmacro define! (name categorie &key &allow-other-keys &rest rest)
  (let ((args (normalize-arguments `,rest)))
    (make-objects categorie ,@args)))
```

```
make-object (categorie &key &allow-other-keys)  [fonction générique]
```

Cette méthode est discriminée sur la catégorie. Par exemple, pour la création d'une décoration nous avons :

```
make-object ((categorie (eql :decor)) &key &allow-other-keys)
                                                          [méthode]
```

```
(defmethod make-object ((categorie (eql :decor)) &key &allow-other-keys
                                          &rest rest)
  ;; créer l'instance de l'objet
  (let ((object (make-frame …) …)
  ;; discriminer les mots-clés potentiellement présentes
  (handle-keywords object categorie rest)
  ;; initialiser correctement la frame pour en faire une décoration valide
  …
 ))
```

Ce type de définition générique ne pose pas de problème particulier. Il est toutefois intéressant de se pencher sur le gestion des mots-clés. Ils sont toujours optionnels pour l'appel de la méthode. Par contre, ils peuvent être absolument obligatoires pour la création de l'objet. De plus, si on fixe les mots possibles après *&key*, on se prive de toute extensibilité (pour tenir compte d'un nouveau mot-clé, il faudra modifier le code et ce n'est pas ce que nous souhaitons).

Dans les méthodes *make-object*, nous récupérons la liste des mots-clés et appelons pour chacun d'eux une méthode (*make-object-handle-keyword*) qui doit également être spécialisée.

```
make-object-handle-keyword (object categorie keyword value)
                                                [fonction générique]
```

Par exemple, pour le keyword *:type* d'une décoration, nous avons la spécialisation suivante :

```
make-object-handle-keyword (object
         (categorie (eql :decor))
         (keyword (eql :type)) value)                        [méthode]
```

Ce type donne satisfaction à condition que les mot-clés soient raisonnablement indépendants. L'ordre d'appel de *make-object-handle-keyword* sur les mots-clés correspond à l'ordre d'écriture dans *define!*.

Le handler est appelé par *handle-keywords.* Bien que cette méthode ne soit l'objet d'aucune modification, il est nécessaire de connaître ce mécanisme d'appel, car *make-object* appelle cette méthode.

Nous constatons plusieurs choses. D'abord, ce mécanisme donne une bonne généricité et extensibilité dans la définition de nouveaux objets. Mais, cela se paye. Il est obligatoire de respecter des contraintes dans l'écriture de nouvelles méthodes spécialisées. Est-il nécessaire de définir une nouvelle méthode qui permettrait à au développeur de n'écrire que ce qui est strictement nécessaire pour définir une spécialisation de *make-object* ? Peut-être, mais cela augmenterait encore davantage la complexité du protocole ainsi que les difficultés de compréhension de l'ensemble des mécanismes. De plus, une méta-fonction ne présenterait en fait qu'un gain relatif, car il n'est pas possible de faire l'économie de l'écriture des spécialisations de *make-object-handle-keyword.*

Enfin, il est toujours possible (et parfois souhaitable) d'écrire une spécialisation de *make-object* qui n'appelle pas *handle-keywords* et traite directement un ensemble non extensible de mots-clés.

Interprétation de valeurs

L'interprétation de valeurs est une extension des frames. Ce mécanisme peut être utilisé indépendamment des décorations et des types de décorations. Il est possible de définir une interprétation de valeurs pour les différentes vues de la facette *value* d'un champ. L'implémentation est une combinaison de démons et de fonctions génériques.

Une vue :interpretation peut être définie pour une facette avec la fonction *set-interpretation.* La fonction *remove-interpretation* supprime toute interprétation particulière de la valeur (ce qui est presque équivalent à l'interprétation *:direct*).

```
set-interpretation (frame slot interpretation)  [fonction générique]
```

La fonction *set-interpretation* peut globalement être définie comme suit (certains mots-clés ne sont pas indiqués, comme *:view* qui permet de sélectionner une facette différente de *common,* la facette par défaut) :

```
(defmethod set-interpretation (frame slot interpretation)
   (frame-add-filler frame slot :if-accessed
         `(frame-get-interpreted-value  !frame !slot !facet !filler
                                        ,interpretation)))
```

```
frame-get-interpret-value (frame slot facet filler interpretation)
                                                    [fonction générique]
```

Cette fonction peut être définie comme suit:

```
(defmethod frame-get-interpreted-value (frame slot facet filler
                                        interpretation)
   (declare (ignore frame slot facet filler))
   (error "invalid interpretation ~a" interpretation))
```

avec la spécialisation suivante :

```
frame-get-interpret-value (frame slot facet filler
          (interpretation (eql :direct)))                    [méthode]
```

```
(defmethod frame-get-interpreted-value (frame slot facet filler
             (interpretation (eql :direct))
   (declare (ignore frame slot facet))
   filler)
```

La méthode non spécialisée pourrait être plus "tolérante" et ne pas être un cas d'erreur mais plutôt le cas par défaut (comme *:direct*). Nous avons ici fait le choix d'une programmation plus défensive.

Pour une interprétation donnée, il est cependant nécessaire d'effectuer un traitement particulier en fonction du type de données. Par exemple, pour l'interprétation :formula, nous avons prévu d'avoir deux formes distinctes acceptables : la liste "quotée" ou la paire lambda-list/argument.

```
(defmethod frame-get-interpreted-value (frame slot facet filler
             (interpretation (eql :formula))
   (declare (ignore frame slot facet))
   (interpret-value filler interpretation)
   )
```

La réalisation immédiate de la fonction *interpret-value* consiste en une analyse par cas selon la forme du filler. Par exemple, nous avons :

```
(defmethod interpret-value (value (interpretation (eql :formula)))
   (cond
      ((quoted-listp value) …)
      ((lambda-pair-p value) …)
      ( t (error "~a has an invalid form for interpretation ~a)
                  filler interpretation))
 ))
```

Cependant, il est impossible de définir de nouvelles formes valides pour l'interprétation *:formula* sans modifier ce code. Il faut pouvoir continuer à utiliser le mécanisme des fonctions génériques qui permet de sélectionner automatiquement la bonne méthode applicable. L'extensibilité est préservée, car une nouvelle forme acceptable pour l'interprétation de *:formula* se réalisera par une nouvelle spécialisation de la méthode *interpret-value*. Malheureusement, la spécialisation de méthode ne peut se faire en CLOS que par sélection de la classe de l'argument ou par égalité forte (eql) de l'argument avec une valeur fixée. Il n'est donc pas directement possible de spécialiser une méthode avec comme critère un prédicat qui doit être vérifié sur un des arguments.

Les raisons de l'absence d'un tel mécanisme peuvent s'expliquer. En effet, une telle spécialisation est très délicate à utiliser bien que son implémentation ne pose pas vraiment de problème. Par exemple, laquelle des deux méthodes suivantes :

```
(defmethod print-number ((number (pred #'even-p))) …)
(defmethod print-number ((number (pred #'(lambda (x) (< x 100))))) …)
```

devrait être sélectionnée pour l'appel suivant ?

```
(print-number 50)
```

Malgré ces problèmes potentiels, nous pensons que ce type de spécialisation peut être particulièrement utile dans le cadre de la définition d'un protocole généralisé, si on s'assure que les cas sont mutuellement exclusifs (il s'agit d'une partition avec le cas par défaut).

Contraintes

L'ensemble des catégories d'objets et de contraintes est implémenté à partir de démons. Un attribut privé *:category-constraints* a comme valeur une liste de contraintes qui doivent être vérifiées en permanence.

Pour une frame, un démon de chaque type (*:if-added, :if-needed, :if-accessed* et *:if-erased*) est associé à chaque attribut. Pour définir un démon, il suffit de rajouter un *filler* (une valeur sur un champ différent de *value* et une vue différente de *common*) dont la vue corresponde à une des quatre catégories.

Par exemple :

```
<:frame ma-frame
     (attribut1 5)>
```

```
(frame-add-filler 'ma-frame 'attribut1 :if-added
   '(print "attribut1 has been modified!"))
```

À chaque ajout d'une valeur pour l'attribut *attribut1*, le message sera affiché.

En pratique, le démon *:restriction* sera utilisé pour n'autoriser que certaines valeurs en fonction de l'ensemble des contraintes. Le démon appelle toujours le même prédicat *valid-value-p.*

Cependant, nous devons tenir compte de l'héritage des contraintes. C'est à nouveau *valid-value-p* qui a la charge de parcourir les hiérarchies de décorations et de types afin de déterminer l'ensemble des contraintes à appliquer. Cet ensemble ne contient qu'une seule contrainte de chaque type.

```
<:decor nombre1
     (:is-a :number)
     (:range 0 10)>

<:decor nombre2
     (:is-a :number)
     (:range -6 3)>

<:decor nombre
     (:is-a '(nombre1 nombre2))>
```

Le mécanisme (par défaut) d'héritage précise que la contrainte sélectionnée est la première rencontrée lors du parcours de la liste de précédence. Dans ce cas, la contrainte *:range* de nombre est celle de nombre1 *(:range 0 10)*.

On pourrait imaginer des possibilités pour effectuer des combinaisons de contraintes. Par exemple, l'intersection des intervalles pour range (on aurait, dans notre exemple, la contrainte résultante *(:range 0 3)*). Nous somme en train de travailler sur des extensions dans cette direction.

La définition d'une nouvelle contrainte implique la spécialisation d'une méthode permettant la vérification de cette contrainte pour une frame :

```
check-constraint (object object-category constraint-name args
          &key slot &allows-other-keys)          [fonction générique]
```

Par exemple, nous avons la spécialisation suivante pour *:range*:

```
(defmethod check-constraint (  (object frame)
                               (object-category (eql :decor))
                               (constraint-name (eql :name))
                               args
                               &key slot &allows-other-keys
 (let (  (min (first args))
         (max (second args))
         (value (frame-get-values object slot)))

   (and  (number value)
         (<= value max)
         (>= value min))
 ))
```

Il faut simplement vérifier si la valeur de l'attribut est bien un nombre compris dans l'intervalle.

La définition de nouvelles contraintes avec *define!* (voir §5.3.2) permet de générer automatiquement cette méthode sur la base du mot-clé *:check-constraint-method*. Ce mot-clé peut être omis, à condition que *check-constaint* soit correctement spécialisé par ailleurs.

Une fois de plus, afin d'assurer une bonne généricité, nous avons dû mettre en place un mécanisme relativement complexe. Toutefois, dans ce cas, la définition d'une nouvelle contrainte reste relativement simple, car il n'est pas nécessaire de connaître l'ensemble des séquences d'appels pour réaliser une nouvelle spécialisation.

6.1.3. Vers une troisième expérience

Le MOP ([Kiczales & al. 1991]) permet d'étendre les fonctionnalités de CLOS. Il est possible de définir de nouvelles métaclasses ayant des comportements particuliers. Quelques expériences d'utilisation du MOP ont déjà été faites avec, par exemple, les travaux de [Dvorak & Bunke 1993] qui visaient à définir un outil hybride de représentation de la connaissance dédié à la vision. L'ensemble de l'outil est développé sur la base de CLOS, et quelques expériences ont été tentées avec le MOP.

Quelques tentatives d'implémentation de DÉCOR à partir du MOP ont été amorcées, et bien qu'il soit trop tôt pour tirer des conclusions générales, il est d'ores et déjà possible de faire les remarques suivantes :

- il ne s'agit pas de définir directement DÉCOR sur le MOP ;

Ce serait refaire la même erreur qu'avec $\text{DÉCOR}_{\text{clos}}$. Le GFP doit toujours être à la base de DÉCOR, et c'est donc lui qu'il faudrait réimplémanter à partir du MOP (voir figure 6.5).

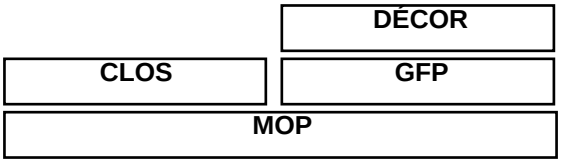


Figure 6.5 : Stratification d'un éventuel $\text{DÉCOR}_{\text{mop}}$.

- complexité ;

Les mécanismes du MOP sont complexes à comprendre. Ils ne sont pas non plus nécessairement toujours bien adaptés aux extensions que nous envisageons.

De plus, il nous semble important, au moins pour le moment, de "garder la maîtrise du code". Déléguer certains mécanismes au MOP peut, dans certains cas, être nuisible. Ce serait comme utiliser la récursivité du langage d'implémentation dans le développement d'un langage. Le contrôle est perdu.

- problème dû à l'instabilité du MOP ;

Le MOP est encore en cours de développement. Les diverses implémentations de CLOS ont intégré l'état actuel de la définition du MOP de façon très variable.

6.2. Vers quelle généricité ?

Le coût de la généricité se justifie-t-il ? Nous pensons que la définition d'un langage noyau pouvant servir de base à un large éventail de langages utilisateurs permet d'adopter une démarche exploratoire. En effet, il est possible pour l'informaticien de répondre plus facilement à de nouvelles demandes des linguistes.

Cette idée de prototypage est bien exposée dans [Krief 1992]. L'informaticien doit être en mesure de concevoir, d'implémenter et de tester différentes solutions répondant à un problème donné. Pour cela, il est nécessaire de fournir des outils qui permettent d'évaluer et de comparer la pertinence et l'adéquation des solutions proposées. Il s'agit donc de "programmation expérimentale" (ou exploratoire). Une telle approche nécessite une conception aussi générique que possible.

6.2.1. Famille de langages

On peut voir DÉCOR comme un système de frames auquel on a rajouté :

- trois types d'interprétations de valeurs ;
- un système de contraintes sur les valeurs associées aux traits (les types) ;
- plusieurs mécanismes de hiérarchie.

DÉCOR peut n'être utilisé que partiellement (par exemple en ignorant les types) afin d'obtenir un langage proche de KR [Guise 1993]. Le développement de GARNET a prouvé que KR peut être utilisé pour concevoir des éléments d'interface utilisateurs. Nous avons mené une expérience similaire, où certains des attributs de classes d'éléments d'interface sont liés à des décorations de DÉCOR.

DÉCOR est un langage extensible, son espace de comportement peut donc être étendu à d'autres objets ou fonctions. On fera la distinction entre l'espace de comportement propre qui correspond au langage avec sa définition actuelle, et son espace de comportement potentiel qui est défini par l'ensemble des extensions possibles (en utilisant les protocoles prévus à cet effet).

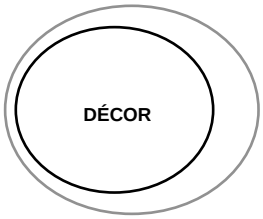


Figure 6.6 : Espaces de comportement propre et potentiel.

Modification du noyau

Définir un nouveau langage à partir de DÉCOR consiste à modifier son espace de comportement propre. Cela peut se faire par :

- Réduction ;
- Extension ;
- Développement.

La *réduction* consiste à ne choisir qu'un sous-ensemble des objets et des fonctions définis originellement. Cependant, cette sélection consiste à réduire l'accès aux fonctions et aux objets et non pas à les éliminer. C'est pourquoi il est alors nécessaire de définir un nouveau protocole plus réduit, qui utilise potentiellement l'ensemble des ressources du langage de départ.

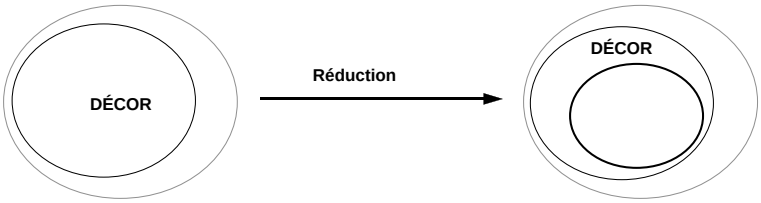


Figure 6.7 : Réduction de l'espace de comportement.

On aboutit ainsi à un langage dont les primitives constituent un sous-ensemble des primitives d'origine.

Par exemple, il est possible de définir un langage de décorations semblable à DÉCOR mais ne disposant pas de la modification dynamique des types. En annexe D, nous proposons un langage externe à partir du noyau DÉCOR. Ce langage n'est plus aussi dynamique. La modification des types est possible au niveau de l'interface utilisateur de l'environnement de développement lors de la mise au point, mais non au niveau du langage lui-même. Une sur-couche logicielle ne sélectionne qu'un sous-ensemble de fonctionnalités de DÉCOR.

L'*extension* consiste à définir de nouveaux comportements contenus dans l'espace de comportement potentiel. Cet espace est défini implicitement par les protocoles permettant l'extension. Donc, ces extensions ne sont faites que par utilisation plus complète des protocoles existants.

En pratique, il est rare de ne pas combiner l'extension avec une réduction. En effet, il est possible que certains des objets prévus par DÉCOR ne trouvent pas d'usage dans un nouveau langage. Ce nouveau langage définit alors des objets inconnus de DÉCOR, mais faisant partie de l'espace de comportement potentiel (c'est-à-dire définissables avec les protocoles d'extension).

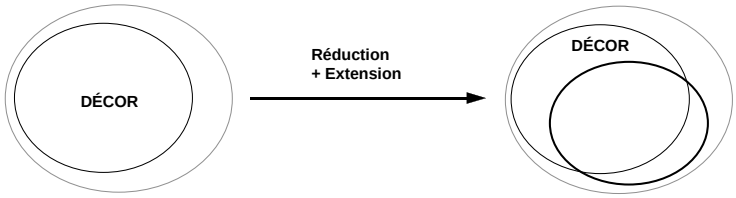


Figure 6.8 : Réduction et extension de l'espace de comportement.

Par exemple, un langage similaire à celui décrit en annexe D incluant les formats d'ARIANE ou les attributs de MÉTAL fait appel à la fois à la réduction et à l'extension.

Le *développement* de l'espace de comportement est une dérive du langage vers des comportements potentiellement non prévus. Il ne s'agit donc plus d'utiliser les protocoles d'extension mais d'effectuer un développement plus lourd.

Si un développement est nécessaire, cela peut signifier deux choses. D'une part que le langage n'est pas bien adapté pour fournir un base de départ vers le nouveau langage. Ou au contraire, qu'il s'agit bien d'un bonne base de départ, mais que les protocoles d'extension n'avaient pas été définis pour ce type d'extension.

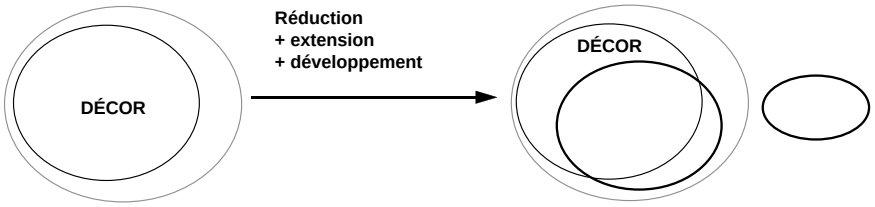


Figure 6.9 : Réduction, extension et développement de l'espace de comportement.

Par exemple, l'implémentation d'un langage comme LIFE à partir de DÉCOR nécessite au moins l'ajout d'un module de calcul des prédicats.

Plusieurs questions surgissent quant à la nature des langages dérivés.

- Doivent-ils et peuvent-ils être à leur tour génériques ?
- Doit-on systématiquement les associer à un langage externe ou doivent-ils rester des langages noyaux ?

Nous n'avons pas de réponses définitives à ces deux questions. À notre avis, il s'agit d'un compromis entre utilisation et effort de développement. La restriction pose un problème si elle doit être liée à la généricité, dans la mesure où il faut redéfinir un protocole complet qui est client du langage noyau et permet en plus l'extensibilité. La définition d'une syntaxe externe restreint l'extensibilité, mais elle est un excellent moyen d'effectuer la réduction.

Voyons maintenant dans quelle mesure il est possible, à partir de DÉCOR, de définir les différents systèmes de décorations définis au chapitre 5. On peut classer l'effort nécessaire selon la réduction, l'extension et le développement.

Les formalismes de décorations (et non pas de grammaires) de LFG, de GPSG et de HPSG constituent des réductions de DÉCOR. En effet, ces théories prévoient principalement des représentations linguistiques et des principes d'instanciation qui sont soit déjà assurés par DÉCOR (principe d'unicité), soit liés implicitement aux types de traits prévus par la théorie (principes de trait de tête, de pied et d'accord/contrôle). Il serait évidemment possible de définir un langage dérivé implémentant ces mécanismes, mais un tel langage ne définirait pas explicitement une théorie linguistique.

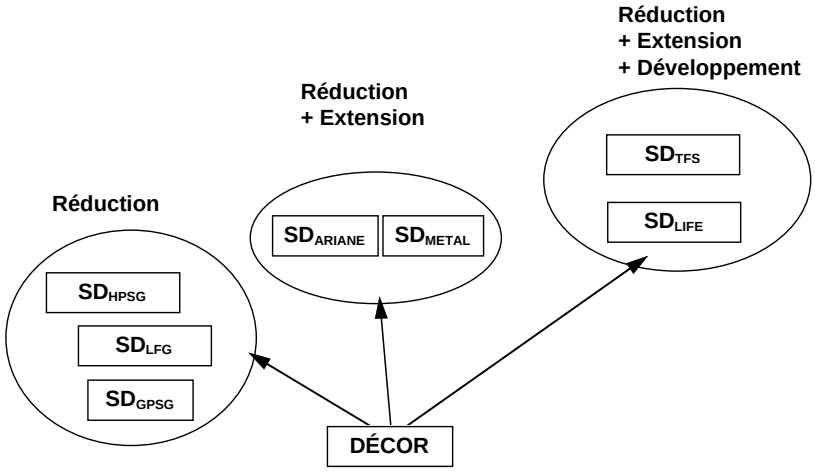


Figure 6.10 : DÉCOR comme base de développement de plusieurs systèmes de décorations.

La généricité de DÉCOR permet au développeur d'implémenter de multiples systèmes de décorations (existants ou à inventer) plus facilement qu'avec seulement des primitives de base d'un langage de programmation classique. Certains systèmes (ARIANE, METAL) ne demandent que de prendre un sous-ensemble de DÉCOR et de rajouter quelques extensions, d'autres nécessitent un développement plus poussé (TFS, par exemple).

Nous avons mené deux expériences concrètes qui ont consisté à définir à partir de DÉCOR des langages noyaux aussi proches que possible de ceux d'ARIANE et de MÉTAL. Certains aspects de ces extensions ont déjà été présentés.

6.2.2. Plusieurs catégories d'objets

À partir de DÉCOR, le développeur peut implémenter de nouvelles catégories d'objets. Le mécanisme de définition générique partage l'esprit du MOP, bien qu'il soit nettement moins abouti.

Les formats d'ARIANE

Le système de décorations d'ARIANE (présenté au §4.1.2.) définit des formats utilisés comme classes morpho-syntaxiques. Un format est une décoration constante et nommée dont chaque attribut a un nom qui correspond à une décoration déjà définie. Le type d'un attribut est le type de la décoration de même nom.

Par exemple :

```
<:decor a
       (:type number)>

<:decor b
       (:type boolean)>

<:decor c
       (:type lexical)>

<:decor f1
       (:is-a :format)
       (a 5)
       (b :vrai)
       (c "hello")>
```

Les trois décorations a, b et c sont définies. Le format f1 spécifie trois attributs a, b et c dont les types sont respectivement ceux des décorations a, b et c.

Pour être un format, les contraintes qui pèsent sur une décoration sont les suivantes :

- les noms des attributs définis doivent correspondre à des décorations définies ;
- les valeurs portées par les attributs doivent être valides pour les décorations correspondantes ;
- le format ne peut pas être modifié après sa définition ;

Commençons par la dernière contrainte. Il s'agit d'empêcher toute modification ultérieure de la décoration.

`:locked (boolean)` `[contrainte]`

Cette contrainte indique que l'objet ne peut plus être modifié. Il n'est donc accessible qu'en lecture. Seul l'attribut *locked* peut encore être accédé en écriture (afin de pouvoir autoriser à nouveau les modifications).

```
(define! :read-only :constraint
   (:arguments (s :boolean))
   (:object-category :decor)
   (:deamons :if-added :if-erased)
 )
(defmethod check-constraint (  (object frame)
                               (object-category (eql :decor))
                               (constraint-name (eql :locked))
                               args
                               &key slot &allows-other-keys
 (if (equal slot :locked)
   t
   (error "cannot add a value to a read-only decoration"))
 ))
```

Il est important d'autoriser l'écriture du champ qui contraint la valeur de la contrainte.

Les deux premières contraintes *(:existing-decorations-for-slot-name* et *:valid-value for-format-slot)* sont définies de manière similaire. La création d'un format (défini par *make-object)* se passe selon la séquence suivante :

- création d'une décoration vide ;
- ajout des deux contraintes *:existing-decorations-for-slot-name* et *:valid-value for-format-slot ;*
- ajout de la contrainte *:locked.*

Cet ordre est particulièrement important car il n'est, par exemple, pas possible de rajouter les attributs ou les valeurs si on commence par verrouiller la frame.

Les attributs de MÉTAL.

À la fin du chapitre précédent, nous avons indiqué que le mécanisme de création de contraintes était généralisable à la création d'attributs semblables à ceux rencontrés dans le système METAL.

6.3. Limites et difficultés de la généricité

Le conception de composants génériques pose plusieurs problèmes. D'une façon générale, il est difficile de concilier la généricité, et donc la création potentielle de nouveaux types d'objets, et la nécessité d'avoir un modèle objet cohérent. Par ailleurs,

des problèmes de compréhensibilité surgissent. Des limites à la généricité surgissent en fait assez rapidement quand on est confronté à son implémentation concrète.

6.3.1. Gestion des objets et des protocoles

Les protocoles génériques définissent un espace où une infinité d'objets nouveaux peuvent être définis. Cependant, il est toujours possible d'imaginer de nouveaux objets dont la définition n'est pas possible pour un protocole donné.

La généricité implique la spécialisation des méthodes contenues dans ces protocoles. Cette spécialisation peut se concrétiser à l'aide de stratifications qui offrent une bonne réutilisabilité, et de plus permettent surtout de diminuer la complexité de compréhension des protocoles.

Espace d'objets

Il n'est pas possible de définir a priori tous les types d'objets ou de contraintes. Supposons que l'on veuille implémenter une contrainte empêchant une frame d'être héritée (on trouve l'idée de cette contrainte dans [Apple Computer Inc. 1992]). Cette contrainte doit être prise en compte lors d'une tentative de définition. Par exemple, on aimerait avoir le comportement suivant :

Soit f une frame "scellée" :

```
<:frame f
     (a 1)
     (b 2)
     (sealed :vrai)>
```

```
(define! sub-f :frame
     (c 3)
     (:is-a f)

error: f is a sealed frame and cannot be subclassed.
> nil
```

Que se passe-t-il lorsqu'une frame est créée avec l'attribut *:is-a* ? Il est nécessaire d'effectuer une vérification massive de l'ensemble des contraintes afin de déterminer si la définition est valide.

Une contrainte comme *sealed* s'applique à la création de la frame. Or, nous n'avons pas défini de démons s'activant à ce moment là. La raison en est qu'un démon appartient à une frame, il n'est donc pas possible d'appeler le démon avant sa création.

Il est cependant possible, en pratique, de contourner ce mécanisme sans remettre en cause l'ensemble du système. Par exemple, si la contrainte *:sealed* est trouvée pour un des parents d'une frame, après sa création, on peut détruire cette frame et produire un message d'erreur. Il faut tout de même s'assurer qu'une frame disposant déjà de parents de plusieurs sous-frames ne puisse pas devenir scellée. On n'autorisera pas la modification à l'exécution d'un tel attribut.

Cette nouvelle contrainte illustre les limites de ce modèle. Il est en effet toujours possible de reconcevoir tout ou partie du système, dès que l'on est confronté à ce genre de problème, mais on risque d'augmenter la complexité de l'ensemble pour un gain somme toute faible.

Plusieurs types de protocoles

Au vu de l'expérience menée avec DÉCOR, nous dégageons trois grandes classes de protocoles :

- Protocole utilisateur ;
- Sous-protocole ;
- Sur-protocole.

Un *protocole utilisateur* (ou développeur) permet d'utiliser les fonctionnalités de l'outil. Pour DÉCOR, il s'agit des fonctions de définition et de manipulation des décorations et des types. Ce niveau de protocole peut être utilisé par un protocole client pour construire un système de plus haut niveau "client" des protocoles utilisateurs, considérées comme *sous-protocoles*.

Par exemple, la méthode *make-object* est un sous-protocole de *define!.*

Notons qu'il ne faut pas céder à la tentation de définir une régression très longue de sous-protocoles, qu'il est ensuite difficile de justifier face à la dégradation des performances et face à la trop grande complexité des mécanismes mis en jeu.

Un *sur-protocole* consiste en des fonctions qui permettent de définir de nouvelles catégories d'objets. Ces fonctions ne sont pas spécialisées, mais utilisées telles quelles. En général, un sur-protocole s'accompagne de sous-protocoles. On parle de stratification vers le haut.

Prenons un exemple. Pour définir de nouveaux types d'objets, il est nécessaire d'écrire une spécialisation de *make-object* ainsi que de *make-object-handle-keyword.* On peut trouver génant d'avoir à faire à une spécialisation éparpillée sur plusieurs fonctions alors que l'écriture d'une seule fonction devrait suffire.

Il est possible de faire une dernière spécialisation sur *make-object* (et donc indirectement sur *define!)* de façon à définir de nouveaux objets uniquement à partir de *define!.* Par exemple, pour préciser comment définir la nouvelle catégorie d'objets que sont les contraintes, on écrirait :

```
(define! :constraint :object
   (:keywords :arguments :object-category :deamons :check-constraint-method)
   (:arguments (… ;; méthode qui gère les arguments)
   (:deamon    (… ;; code qui gère les démons)
…
)
```

Nous venons d'obtenir un "point-fixe" pour ce sur-protocole, car tous les objets peuvent être exprimés avec *define!,* y compris les définitions de *define!* nécessaires à de nouveaux objets.

Même si finalement, on peut limiter la régression infinie des sur-protocoles, nous avons un problème d'extensibilité, car la liste des mots-clés acceptables pour *define!* est fixe. Il est possible de créer des fonctions qui ajoutent dynamiquement de nouvaux mots-clés ainsi que la définition des fonctions qui les gèrent. Cependant, il faut garder à l'esprit qu'un outillage important est nécessaire pour que l'ensemble reste utilisable.

6.3.2. Gestion de la complexité

La stratification vers le bas simplifie le travail de spécialisation en le morcelant. La réutilisabilité et dans une certaine mesure la qualité du code sont améliorées. Cependant, la stratification vers le bas a tendance à contraindre la forme que peut prendre la réalisation d'une spécialisation et aussi à être trop fortement morcelée. Une stratification vers le haut permet de regrouper la spécialisation en une seule fonction. Par contre, cette solution a tendance à figer les définitions et fait perdre de la généricité, sauf s'il est possible de trouver un point fixe. Les modalités d'utilisation de la fonction qui représente ce point fixe doivent être acceptables pour le développeur.

Analyse du modèle

L'approche naïve consiste à implanter des objets dès qu'on les "voit". Cependant une telle approche peut poser des problèmes d'efficacité. Il est parfois nécessaire de factoriser des instances ou de s'écarter du modèle original[56]. Une modélisation moins naïve est toujours possible, mais elle devient plus difficilement compréhensible.

Nous n'avons pas abordé les problèmes d'efficacité. On peut déjà constater que la forme des protocoles peut être influencée par ces considérations. Il en résulte, en général, une plus grande complexité statique mais une diminution de la complexité algorithmique (ou dynamique).

Compréhensibilité

Le plus grand écueil lié à la définition de protocoles stratifiés est leur complexité croissante. Pour rester compréhensibles, ces protocoles doivent faire l'objet d'une documentation très précise.

Par contre, le grand avantage est que la stratification morcelle les problèmes, et qu'il n'est plus nécessaire de connaître l'ensemble du système pour pouvoir l'étendre.

La généricité se paye par une plus grande complexité. Avant de rendre un système plus générique, il est donc nécessaire de se demander si le gain résultant justifie l'accroissement de complexité.

---

56 Une expérience qui éclaire relativement bien ces difficultés au niveau de l'implémentation est rapportée dans [Lafourcade 1993a] et [Lafourcade & Sérasset 1993b]. La construction d'un grapheur générique est certes éloignée de celle des LSPL, mais des problèmes d'efficacité liées à une modélisation trop naïve peuvent apparaître dans les deux cas.

Conclusion de la seconde partie

Nous avons défini un langage de décoration selon une double perspective. D'une part, il s'agissait de fournir un langage noyau dont les primitives pouvaient trouver un usage dans le TALN mais aussi pour d'autres applications. D'autre part, ce langage a fourni un terrain d'expérimentation pour la recherche de la généricité.

Construire un langage à partir d'un langage noyau se fait à la fois en sélectionnant le sous-ensemble des objets et des primitives strictement nécessaires, et en construisant de nouveaux types d'objets. Pour pouvoir créer et gérer de nouveaux objets sans modification du code existant, nous utilisons des protocoles stratifiés.

La stratification permet de morceler les protocoles et de réduire ainsi la complexité, mais aboutit à des spécifications de plus en plus contraignantes où il reste très peu de latitude pour des implémentations différentes.

Enfin, nous avons constaté que, pour qu'un outil soit générique, il faut le munir de mécanismes qui permettent d'étendre ses fonctionalités. La généricité est donc directement dépendante de l'extensibilité.

Troisième partie

Extensibilité et langages spécialisés

Introduction de la troisième partie

Quelles formes peut prendre l'extensibilité dans les LSPL ? Les LSPL auxquels nous nous intéressons ici sont les "moteurs", tels qu'ils ont été définis par le modèle LEAF. L'analyse et le développement de tels outils n'est pas facile et il est semble nécessaire de leur assurer une certaine généricité. Mais, nous l'avons vu, la généricité passe entre autres par l'extensibilité. Nous devons donc développer des techniques permettant de rajouter facilement de "nouveaux comportements" aux moteurs.

Dans [Bouchard & Emirkanian 1990], au moins deux questions importantes sont posées. Comment construire efficacement des prototypes pour la linguistique informatique ? Comment adapter des outils spécialisés ? Le paradigme de la programmation exploratoire semble être un des éléments de réponse à la première question. Disposer d'un outil extensible permet une programmation exploratoire, car des familles de fonctionnalités (à l'image des familles de langages) peuvent être définies sans avoir à supporter le développement lourd de l'ensemble d'un outil. S'ils sont extensibles, les outils spécialisés peuvent devenir génériques ce qui ne veut pas dire qu'ils deviennent généraux. Ils restent spécialisés mais peuvent aussi être spécialisés pour d'autres activités. La généricité débouche également sur des familles d'outils.

L'extensibilité se base sur la programmation par objets, et bien entendu sur des protocoles. Dans [Habert 1991] et [Habert 1992], une expérience a été menée avec la réalisation d'un "analyseur à objets". La combinaison de classes permet dans une certaine mesure la combinaison des comportements. Cette approche, qui consiste à mélanger des objets aux compétences différentes afin d'obtenir un objet hybride qui dispose de la somme des compétences, semble particulièrement intéressante. L'idée peut même être poussée plus loin avec la perspective d'analyseurs réflexifs [Habert & Fleury 1993b].

Notre exploration de ces techniques a commencé avec le langage spécialisé ATEF. Nous avons analysé quelques aspects pouvant être généralisés dans le modèle original. Les protocoles sont pleinement utilisés afin de définir des mécanismes

d'extension du moteur. Les protocoles permettent soit d'ajouter de nouveaux modes de fonctionnement, soit de fournir un langage commun à certains composants qui deviennent ainsi particulièrement modulaires. Nous continuons notre expérimentation sur l'extensibilité avec ROBRA, un LSPL qui permet de décrire des systèmes transformationnels travaillant sur des arbres décorés. La définition et l'utilisation de classes fusionnantes semble un bon moyen d'offrir une extensibilité "à la carte". Nous essayons ensuite de généraliser les techniques utilisées pour ces deux LSPL.

Chapitre 7

ATEF – Un langage pour l'analyse morphologique

ATEF est un LSPL basé sur un modèle de transducteur chaîne-arbre. Ce transducteur d'états finis est non-déterministe n-aire. L'analyse du langage original sert de base à une redéfinition extensible. L'idée de l'extensibilité consiste à déporter la plus grande partie du contrôle du moteur dans les données. Cette idée est entre autres appliquée pour l'implémentation de fonctions heuristiques. Ainsi, de nouvelles fonctions de contrôle du non-déterminisme peuvent être rajoutées sans modifier le code existant.

Nous présentons quelques-uns des protocoles que nous avons créés pour la nouvelle version d'ATEF, ainsi que les classes qui les implémentent. Nous constatons qu'une définition générique et extensible offre des perspectives de réutilisabilité intéressantes, mais qu'elle peut contraindre et compliquer sensiblement l'implémentation.

7.1. Modèle original

ATEF (Analyse de Textes en États Finis) est un langage spécialisé qui permet d'écrire des analyseurs morphologiques de langue naturelle [Chauché 1974, Chauché 1975]. Conçu et implémenté entre 1971 et 1973 par J. Chauché, P. Guillaume et M. Quézel-Ambrunaz, ATEF a été utilisé depuis pour l'écriture d'analyseurs du français, du russe, du portugais, de l'anglais, du japonais et de l'allemand. Pour une introduction générale à ATEF, voir [Lafourcade 1991a].

7.1.0. Présentation générale

ATEF est fondé sur le modèle de transduction d'états finis non-déterministe n-aire. Les données externes linguistiques comprennent :

- des déclarations de variables et de formats. Ces types de structures ont été présentés dans le chapitre 4 ;
- des dictionnaires ;
- des grammaires.

Le système accepte en entrée un texte (une chaîne de caractères pour simplifier). Chaque forme[57] du texte est analysée successivement en examinant toutes les analyses possibles. Une étape de l'analyse consiste à découper un segment de ce qui reste de la forme courante et à appliquer une des règles de la grammaire. Les règles sont référencées par un format "morphologique" en correspondance avec les segments contenus dans les dictionnaires. Les variables et les formats sont divisés en deux catégories, "morphologiques" et "syntaxiques".

Les conditions exprimées dans les règles peuvent porter sur les résultats d'analyse des quatre formes précédentes.

7.1.1. Éléments linguistiques

Les éléments linguistiques sont :

- un système de décorations qui permet de définir des variables et des formats ;
- des dictionnaires ;
- des grammaires composées de règles.

Système de décorations

Les structures de données linguistiques ont déjà été présentées dans la seconde partie. Nous les représentons brièvement ici, réduites au LSPL ATEF. Les dictionnaires et les règles se basent sur le formalisme d'ARIANE (voir §4.1.2) pour coder leurs informations.

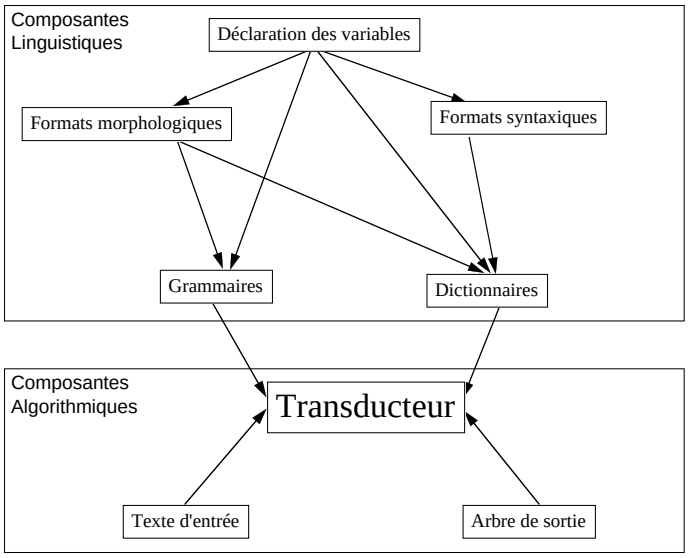


Figure 7.1 : Organisation des éléments linguistiques et algorithmiques d'ATEF.

57 Les formes sont les sous-chaînes de caractères comprises entre des caractères espace.

Les *variables* correspondent aux catégories linguistiques formalisées. Il existe deux types de variables : les variables morphologiques et les variables syntaxiques. En fait, cette distinction est relativement artificielle, les linguistes pouvant utiliser les deux pour coder n'importe quel type d'information.

Exemples de variables (ces exemples correspondent à la morphologie de l'allemand) :

```
SUBN == (PR, PF, IP).
```

PR se rapporte aux noms propres, PF aux noms parfaits pouvant se passer d'article et IP aux noms imparfaits nécessitant un article.

```
SUBCOP == (COOR, CJS, PRP).
```

SUBCOP est une sous classification des lexèmes d'hypotaxe ou de parataxe. COOR signifie conjonction de coordination, CJS conjonction de subordination et PRP préposition.

Une variable est définie par un ensemble de valeurs élémentaires et un caractère d'exclusivité (-EXC-) ou non-exclusivité (-NEX-). Une *variable exclusive* ne peut prendre qu'une seule valeur à la fois parmi celles possibles. Une *variable non-exclusive* peut avoir comme valeur un sous-ensemble des valeurs possibles.

Les variables non-exclusives peuvent être manipulées à l'aide d'opérations ensemblistes (union, intersection, égalité, etc.). L'utilisation de ce type de variables permet de simplifier l'écriture des conditions et de diminuer sensiblement le nombre de règles de la grammaire (celle-ci gagne alors en clarté).

Il existe deux noms de variables réservés : UL et DICT.

- UL (Unité Lexicale) indique la référence lexicale d'une forme. Par exemple, les formes conjuguées du verbe "être" renvoient toutes à la même référence lexicale (qui est le lemme "être"). UL est une variable exclusive.
- DICT permet de déclarer les dictionnaires utilisés. Il n'est pas possible d'avoir plus de 6 valeurs (6 dictionnaires au maximum). Cette variable doit être déclarée comme morphologique non-exclusive, mais est traité spécialement[58].

La déclaration des variables est répartie sur deux fichiers, l'un contenant les variables morphologiques et l'autre les variables syntaxiques. À l'intérieur de ces fichiers, les variables exclusives et non-exclusives sont regroupées et précédées respectivement par les mots-clés "-EXC-" ou "-NEX-".

Le linguiste peut créer des variables dites "stratégiques". Elles ne contiennent pas d'informations linguistiques, mais permettent de contrôler ou de guider l'application des règles.

Un format correspond à une classe de mots définie par le linguiste. Un format morphologique appelle aussi une ou plusieurs règles de la grammaire. À chacune des classes qu'il a définies, le linguiste fait correspondre une combinaison de valeurs. Cet ensemble de valeurs constitue un format. Chaque format a un nom.

---

58 L'affectation VARNM(C) := VARNM(A) ne modifie pas la variable DICT. DICT est considérée comme une "variable de contrôle".

Exemple de format :

```
FSCOOR == KMS (COP), SUBCOP(COOR).
```

KMS doit valoir CP et SUBCOP doit valoir COOR.

Comme pour les variables, il existe deux types de formats : les *formats morphologiques* (FTM) et les *formats syntaxiques* (FTS). L'information peut donc être (grosso modo) d'ordre morphologique ou syntaxique. Ce système impose certaines contraintes, par exemple l'impossibilité d'utiliser une variable syntaxique (VARS) dans un format morphologique (FTM).

Un format peut être vide. Dans ce cas (et si c'est un FTM), son rôle consiste seulement à activer une ou plusieurs règles.

Nous pouvons prendre comme exemple des extraits d'une application ATEF pour l'analyse des noms chimiques (que l'on nommera dans la suite atef-chem).

Les variables syntaxiques exclusives sont :

```
-EXC-
LAISON == (ALCANE, ALCENE, ALCYNE, PHENYL).
NIV == (PRINCIP, SECOND, TERT).
STRUC == (CYCL, LINE).
VALENCE == (1, 2, 3).
MULTIP == (1, 2, 3, 4, 5, 6, 7, 8, 9).
```

La variable morphologique exclusive est :

```
SEG == (PREF, BASE, SUFF).
```

Les variables syntaxiques non-exclusives sont :

```
-NEX-
FONCTIO == (ALIPH, ACIDE, ETHER, ESTER, AMIDE, AMINE, ALDEHYDE,
            CETONE, AROMAT).
SUBST == (CHLORO, NITRO, SULFO, IODO, BROMO).
```

La variable morphologique non-exclusive est la variable obligatoire DICT :

```
DICT == (1, 2, 3).
```

Les formats syntaxiques (extraits) sont :

```
-FTS-
AL     01==.**FONCTIO-E-ALDEHYDE.
ALIPH  01==.**STRUCT-E-LINE.
AMIDE  01==.**FONCTIO-E-AMIDE.
…
```

Les formats morphologiques (extraits) sont :

```
-FTM-
BASE      01==.**SEG-E-BASE.
CYCLM     01==.**SEG-E-PREF.
…
```

Dictionnaires

Les dictionnaires associent des informations grammaticales et des références lexicales aux morphes d'une langue. Un morphe est une chaîne élémentaire intervenant lors du découpage des occurrences du texte par l'automate.

Les dictionnaires "de bases" contiennent une UL par article, des formes complètes fléchies, des mots invariables, des tournures figées, etc. Typiquement, leurs morphes sont des "bases" (par exemple, MANG pour manger). Les dictionnaires de désinences, sans UL, contiennent les affixes (désinences et préfixes lexicalement vides).

À chaque article de dictionnaire sont associés les deux formats FTS et FTM qui définissent les informations linguistiques liées au morphe.

Exemple d'entrée de dictionnaires :

```
UND == FMINV (FSCOOR, UND).
```

Cet exemple est encore tiré de la morphologie de l'allemand de [Guilbaud 1980]. Le premier UND est l'article du dictionnaire. FMINV est le nom de format FTM. FSCOOR est le nom de format FTS. Le deuxième UND est la référence lexicale (UL).

Il ne peut y avoir découpage des mots qu'après consultation des dictionnaires. L'accès aux dictionnaires est contrôlé dans la grammaire à l'aide de la règle obligatoire RDICT (les dictionnaires disponibles au début du traitement)[59]. Il existe aussi une règle automatique d'ouverture des dictionnaires, elle correspond aussi à l'affectation automatique de la valeur d'UL.

Voici quelques articles des dictionnaires d'atef-chem ([Chauché 1974]):

```
** Dictionnaire de bases
.                == SEPA       (VIDES, POINT).
BENZENE          == BENZT      (BENZ, BENZENE).
BUT              == BASE       (ALIPH, BUTANE).
EICOS            == BASE       (ALIPH, EICOSNE).
…
** Premier dictionnaire d'affixes
-                == ELIMA      (VIDES).
AL               == SFCTS      (AL).
AMIDE            == SFCTS      (AMIDE).
…
** Deuxième dictionnaire d'affixes
BROMO            == PRESUB     (BROMO).
CHLORO           == PRESUB     (CHLORO).
CYCLO            == CYCLM      (CYCLO).
Di               == PREFN      (ISO).
…
```

Règles

Une grammaire est une suite de règles. Les règles RDICT et MOTINC sont deux règles spéciales obligatoires. La structure d'une règle normale est la suivante :

```
nom : format [-format]* ==  action[/condition
                            [/modification d'entrée
                            [/condition S[/sous-règles]]]].
```

Une règle se compose d'une partie gauche et d'une partie droite (de part et d'autre de "=="). En partie gauche, on trouve le nom de la règle et une liste (non vide) de noms

59 Il peut aussi y avoir un contrôle dans les autres règles, mais ce n'est pas obligatoire.

de formats FTM[60]. Les règles sont appliquées sur les morphes dont la classe correspond à l'un des formats de la liste de la partie gauche. En partie droite, on trouve des *instructions* et des *conditions*.

Les instructions peuvent être des affectations de valeurs aux variables ou des transformations de chaînes (TChaînes). Les conditions doivent être vérifiées pour autoriser l'application de la règle. Des sous-règles constituent également des conditions (dites complémentaires). Une des sous-règles, au moins, doit s'appliquer pour que la condition soit vérifiée, et ce récursivement.

Exemple de règle :

```
TC01A : V01A ==  VARS(C) := VARS(A), VAREM(C) := VAREM(A),
                 VARNM(C) := VARNM(C) -U- VARNM(A) /
                 / TCHAINE(0,'','01').
```

Cet exemple est encore tiré de la morphologie de l'allemand. Le nom de la règle est TC01A. Elle est appliquée si le segment courant correspond au format V01A. Dans ce cas, les valeurs des variables VARS (variables syntaxiques) du segment trouvé dans le dictionnaire (A) sont affectées aux variables correspondantes du segment courant (C). Il en est de même pour les VAREM (variables exclusives morphologiques). Pour les variables VARNM (variables non-exclusives morphologiques), on demande l'union entre les valeurs courantes et les valeurs issues du dictionnaire. D'autre part, on applique une TChaine (on modifie la chaîne courante en insérant à la position courante[61] la chaîne "01").

Voici quelques règles d'atef-chem :

```
** Règles concernant les suffixes
RSUF1:    SFCT   ==   VAR(C):=VAR(A), DICT(C) := 2/
                      SEG(C)-E-SEG0.
RSUF2:    SFCTS  ==   VAR(C):=VAR(A)/SEG(C)-E-SEG0/
                      TCHAINE(0,'','E').
RSUF3:    SFCT   ==   LIAISON(C):=LIAISON(A),NIV(C):=NIV(A),
                      DICT(C):=2/
                      FONCTIO(C)-DANS-(-N(ALIPH))-ET-SEG(C)-E-SUFF.
RSUF4:    STCTD-SFCTE
                  ==   VAR(C):=VAR(A), -ARRET-, DICT(C):=2/
                      VAR(PS1)-NE-VAR0.
** Règles concernant les bases
RBAS1:    BENZT  ==   VAR(C):=VAR(A)/VAR(C)-E-VAR0///
                      RNCST, RCSTV3, RCSTV1, RCSTV2.
…
```

7.1.2. Fonctionnement de l'automate

La grammaire d'une application écrite en ATEF spécifie un automate non-déterministe d'états finis, dont la fonction est de segmenter les formes du texte. Une forme est considérée comme une chaîne de caractères entre deux blancs. Le résultat de la segmentation est un ensemble de morphes compatibles. Cette compatibilité est assurée à l'aide du traitement linguistique qui associe des informations linguistiques à ces morphes pour aboutir aux valeurs représentatives du découpage.

---

60 Ces formats FTM constituent le vocabulaire d'entrée de l'automate.

61 C'est l'origine 0, c'est-à dire la fin du morphe courant.

Le parcours de l'arbre des solutions possibles se fait en profondeur d'abord (figure 7.2). Toutes les solutions possibles sont produites, à moins que le linguiste ne fasse intervenir, au niveau des règles, des fonctions heuristiques qui permettent de contrôler le non-déterminisme (voir §7.1.3.).

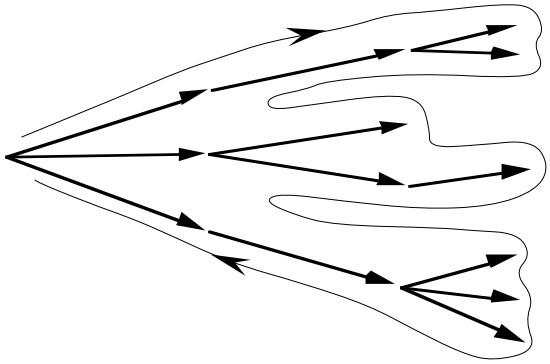


Figure 7.2 : Parcours de l'arborescence des choix sans fonction de contrôle du non-déterminisme.

L'état courant de l'automate est déterminé à un instant donné par :

- la position d'un pointeur sur la forme à analyser ;
- l'ensemble des informations linguistiques du moment.

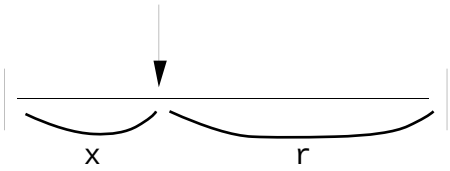


Figure 7.3 : Le pointeur dans le chaîne de caractères courante.

x est la sous-chaîne déjà analysée et r est la sous-chaîne restant à analyser. On a ici une analyse de la gauche vers la droite.

Les informations linguistiques sont contenues dans la décoration "courante" C, qui contient toutes les variables (VARM, VARS) ainsi que l'UL, et la variable spéciale DICT. Au départ C est vide. Nous appellerons souvent, par abus de langage, *état du système* la valeur de C[62].

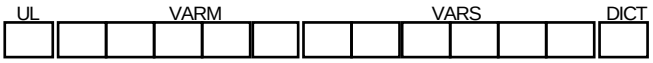


Figure 7.4 : État courant des variables linguistiques.

l'état courant contient l'ensemble des variables déclarées. Il indique quelle valeur a l'état courant pour chaque variable.

Un morphe est *candidat* dans l'état courant s'il appartient à un dictionnaire ouvert et s'il préfixe ou suffixe ce qui reste à analyser (la sous-chaîne r de la figure 7.4), selon que l'analyse se fait de la gauche vers la droite ou de la droite vers la gauche.

À chaque morphe *m* candidat sont associées ses informations linguistiques[63], ainsi qu'une ou plusieurs règles de la grammaire. Une règle (potentiellement applicable) est

62 On parle également de registre C.

63 C'est la décoration obtenue en faisant l'union du FTS, du FTM et de l'UL associé à m dans l'article de dictionnaire. On la note A.

appelée si et seulement si le format FTM du morphe fait partie de la liste de formats FTM de la règle.

Le corps de la règle appelée est applicable si ses conditions sont vérifiées. Ces conditions portent sur C et/ou A(m). Si les conditions sont vérifiées, la partie action de la règle peut être exécutée. La règle est entièrement applicable si le corps et les sous-règles sont applicables.

Si une des règles associées au morphe candidat est applicable, le morphe est dit *acceptable*. Dans ce cas, C est modifié selon la partie action de la règle et le repère est déplacé (figure 7.5).

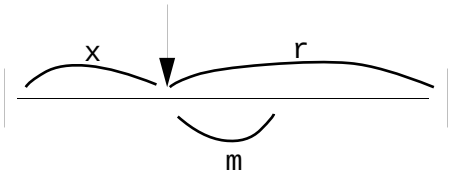


Figure 7.5 : Déplacement du repère.

Si le repère se trouve en bout de chaîne et si UL(C) est différent de UL0 (valeur indiquant qu'on a pas trouvé d'unité lexicale), il y a succès du découpage. C constitue alors un résultat de l'analyse de la forme.

L'automate effectue alors un retour en arrière à l'état immédiatement précédent. Il essaye alors les autres solutions possibles de découpage. L'alternative est la suivante :

- soit la règle suivante est applicable pour le même morphe ;
- soit un autre morphe candidat est essayé.

Si le repère n'est pas en bout de chaîne et si aucun morphe n'est acceptable, il y a *échec* de l'analyse en cours. Il y a alors retour en arrière, et une autre solution est essayée.

Pour un "pas" de segmentation effectué, trois opérations élémentaires sont réalisées :

- le choix d'un dictionnaire ;
- le choix d'un segment ;
- l'application d'une règle.

Par exemple, la segmentation de «pentaméthylpropobutylidèneoctanéol»par atef-chem est représentée par la figure 7.6. Cette grammaire s'applique de la droite vers la gauche, ce qui explique que la segmentation commence par la fin du mot. Seul le premier chemin a abouti à un découpage complet. L'échec des autres chemins peut être dû soit à l'inexistence d'une sous-chaîne dans les dictionnaires, soit à l'échec d'une règle.

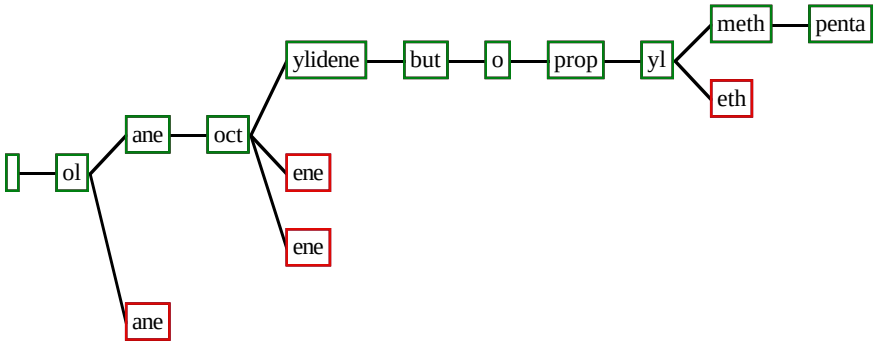


Figure 7.6 : Découpage du mot «pentaméthylpropobutylidèneoctanéol».

7.1.3. Contrôle du non-déterminisme

Les fonctions heuristiques permettent de contrôler le non-déterminisme de l'automate, c'est-à-dire de ne calculer que les solutions les plus prometteuses. Elles ne permettent pas de modifier la stratégie de parcours de l'arbre des calculs possibles (toujours en profondeur d'abord), elles ne font qu'élaguer cet arbre.

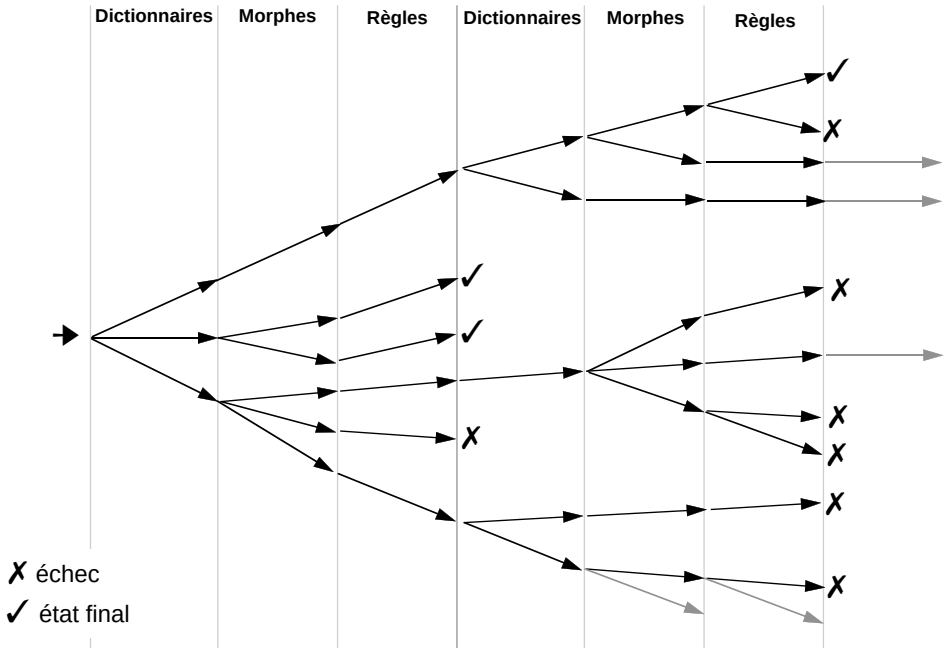


Figure 7.7 : Arborescence des choix dans ATEF.

Les fonctions FINAL, STOP, ARRÊT, ARD, ARF permettent d'éliminer les choix faits ou à faire. Ces fonctions sont activées au retour d'une chemin qui a mené à une solution (figure 7.7) .

FINAL

Cette fonction permet de ne garder qu'un seul chemin dans l'arborescence des choix. Ce chemin est celui qui part de la racine de l'arborescence, passe par la branche contenant FINAL, et continue sur le premier tronçon qui mène à une solution. Si la règle contenant FINAL ne mène à aucune solution, cette règle échoue, et FINAL n'a aucun effet.

Le schéma 7.8 montre une segmentation en deux pas. Dès le premier pas réussi, tout ce qui a précédé est rendu caduc par FINAL et la première solution du deuxième pas sera forcément la bonne. L'effet de FINAL ne s'applique que sur le pas courant et le pas suivant.

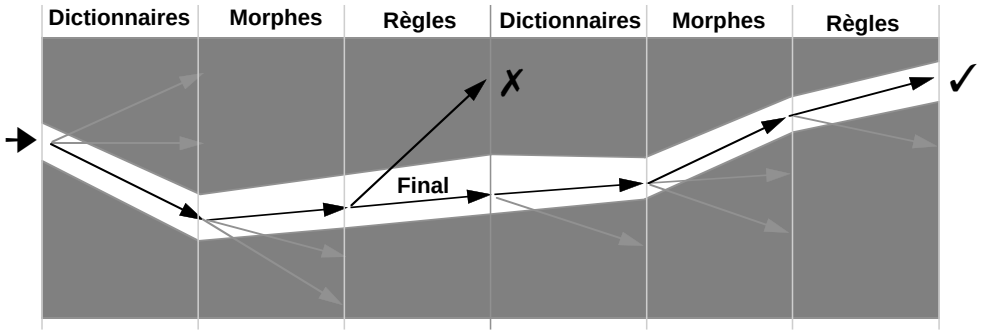


Figure 7.8 : Effet de la fonction de contrôle FINAL sur l'arborescence des choix.

Cette fonction est utilisée pour les mots non-ambigus qui pourraient donner lieu à des découpages parasites ou à des essais de découpage voués à l'échec.

STOP

La fonction STOP permet d'arrêter le traitement si la règle en question conduit à une solution. Dans ce cas, toutes les solutions précédemment trouvées sont conservées.

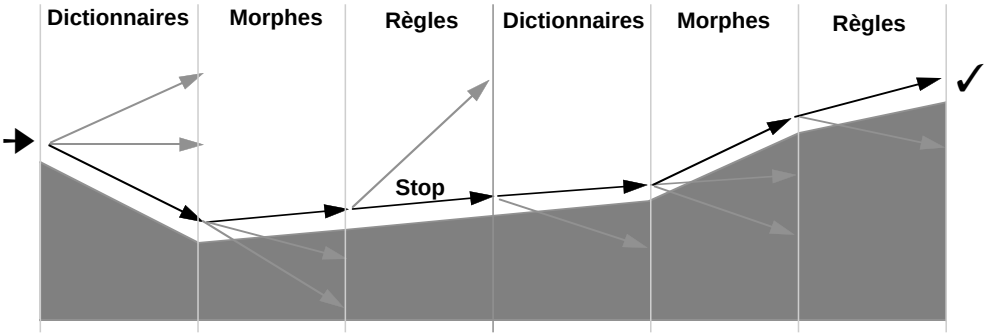


Figure 7.9 : Effet de la fonction de contrôle STOP sur l'arborescence des choix.

ARRÊT

Si la règle courante est applicable, la fonction ARRÊT a l'effet suivant :

- éliminer les règles ultérieures qui pourraient s'appliquer sur le segment courant ;
- éliminer les morphes provenant du même dictionnaire et strictement plus courts, ainsi que les morphes provenant de dictionnaires moins prioritaires ;

- éliminer toutes les segmentations et toutes les règles pour les pas de rang inférieur.

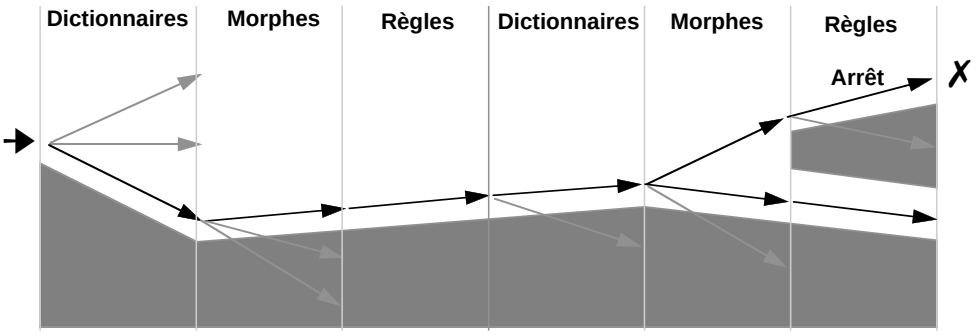


Figure 7.10 : Effet de la fonction de contrôle ARRÊT sur l'arborescence des choix.

ARD

L'arrêt dictionnaire (ARD) permet de :

- empêcher l'essai d'application des autres règles candidates au traitement du segment courant ;
- éliminer les morphes en attente qui proviennent du même dictionnaire et qui sont de longueur inférieure.

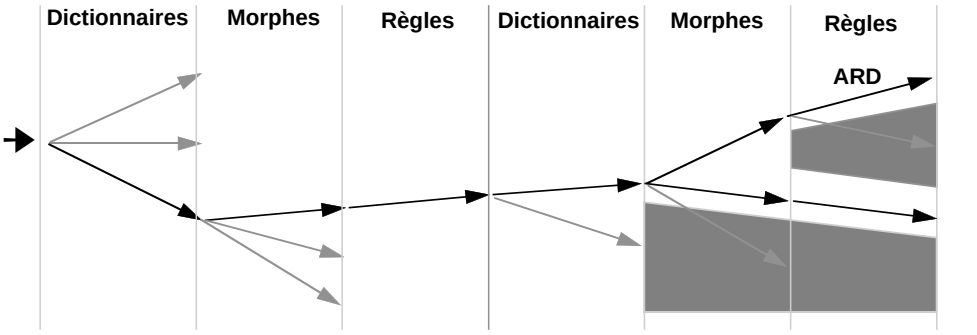


Figure 7.11 : Effet de la fonction de contrôle ARD sur l'arborescence des choix.

ARF

La fonction d'arrêt fort (ARF) permet de :

- éliminer les règles ultérieures qui pourraient s'appliquer sur le segment courant ;
- éliminer les morphes en attente strictement plus courts ou provenant d'autres dictionnaires, qui correspondent au niveau courant de segmentation.

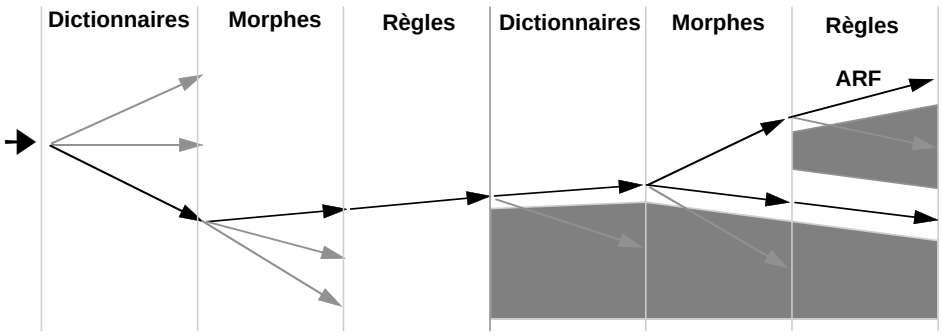


Figure 7.12 : Effet de la fonction de contrôle ARF sur l'arborescence des choix.

7.2. Recherche de l'extensibilité et de la généricité

La réingénierie passe tout d'abord par une phase de reconception. Dans notre perspective, cette phase cherche à déterminer où la généricité et l'extensibilité peuvent être introduites de manière profitable. Nous présentons quelques aspects pour lesquels une telle recherche a été effectuée, dont en particulier :

- la gestion des règles ;
- la gestion du non-déterminisme ;
- la production de multiples formes de sortie.

Dans ce qui suit, nous désignerons le langage classique par ATEF et la nouvelle version par ATEFlisp.

7.2.1. Généralisation et simplification du modèle original

Nous avons cherché dans le modèle original d'ATEF ce qui pouvait faire l'objet d'une généralisation. Le but de cette généralisation est de "reporter" sur les données la partie la plus importante possible du contrôle "câblé" dans le moteur. La conséquence est que les généralisations sont la plupart du temps extensibles et permettent une mise au point aisée et incrémentale. L'exemple le plus clair, pour les LSPL, est de gérer leurs piles de récursivité et de non-déterminisme et surtout de ne pas baser la réalisation sur la pile de récursivité du langage d'implémentation. Pour le non-déterminisme, cette approche avait déjà été adoptée dans l'implémentation originale d'ATEF.

Plusieurs niveaux de modèles

La définition d'un ATEF généralisé disposant de nombreux points d'extensibilité ne doit pas masquer la nécessité d'une compatibilité entre les données de l'ancien modèle et celles du nouveau modèle. Il est hors de question de reconcevoir tous les dictionnaires et toutes les grammaires qui forment autant d'applications ATEF.

Afin de permettre une migration aussi douce que possible, il est nécessaire de définir plusieurs niveaux de modèle. Un modèle très proche du modèle initial est défini de façon à pouvoir assurer une compatibilité satisfaisante pour les linguiciels. Le nouvel

ATEF se base sur un modèle plus général et plus abstrait à partir duquel il est possible de définir des alternatives.

Découplage entre données linguistiques et moteur

Le modèle classique est intimement lié à son système de décorations. Avec la notion de protocole, nous pouvons adopter une approche où un système de décorations pourrait très bien être remplacé par un autre. La contrainte est que ces systèmes de décorations sont des instances du même taxon.

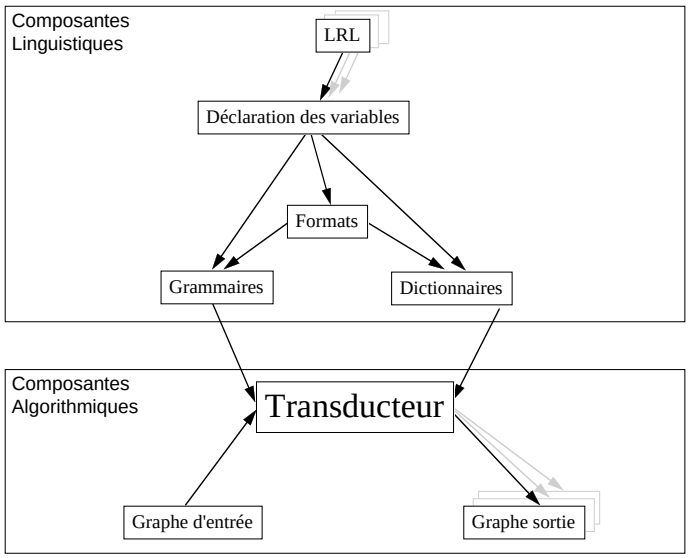


Figure 7.13 : Organisation des éléments linguistiques et algorithmiques de ATEF$_{lisp}$.

La notion de format a été simplifiée par la suppression de la distinction entre formats morphologiques et syntaxiques. Les formats sont réalisés sur la base de DÉCOR.

Moteur

Contrairement au modèle classique, ATEF$_{lisp}$ accepte une liste de chaînes de caractères sous la forme d'un graphe. Les informations de chaque nœud de ce graphe doivent au moins contenir une chaîne de caractères[64]. ATEF$_{lisp}$ peut aussi accepter une chaîne de caractères comme entrée.

Nous avons distingué deux types d'objet : l'outil et le moteur. Le moteur dispose d'un protocole restreint qui permet l'initialisation et l'exécution du traitement. L'outil est assez tolérant sur la nature des objets à analyser qu'il reçoit. Il est capable d'initialiser et de lancer correctement le moteur.

Le moteur exécute plusieurs boucles imbriquées :

- pour chaque forme à analyser ;
- pour chaque segmentation en cours ;
- pour chaque dictionnaire, trouver les morphes candidats ;
- pour chaque morphe candidat, trouver les règles correspondantes ;
- pour chaque règle, l'exécuter.

64 Un modèle objet et un protocole ont été définis afin de manipuler un graphe (de type LEAF) de façon générique.

L'initialisation de la segmentation d'une forme consiste à la dépiler et à produire une forme interne correspondant à un graphe avec un premier nœud vide (il contient le reste de la segmentation qui est à ce moment la forme elle-même). Ensuite, il faut segmenter chaque nœud non terminal (qui n'est ni un échec, ni une segmentation complète) du graphe. Ce processus cherche pour chaque dictionnaire des affixes[65] candidats (ce sont les morphes). On considère pour chaque morphe l'ensemble des règles candidates. Le succès d'une règle offre un solution de segmentation partielle pour la forme. Avant l'application de la règle, un nœud est créé dans le graphe. Ce nœud contient l'ensemble des informations pertinentes (voir plus bas).

Le test de segmentation complète (pour une forme) consiste à avoir un graphe qui ne contient que des nœuds terminaux (échec ou succès)

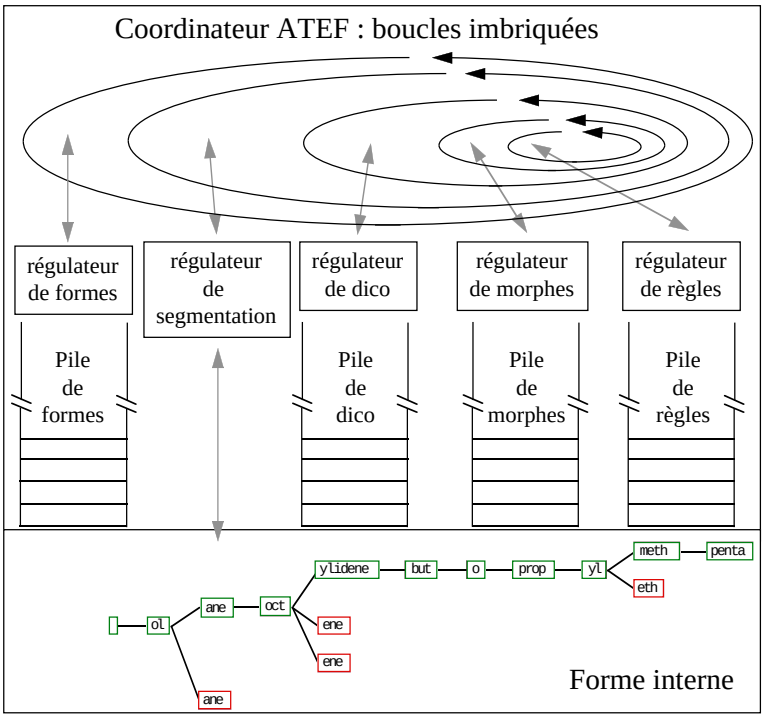


Figure 7.14 : Anatomie du moteur ATEF$_{lisp}$.

Le moteur d'ATEF peut être vu selon l'architecture de tableau blanc. Un coordinateur manipule une structure de données interne (une liste de graphes). Le coordinateur dispose de cinq boucles (de formes, de segmentations, de dictionnaires, de morphes et de règles). Chaque boucle s'exécute de façon autonome tant que les régulateurs n'indiquent pas l'arrêt. Un régulateur fournit, depuis sa pile, chaque élément nécessaire pour la boucle et peut éventuellement modifier le contenu de sa pile.

Nous avons défini quatre piles comme structures pour le contrôle des boucles. Ces piles disposent de différents modes d'accès et de contrôle (pour plus de détail voir les §7.2.2. et §7.3.3.). Chaque pile est associée à un régulateur qui fournit le prochain objet demandé dans la boucle. Ce sont les régulateurs qui imposent la condition d'arrêt de la boucle (qui n'est pas nécessairement la condition de pile vide). Les régulateurs peuvent aussi manipuler le contenu dans la pile si nécessaire. Ces structures de

65 On parle d'affixes car la segmentation peut être effectuée de gauche à droite ou de droite à gauche.

données ont été choisies et systématisées pour toutes les boucles, car elles permettent d'avoir une maîtrise fine sur l'exécution des boucles.

L'idée importante est de rendre le contrôle du moteur le plus souple possible. Le contrôle ne doit pas être câblé une fois pour toute dans le moteur, mais doit être l'objet d'une forte paramétrisation. Ces paramètres eux-mêmes ne doivent pas faire directement l'objet d'une *analyse par cas* qui compromettrait toute définition future de nouveaux modes de contrôle. Il est donc nécessaire de définir des protocoles multiples permettant une telle extensibilité.

Grammaires et règles

Dans le modèle classique, les règles contiennent un format morphologique et un format syntaxique. Nous avons remplacé ces informations par une liste de formats qui constitue une bonne généralisation. Cette liste peut être vide. Si tel est la cas, la règle ne sera jamais accédée. La valeur du registre A est l'union des formats de la liste[66].

Un protocole (nommé *object-data*) permet d'abstraire la structure exacte d'une règle. Ce protocole est défini par deux fonctions génériques :

```
get-object-data (object selector)              [fonction générique]
set-object-data (object selector value)        [fonction générique]
```

Ces deux fonctions génériques peuvent être spécialisées d'une part sur la classe de l'argument *objet* mais aussi sur la valeur de l'argument *selector*.

Pour les règles d'ATEF, nous avons les spécialisations avec les sélecteurs suivants :

- :name — le nom de la règle ;
- :dispatch-mode — le mode de distribution des sous-règles (voir plus bas) ;
- :format-name-list — la liste des formats associés à la règle ;
- :subrule-name-list — la liste des noms des sous-règles.

L'ensemble des objets liés à ATEF$_{lisp}$ répond à ce protocole. Selon les objets et les spécialisations, les sélecteurs peuvent varier[67]. Ce protocole est différent de l'accès standard aux champs d'une classe CLOS dans la mesure où les sélecteurs ne correspondent pas directement à des attributs. Les classes de règles utilisables pour ATEF$_{lisp}$ doivent répondre à ce protocole, mais leur structure peut varier. Par

---

66 L'union est l'opération par défaut et une autre opération peut être spécifiée règle par règle.

67 Un protocole aussi simple nécessite un outillage non négligeable. Par exemple, tous les objets répondent au sélecteur *:selectors* qui permet d'obtenir l'ensemble des sélecteurs auxquels répond l'objet. Si on considère qu'il est intéressant de pouvoir activer ou désactiver des sélecteurs dynamiquement, nous passons à un niveau supérieur de complexité.

Une solution consiste à imposer un format d'écriture pour les méthodes qui implémentent les sélecteurs. Il s'agit par exemple de vérifier systématiquement si le sélecteur est actif ou pas.

```
(defmethod get-object-data ((object object-class) selector)
       (if (active-selector-p object selector)
              …      ;; calculer la valeur
              …      ;; faire autre chose pour le sélecteur désactivé
 ))
```

Cependant, si la forme d'écriture d'une méthode doit être fortement contrainte pour être conforme au protocole, il est souhaitable de définir une "meta-fonction" qui permette de définir automatiquement la fonction de sélection. Une telle approche simplifie le développement mais aussi réduit les risques d'erreur. Par contre, elle complique sensiblement l'ensemble du protocole.

exemple, une règle peut suivre le modèle classique et définir deux attributs pour représenter le format morphologique et le format syntaxique. Une autre classe de règles peut contenir une liste ouverte de formats.

Nous avons également généralisé l'exécution des sous-règles. Dans le modèle classique, une règle réussit si, après évaluation, au moins une des sous-règles réussit. Nous avons défini un mécanisme extensible permettant de spécifier différents modes d'exécution pour les sous-règles. Ce mécanisme de *distribution des sous-règles* est indiqué pour chaque règle. Par exemple :

- mode *:or ;*

Il s'agit du mode par défaut qui correspond au modèle classique.

- mode *:and ;*

Toutes les règles doivent être appliquées avec succès pour que la règle appelante réussisse. L'état du système est restauré entre chaque appel de sous-règle.

- mode *:progn ;*

Il s'agit de la distribution séquentielle. Après l'application d'une sous-règle, l'état du système est transmis à la prochaine sous-règle. Le processus est itéré pour toutes les sous-règles. Cette distribution est celle du LSPL SYGMOR.

Le détail du protocole qui permet de définir de nouvelles distributions de règles est présenté au §7.3.1.

Une grammaire doit fournir les règles à la demande. Tous les objets appartenant à des ensembles (comme les grammaires pour les règles ou les articles pour les dictionnaires) ont une spécialisation sur le sélecteur *:key*. Ce sélecteur retourne la valeur de la clé d'accès pour cet objet. Le protocole DOP (voir §3.3.2.) est systématiquement utilisé pour les grammaires et les dictionnaires.

Le moteur demande à une grammaire de lui fournir des règles sur la base de critères variables. Par exemple, une règle peut être demandée par son nom, ou par une condition exprimant qu'un de ses formats appartient à une liste de formats donnée. De manière générale, toutes les règles qui vérifient un prédicat quelconque sont potentiellement sélectionnables. La solution la plus simple consiste à parcourir toutes les règles de la grammaire et à ne sélectionner que celles qui vérifient le prédicat. Cependant, un tel processus est particulièrement inefficace. L'utilisation du protocole DOP permet de ranger des éléments selon plusieurs références et donc d'avoir un accès direct[68] pour une référence. Il est donc nécessaire de savoir à l'avance avec quelles références les règles vont être accédées[69].

Dictionnaires et Articles

Les éléments des dictionnaires ont une structure figée dans le modèle classique. Il est intéressant de pouvoir :

---

68 Le temps d'accès est à peu près constant (aux collisions près) avec l'utilisation de tables de hachage.

69 Nous devons bien garder à l'esprit que même avec une référence non prévue, l'accès est possible si l'association de cette référence et d'un objet du dictionnaire permet de déterminer une valeur. Un accès non indexé sera simplement moins efficace.

- définir des dictionnaires avec des articles ayant des structures différentes de celle d'ATEF ;
- définir des dictionnaires avec des articles ayant des structures hétérogènes.

Le protocole *object-data* a été également appliqué aux articles de dictionnaires. Nous avons les spécialisations avec les sélecteurs suivants :

- :ref-lex une référence lexicale associée à l'article[70] ;
- :rule-selector le sélecteur de règle.

Ce sélecteur retourne un symbole qui permet au système de savoir sur quel critère les règles doivent être retrouvées à partir des informations fournies par l'article de dictionnaire. Le critère par défaut est le symbole *:format-name-list* et correspond au modèle classique où une règle est sélectionnée si l'un de ses formats d'appel correspond à un des formats de l'article de dictionnaire.

Un autre *mode de sélection de règle* consiste à fournir directement son nom. Dans ce cas, la sélection se fait sur le symbole *:rule-name-list.*

Le mécanisme général pour l'exploitation par le moteur de ce sélecteur est quelque peu délicat. Pour chaque mode de sélection, il faut en effet définir une méthode qui permet de savoir si une règle est candidate. Le dictionnaire doit pouvoir répondre à son propre sélecteur de règle afin de retourner l'information. Par exemple, avec une sélection de règle par nom, l'article de dictionnaire doit pouvoir répondre au sélecteur :rule-name-list.

Pour un article de dictionnaire donné, le moteur lui demande quel est son sélecteur et ensuite demande l'information associée à ce sélecteur. Par exemple, dans le cas de l'accès par nom, le moteur disposera de deux informations :

```
:rule-name-list
'(r1 r2 r3)
```

Les grammaires sélectionnent les règles candidates à partir de ces deux informations. Une méthode est associée pour chaque type de sélecteur.

7.2.2. Forme des productions de sortie

Dans ATEF, le produit d'une analyse morphologique peut prendre des formes diverses. Le moteur produit une forme interne dont la structure est fixe. Cette structure est passée à l'outil qui la transforme en autant de formes de sortie différentes souhaitées. ATEF$_{lisp}$ implémente cela de façon plus générique et extensible.

Formes internes

Le moteur d'ATEF$_{lisp}$ dispose d'une forme interne d'entrée et d'une forme interne de sortie. Ces formes sont toutes les deux des treillis (c'est-à-dire des graphes orientés sans cycle et avec un unique premier nœud et un unique dernier nœud).

L'information portée par un nœud du treillis d'entrée est réduite à une chaîne de caractères.

---

70 On voudrait souvent disposer de plus d'une référence lexicale. Par exemple, trois : lemme + unité lexicale + concept. Le contenu de *:rex-lex* peut être une liste d'association.

La forme interne produite par le moteur d'ATEF$_{lisp}$ est appelée graphe de segmentations. Une segmentation est produite pour chaque forme à analyser et est représentée par un graphe (voir figures 7.6 et 7.14). Un nœud d'une segmentation correspond à une occurrence de règle pour un morphe provenant d'un dictionnaire. Certaines des informations portées par les nœuds du graphe sont les suivantes :

- le morphe m (c'est une chaîne de caractères) ;
- la chaîne r restant à segmenter ;
- le dictionnaire d'où provient le morphe ;
- des informations sur la règle :

  le nom de la règle ;

  un indicateur qui précise l'état de la règle (:null si la règle n'a pas encore été appliquée, t si la règle a été exécutée avec succès, nil si la règle a échoué) ;

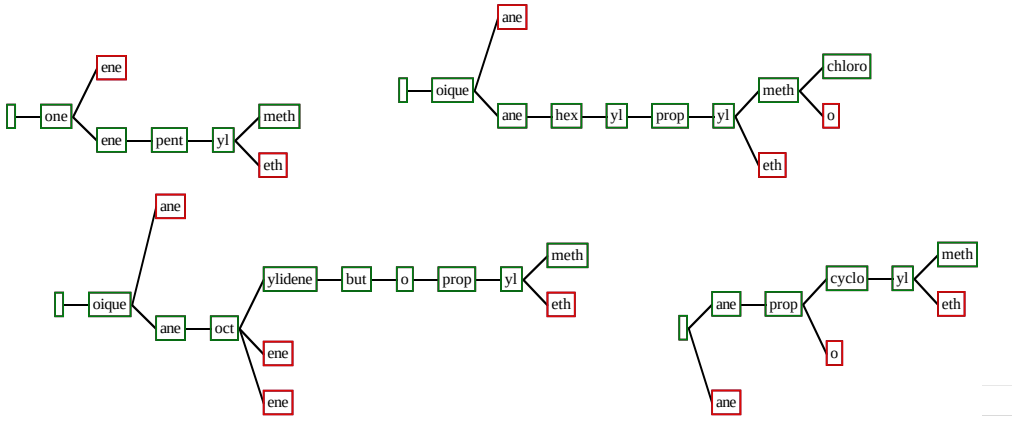
  un arbre d'évaluation des sous-règles.
- l'état du système après application de la règle.

la forme interne répond au protocole *object-data* sur des mots-clés qui correspondent aux informations présentées ci-dessus : :morph, :remaining-string, :dictionary, :rule-name, :rule-state, :engine-state, …

Le graphe dont les nœuds contiennent respectivement les formes suivantes:

```
"methylpentenone"  "chloromethylpropylhexaneoique"
  "methylpropobutylideneoctaneoique"  "methylcyclopropane"
```

produira ainsi un graphe de segmentations dont les nœuds contiendront respectivement les graphes suivants:



Un protocole spécialisant le protocole sur les graphes (défini à partir du modèle LEAF) permet de manipuler la forme interne. Il est par exemple possible de rajouter et de supprimer des nœuds, ou de modifier les informations portées. Pour chaque nœud, l'accès aux informations répond au protocole *object-data* .

Lors de l'analyse d'un texte, l'outil accepte en entrée un graphe de chaînes. Il est possible pour cet outil de fournir au moteur tout ou partie de ce graphe de chaînes. Si plusieurs formes sont données en une seule fois au moteur, les tests sur les précédents pourront être utilisés. Si le moteur ne reçoit qu'une chaîne à la fois, la forme interne sera une liste à un seul élément et toute condition sur les précédents échouera par absence de précédent.

Formes d'entrée et de sortie

L'outil $ATEF_{lisp}$ accepte plusieurs types de formes d'entrée. La méthode *tool-run* demande à l'outil de commencer le traitement.

`tool-run (tool atef-tool) data &keys` [méthode]

Les données à traiter sont contenues dans l'argument data. Certains paramètres de fonctionnement (trace, etc.) peuvent être précisés avec des mots-clés (nous ne les détaillerons pas ici).

Les mots-clés permettent de changer localement l'initialisation du moteur. Par exemple, la direction de segmentation.

```
(tool-run atef-chem "methylpropobutylideneoctanoique"
                :direction :right-left
 )
```

Dans cet exemple, la forme d'entrée est une chaîne de caractères. On peut encore lancer l'outil sur certaines formes de graphes ou sur des listes de chaînes de caractères.

```
(tool-run atef-chem   '("methylpentenone"   "chloromethylpropylhexaneoique"
                        "methylpropobutylideneoctaneoique"
                        "methylcyclopropane")
 )
```

La méthode *tool-run* appelle *tool-produce-input-form* qui convertit la forme d'entrée en une forme d'entrée interne acceptable pour le moteur (c'est-à-dire un treillis de chaînes de caractères). La forme d'entrée privilégiée pour l'outil est évidement un treillis de chaînes de caractères, où aucune conversion n'est nécessaire.

`tool-produce-input-form (tool atef-tool) data` [méthode]

Les données *(data)* sont converties vers la forme interne.

La réalisation de ce protocole peut sembler triviale. En effet, il suffit que la méthode *tool-produce-input-form* réalise une analyse par cas de toutes les formes d'entrée possibles et applique une fonction de transformation. Cependant, une telle approche compromet l'extensibilité. Il ne sera pas possible de définir un nouvelle forme d'entrée acceptable sans modifier le code (par rajout d'un cas) de *tool-produce-input-form.*

Une idée qui assure l'extensibilité est la suivante. Une liste de mots-clés qui désigne toutes les types d'entrées possibles est conservée. On associe à chaque mot-clé un prédicat *pred* et une fonction de transformation *trans*. La production de la forme interne passe d'abord par une identification de la forme d'entrée. La forme d'entrée est identifiée au type d'entrée *te* dès qu'elle vérifie le prédicat $pred_{te}$. Elle est ensuite transformée par $trans_{te}$. Une contrainte forte consiste à s'assurer que tous les prédicats sont bien exclusifs[71].

L'ajout d'une nouvelle forme d'entrée acceptable nécessite donc la mise à jour d'une liste de mots-clés, la définition d'un prédicat et d'une fonction de transformation. On pourrait demander au développeur de spécialiser deux méthodes (*tool-input-form-p* et *tool-input-form-transform*), mais comment mettre à jour la liste qui est une variable

---

[71] Cette contrainte pose un problème dans la mesure où des cas exclusifs avant extension peuvent ne plus être exclusifs après. Cependant, nous ne voyons pas comment résoudre ce problème de façon général.

interne de l'outil[72] ? Il ne s'agit plus ici d'une définition statique mais dynamique, c'est-à-dire qu'une instance d'outil doit être créée.

Une solution à ce problème est de créer effectivement une liste de mots-clés. Cette liste sert de référence au moteur pour toutes les formes d'entrées acceptables. Chaque instance de moteur prend la valeur de cette variable comme valeur par défaut pour les formes d'entrées acceptables.

```
*atef-tool-acceptable-input-forms*                          [variable]
```

Un méta-protocole nous permettra de définir de nouvelles formes d'entrées. Ce protocole est composé :

- d'une fonction générique à appeler pour enregistrer une nouvelle forme d'entrée.

```
tool-register-input-form (tool-class input-form-name)
                                                  [fonction générique]
```

- et de deux méthodes à spécialiser :

```
tool-input-form-p ((tool atef-tool) input-form-name input-form)
                                                          [méthode]

tool-input-form-transform ((tool atef-tool) input-form-name
                           input-form)                    [méthode]
```

Si une nouvelle forme d'entrée est enregistrée mais ne correspond à aucune spécialisation, une erreur se produira à l'exécution.

Nous avons par exemple :

```
(tool-register-input-form :atef 'string)
> t

(defmethod tool-input-form-p ( (tool atef-tool-class)
                               (input-form-name (eql 'string))
                               (input-form string))
  t)

(defmethod tool-input-form-transform ((tool atef-tool-class)
                                      (input-form-name (eql 'string))
                                      (input-form string))

;; produire un graphe à partir de la segmentation de la chaine
 …
 )
```

L'outil ATEF$_{lisp}$ peut produire des formes de sortie de diverses structures. *:output-forms* est un mot-clé acceptable de la méthode *tool-run* qui indique les formes de sortie devant être générées par l'outil à partir de la forme interne produite par le moteur. L'argument demandé est une liste de symboles qui désigne les formes de sortie connues de l'outil. Par exemple :

---

72 L'intérêt de gérer des listes de mots-clés pour la sélection de méthode plutôt que d'utiliser le MOP pour obtenir toutes les spécialisations est double. D'une part, le mécanisme du MOP est lourd, mais surtout il est ainsi possible d'activer ou de désactiver certaines spécialisations tout en gardant leur définition en mémoire.

```
(tool-run atef-chem "methylpropobutylideneoctanoique"
                :direction :right-left
                :output-forms '(output-graph segmentation-list) )
```

Les formes de sorties demandées sont celles qui correspondent aux symboles *output-graph* et *segmentation-list.*

La fonction générique *tool-produce-output-form* indique à l'outil ATEF comment produire une forme de sortie à partir de la forme interne. Cette forme interne ne doit pas être modifiée (elle est en lecture seulement). La signature de cette méthode est la suivante :

```
tool-produce-output-form ((tool atef-tool)
               ouput-form-name internal-form)    [fonction générique]
```

Ce handler doit être spécialisé sur l'argument *output-form-name*. Nous avons produit des formes de résultats telles qu'elles ont été définies dans le modèle classique. Pour ces formes, aucun décodage n'est nécessaire puisque l'on part toujours de la forme interne. Nous avons défini, par exemple, une forme compatible avec les systèmes-Q (nous ne détaillerons pas l'implémentation des méthodes qui la réalisent) :

```
tool-produce-output-form ((tool atef-tool)
          (ouput-form-name (eql :q-graph)) internal-form)   [méthode]
```

Cette forme est un graphe de chaînes qui rend exactement compte des chemins de segmentation.

Nous avons également spécialisé ce handler pour des formes de sorties arborescentes avec et sans homophrases. Cette dernière est la forme de sortie par défaut dans le modèle classique.

Les spécialisations de l'argument *ouput-form-name* qui existent actuellement sont les suivantes :

:internal	retourne la forme interne ;
:q-graph	graphe-q ;
:tree	forme arborescente sans homophrase ;
:tree-with-homophrase	forme arborescente avec homophrase.

L'objectif de ce protocole est de pouvoir ajouter facilement à de nouvelles formes de sorties. Non seulement ce protocole nous fournit de multiples formes externes, mais il donne également un moyen simple de définir plusieurs niveaux de trace.

Calcul des informations strictement nécessaires

Le moteur calcule une forme interne très riche en informations. Certaines de ces informations sont nécessaires pour l'exécution du moteur, d'autres ne sont calculées que pour la production d'une forme externe. Si une forme externe très simple est demandée, par exemple juste la segmentation avec l'état du système à chaque étape, il est inutilement coûteux de garder des informations sur les règles ou les dictionnaires. C'est pourquoi il est intéressant d'indiquer au moteur les informations qu'il est utile de garder dans la forme interne.

La méthode *tool-produce-output-from* a accès à la forme interne du moteur. Cette forme étant connue, il n'est pas inacceptable de demander au développeur d'une nouvelle forme de sortie d'indiquer quels arguments du protocole *object-data* appliqué à la forme interne vont être utilisés.

Notre protocole de définition de formes externes hérite d'une nouvelle méthode :

```
tool-output-form-required-data ((tool atef-tool)
          ouput-form-name)                                    [méthode]
```

Par exemple :

```
(defmethod tool-output-form-required-data (tool atef-tool)
                                          (ouput-form-name (eql :q-graph))
'(:morphe :engine-state)
```

Le moteur fait l'union des informations qui lui sont demandées (en n'oubliant pas les informations dont il a lui-même besoin) et ne calcule que celles-là.

7.2.3. Fonctions de contrôle du non-déterminisme

Les fonctions de contrôle[73] agissent sur plusieurs éléments du moteur :

- la pile de formes ;
- la pile de dictionnaires ;
- la pile de morphes ;
- le pile de règles ;
- la forme interne.

Nous avons vu que les fonctions de contrôle modifient des résultats déjà calculés, et surtout ont une action sur les traitements à venir. L'ajout de démons et de filtres aux piles constitue une solution générique au problème de l'implémentation et permet l'extensibilité à de nouvelles fonctions de contrôle du non-déterminisme.

Trois types de démons différents peuvent être déclenchés, par trois types d'événements :

- :push-item ajout d'un élément en sommet de pile ;
- :pop-item retrait d'un élément en sommet de pile ;
- :empty-stack condition de pile vide vérifiée.

Un filtre n'autorise l'ajout d'un élément sur la pile que si cet élément satisfait un prédicat associé au filtre. Une pile dispose en permanence d'une liste de filtres. Un élément doit passer tous les filtres actifs pour pouvoir être empilé.

Nous distinguons deux types de filtres. Les filtres statiques n'évoluent pas dans le temps. Les filtres dynamiques consituent une généralisation des filtres statiques et leurs conditions peuvent dépendre de variables.

Les piles que nous manipulons peuvent disposer de variables. Ces variables sont manipulées par les démons. Le démon *push-item* n'est activé que si l'élément est passé à travers le filtre. Comme il est parfois intéressant d'associer une action à une tentative d'empilement, nous définissons un nouveau démon :

---

73 Dans ce qui suit nous ne considérons qu'une version simplifiée des fonctions de contrôle (et qui correspond à l'implémentation actuelle). Cette version ne considère pas l'application des fonctions dans le cas de sous-règles.

- :try-push-item

Si de plus, nous ajoutons des fonctions de manipulation sur le contenu des piles, nous pouvons implémenter les différentes fonctions de contrôle du non-déterminisme. Par exemple, considérons la fonction FINAL:

Il est possible que des solutions complètes aient précédemment été calculées. Nous devons diviser notre analyse en deux parties. Comment modifier ce qui a déjà été calculé (altération statique) et comment contraindre ce qui va l'être (altération dynamique) ?

- altération statique

Il faut supprimer toutes les règles en attente, tous les morphes en attente et tous les dictionnaires en attente. Il faut aussi modifier les segmentations de la forme courante dans la structure interne. Pour cela, il faut supprimer tous les nœuds frères du nœud correspondant au morphe courant.

- altération dynamique

Le calcul s'arrêtera à la première solution pour le morphe suivant. Quand une règle est dépilée, le démon *:pop-item* vérifie si la règle n'était pas une sous-règle et si elle s'est bien appliquée avec succès (ces deux conditions indiquent qu'une segmentation a réussi). Si oui, il supprime tous les dictionnaires, morphes et règles en attente. Il faut également invalider toutes les segmentations en attente (sauf la segmentation courante) dans la forme interne.

Pour la fonction ARF, nous avons :

- altération statique

Suppression de toutes les règles de la pile des règles. Suppression de tous les morphes plus courts dans la pile de morphes. Suppression de tous les dictionnaires de la pile des dictionnaires.

- altération dynamique

Rien

7.3. Protocoles et classes

L'ensemble des classes d'ATEF hérite d'une classe abstraite *ATEF*. Cela permet d'outiller toutes les classes d'ATEF en factorisant le code nécessaire au sein de cette classe abstraite. Certaines des classes héritent également d'autres classes. Les protocoles sont catégorisés selon leur fonction. Ceux présentés ici ne disposent en général que d'un seul taxon qui est une hiérarchie de classes liées à *ATEF*.

7.3.0. Principes généraux

Pour l'analyse par objets, nous respectons les principes suivants :

- Un protocole ne définit qu'un seul taxon (c'est-à-dire une seule hiérarchie de classes) ;
- Cette hiérarchie de classes est dominée par une classe abstraite qui donne son nom au protocole ;

- Tout objet qui répond au protocole doit être une sous-classe (pas nécessairement directe) de la classe abstraite qui représente le protocole.

Ces principes permettent d'outiller les protocoles. En effet, il est intéressant de pouvoir non seulement lister les fonctions (au sens large) définies dans un protocole mais aussi de les définir ou de les supprimer dynamiquement.

En effet, si deux classes répondent au même protocole, il peut être intéressant de les chapeauter sous une classe abstraite commune. Il s'agit en fait plus d'une question d'outillage des protocoles que d'analyse liée à ATEF.

Nous verrons dans la suite que nous avons intérêt à outiller les protocoles indépendamment des classes, mais que cela induit une définition assez lourde pour leur gestion[74]. Les objets virtuels que sont les protocoles auront peut-être, dans certains cas, intérêt à se concrétiser sous forme d'instances de classes.

7.3.1. Objets linguistiques

Les objets linguistiques se divisent en deux grandes catégories, les collections (d'après Booch) et les éléments qui composent les collections. Certaines collections peuvent avoir des éléments hétérogènes.

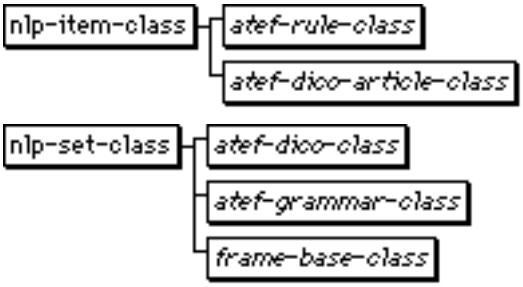


Figure 7.15 : Hiérarchie de classes linguistiques.

Ces classes réalisent les dictionnaires et les grammaires ainsi que leurs composants respectifs, les articles et les règles. Les bases contenant des décorations sont aussi comprises dans cette hiérarchie.

Les collections sont typiquement :

- les dictionnaires;
- les grammaires ;
- les ensembles de décorations.

Les éléments composent les collections. Il s'agit des règles pour les grammaires, des articles pour les dictionnaires et des décorations pour les bases de décorations (les classes de décorations ne sont pas représentées sur le figure 7.15). Les décorations sont bien un taxon du protocole sur les collections et les éléments, mais elles n'appartiennent pas à la même hiérarchie de classes.

Les collections répondent au protocole DOP, et peuvent également être des taxons de protocoles spécialisés.

---

74 Cela amène à définir un méta-protocole sur les protocoles.

Règles

La distribution des règles se fait sur la base d'un protocole qui définit des handlers. Les handlers sont appelés par le système à des étapes précises du traitement. Il est possible de spécialiser ces handlers de façon à manipuler les différentes piles de manière appropriée.

Une pile annexe a été définie pour la gestion des règles : la pile de résultats. Cette pile contient des informations liées aux dernières évaluations de règles. Deux conditions doivent toujours être vérifiées avant de modifier les piles de règles et de résultats :

- la dernière règle appliquée est au sommet de la pile quand le handler est appelé. Le sommet de la pile est dépilé après l'appel du handler. Ce n'est donc pas au handler de supprimer le sommet de la pile ;
- le dernier résultat qui est l'état courant en cas de succès, ou NIL en cas d'échec, est toujours placé au sommet de la pile des résultats. Le résultat est dépilé après l'appel du handler.

Le protocole est basé sur les fonctions génériques suivantes, dont l'appel est géré par le régulateur de règles. Pour définir de nouveaux modes de distribution de règles, il suffit de spécialiser ces handlers sur l'argument *mode*. L'argument *stack-item* contient toutes les informations qui peuvent être nécessaires à la distribution : la règle courante, le niveau d'appel (0 si ce n'est pas une sous-règle), l'état des variables du système, un drapeau *done* qui indique si la règle a été traitée, et la liste des résultats des sous-règles (qui n'est valide que si *done* est à vrai).

Les handlers spécialisables sont les suivants :

```
dispatch-handler-body-success (rule-dispatcher stack-item mode)
                                                              [méthode]
```

Ce handler est appelé quand le corps de la règle courante a été appliqué avec succès. Les sous-règles (s'il y en a) doivent être empilées sur la pile des règles.

```
dispatch-handler-body-failure (rule-dispatcher stack-item mode)
                                                              [méthode]
```

Ce handler est appelé quand l'application du corps de la règle courante a échoué (la condition de la règle n'a pas été vérifiée). Normalement, les sous-règles ne seront pas empilées et on doit laisser la règle courante être (automatiquement) dépilée du sommet de pile.

```
dispatch-handler-body-subrule-completed
         (rule-dispatcher stack-item mode)                    [méthode]
```

Ce handler est appelé quand l'application d'une sous-règle a été effectuée entièrement (c'est-à-dire corps et "sous-sous-règles" éventuelles).

```
dispatch-handler-body-rule-completed
         (rule-dispatcher stack-item mode)                    [méthode]
```

Ce handler est appelé quand la règle courante a entièrement été appliquée (corps et sous-règles).

stack-item est l'élément de la pile dont on est train d'évaluer la règle (il n'est pas forcément en sommet de pile). Cet objet est accessible par le protocole *object-data* avec les sélecteurs suivants :

- 'rule — la règle ;
- 'level — le niveau d'appel de la règle. Ce nombre est supérieur à 0 s'il s'agit d'une sous-règle ;
- 'state — l'état du moteur avant l'appel de cette règle ;
- 'body-result — résultat de l'évaluation du corps de la règle. Vaut T si le corps à réussi à s'appliquer, NIL autrement. Si le corps n'a pas été évalué vaut le symbole :null ;
- 'subrule-results — liste de résulats d'évaluations des sous-règles. Les mêmes conventions qu'au-dessus s'appliquent ;
- 'rule-result — résultat de l'évaluation de la règle (corps et sous-règles).

La réalisation du mode de distribution par défaut est relativement simple (le code a été simplifié afin de servir d'illustration) :

```
(defmethod dispatch-handler-body-success ((self rule-dispatcher)
                                          item (mode (eql :or)))
  (cond
   ((atef-rule-has-subrules? (get-object-data item 'rule))
    (push-subrules self)
    )
   (t nil)
   ))
```

La règle a réussi : si elle a des sous-règles, il est alors nécessaire de les empiler sur la pile (nous ne présenterons pas les sous-fonctions qui effectuent le détail de ces actions). Sinon, il n'y a rien à faire.

```
(defmethod dispatch-handler-body-failure ((self rule-dispatcher)
                                          item (mode (eql :or)))
  (declare (ignore item))
  (engine-restore-state  (owner self)
                         (get-object-data item 'state))
  )
```

En cas d'échec du corps de la règle, il faut restaurer l'ancien état courant, qui est contenu dans le champ *'state* de *item*.

```
(defmethod dispatch-handler-rule-completed ((self rule-dispatcher)
                                            item (mode (eql :or)))
  (declare (ignore mode))
  (cond
   ((= (get-object-data item 'level) 0)
    (let* (   (subrule-results (get-object-data item 'subrule-results))
              (success (or-list subrule-results)) )
      (set-object-data item 'rule-result success)
      (engine-gather-result (owner self) :success success) )
    )

   ((> (get-object-data item 'level) 0)
     (let* (  (subrule-results (get-object-data item 'subrule-results))
              (success (or-list subrule-results)) )
      (set-object-data item 'rule-result success) )
    )

   (t (error "level ~s is not correct." (get-object-data item 'level))))
```

```
;; don't forget to call the next-method (restore the state)
(when (next-method-p) (call-next-method))
)
```

Si la règle est complétée, on doit distinguer s'il s'agit d'un appel de sous-règle ou non (le niveau d'appel est égal à 0 dans ce dernier cas). S'il s'agit d'une règle appelée à la suite d'une consultation de dictionnaire, nous avons un résultat valide pour la segmentation (échec ou réussite) que l'on doit valider dans l'élément de pile et que le moteur doit récupérer. Si on a une sous-règle, on doit seulement valider le résultat.

```
(defmethod dispatch-handler-subrule-completed ((self rule-dispatcher)
                                               item (mode (eql :or)))
  (declare (ignore item)))
```

Si une sous-règle a été entièrement évaluée, on ne fait rien. Dans le cas d'une distribution *ou-alors*, il serait nécessaire de supprimer les sous-règles restantes dès la réussite de l'une des sous-règles. Les autres sous-règles ne doivent pas être calculées.

Le protocole de distribution des règles permet d'offrir un contrôle généralisé par rapport au modèle original. La définition des handlers est restée relativement simple; mais la complexité du système a augmenté.

Le protocole doit s'accompagner de conditions d'entrée et de sortie des handlers. Ces conditions pourraient être vérifiées à l'exécution, ce qui n'est pas la cas actuellement. Nous pensons qu'il est inutile de rajouter ce niveau de complexité. En effet, la vérification dynamique complique encore davantage le code et s'avère coûteuse à l'exécution. Elle est superflue si le développeur respecte bien le protocole.

Articles de dictionnaires

Contrairement aux règles, les articles de dictionnaires (les morphes) sont passifs. Ils se contentent de retourner les informations qui leur sont demandées. Ces informations sont accessibles par le protocole *object-data* sur au moins les sélecteurs suivants : *:reflex*, *:rule-selector*, *:formats* et *:key*. Les spécialisations des classes d'article de dictionnaires peuvent fournir d'autres sélecteurs.

7.3.2. Moteur et régulateurs

Le moteur dispose de 5 régulateurs (de formes, de segmentations, de dictionnaires, de morphes et de règles). Il doit connaître à chaque instant les élements sur lesquels portent les traitements (forme courante, segmentation courante, dictionnaire courant, morphe courant et règle courante). De plus, il dispose d'un certain nombre d'options : direction de segmentation, liste d'informations à conserver dans la forme interne, etc.

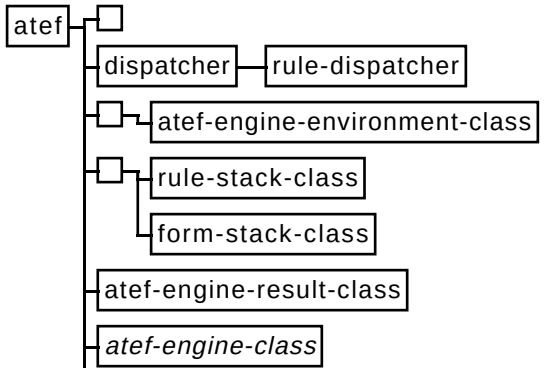


Figure 7.16 : Hiérarchie de classes ATEF liées au moteur.

Mis à part, l'initialisation et l'exécution, le moteur répond principalement à des protocoles liés au boucles.

```
engine-loop ((engine atef-engine-class) loop-kind
          &key &allow-other-keys)                          [méthode]
```

Cette méthode peut être spécialisée sur le type de la boucle. Les mots-clés sont ouverts de façon à pouvoir passer des informations complémentaires (par exemple sur des modes de parcours des éléments par la boucle).

Pour ATEF, nous avons spécialisé ces méthodes sur *loop-kind* avec les valeurs suivantes :

- :forms
- :segments
- :dictionaries
- :morphs
- :rules

Un régulateur se charge de répondre à une méthode *get-next-item* mais c'est également lui qui gère les appels aux objets. Par exemple, l'exécution d'une règle se fait par la méthode suivante :

```
dispatcher-do ((dispatcher rule-dispatcher)
          (action-name (eql :execute-next-rule)))          [méthode]
```

On demande au corps de la règle de s'exécuter et on obtient son résultat (NIL en cas d'échec). Le champ body-result est mis à jour.

Si le niveau de la règle est différent de 0, il s'agit d'une sous-règle et on ajoute le résultat de l'exécution de la sous-règle à l'élément de la pile des règles qui contient la règle appelante. S'il ne s'agit pas d'une sous-règle, on met à jour son résultat, on déclare l'objet en sommet de pile comme ayant été évalué.

À ce moment est appelée, s'il y en a une, la fonction de contrôle du non-déterminisme. Les piles peuvent être modifiées dans leur contenu ou leurs filtres.

Enfin, on distribue les règles (méthode *dispatch*).

```
dispatch  ((dispatcher rule-dispatcher))                   [méthode]
```

Tous les régulateurs répondent à ce protocole. Pour le régulateur de règles, il s'agit d'appeler les handlers *dispatch-handler-body-sucess* ou *dispatch-hanlder-*

body-failure. Il est ensuite nécessire de "réduire la pile". La réduction de pile consiste à évacuer toutes les règles complètement évaluées. C'est à ce moment que les handlers *dispatch-handler-ruled-completed* et *dispatch-hanlder-subrule-completed* sont appelés.

7.3.3. Piles

Des classes de piles ont été définies pour ATEF à partir de la classe générique *filtered-stack.*

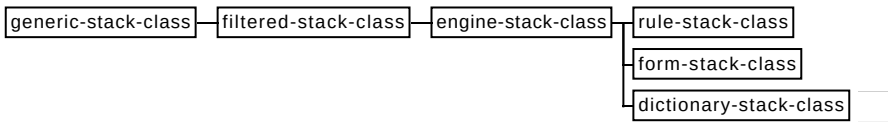


Figure 7.17 : Hiérarchie de classes de piles.

Les piles sont utilisées par certains régulateurs. Nous avons défini les piles de façon très générique, à tel point que le nom de "pile" est peut-être un peu réducteur.

```
stack-push! ((pile generic-stack) object)                    [méthode]

stack-pop! ((pile generic-stack))                            [méthode]

get-object-data ((pile generic-stack) (selecteur (eql :top)))
                                                             [méthode]

stack-purge! ((pile generic-stack)&key test)                 [méthode]
```

Pour le vidage de pile, un *test* peut être précisé. Un élément de la pile doit le vérifier pour être supprimé. Par défaut, toute la pile est vidée (le prédicat vaut toujours T).

Une extension consiste à considérer une pile comme pouvant aussi être accédée "par le bas" (comme une file). Il est donc possible de modifier à l'exécution le mode d'accès (qui est par défaut à *:top*):

```
get-object-data (pile (selecteur (eql :access)))             [méthode]

set-object-data (pile (selecteur (eql :access)) valeur)      [méthode]
```

Les valeurs acceptables pour *valeur* sont les symboles *:top* et *:bottom.* La pile de formes est en accès par le bas (*:bottom*). Elle se comporte en fait comme une file.

Il est également possible de trier les éléments de la pile, d'en insérer ou d'en éliminer.

Il est possible de spécifier un démon pour une pile avec :

```
stack-add-deamon (pile type-demon function)                  [méthode]
```

L'argument *function* demande une fonction d'un seul argument obligatoire. L'argument de cette fonction sera la pile. Dans le cas d'ATEF, les démons ont accès aux autres piles ou à la forme interne qu'ils doivent modifier. En effet, ces objets ont des liens vers les autres piles d'ATEF (il s'agit de spécialisations de piles génériques auxquel nous avons ajouté des attributs pour réaliser ces liens).

L'argument type-demon peut prendre une des valeurs : *:push-item, :pop-item, empty-stack* et *:try-push-item* .

`stack-remove-deamon (pile type-demon)` [méthode]

Détruit le démon *type-demon* pour la pile.

`stack-deactivate-deamon (pile type-demon)` [méthode]

Désactive (mais ne détruit pas) de démon *type-demon* pour la pile.

`stack-activate-deamon (pile type-demon)` [méthode]

Active le démon *type-demon* pour la pile.

Le mécanisme d'activation/désactivation permet de définir des collections de démons dont l'utilisation peut se faire à discrétion. Les démons actifs sont appelés dans les corps des méthodes présentées ci-dessus.

Le protocole pour les filtres est similaire à celui pour les démons[75] :

`stack-add-filter (pile filter-name predicat)` [méthode]

`stack-remove-filter (pile filter-name)` [méthode]

Pour une pile donnée, les filtres doivent avoir un nom unique.

`stack-deactivate-filter (pile filter-name)` [méthode]

`stack-activate-filter (pile filter-name)` [méthode]

Par défaut, un objet passe au travers des filtres si l'évaluation de chaque prédicat retourne vrai. Il s'agit d'un *et* logique. D'autre fonctions logiques peuvent être définies à l'aide de (meta-)fonctions qui permettent une telle extensibilité.

Le reste de l'implémentation d'ATEFlisp suit les mêmes principes. S'il fallait être complet, il aurait encore fallu détailler :

- la production du graphe quadrichrome arrière (forme interne) et le traitement des conditions contextuelles arrières ;
- le traitement des conditions et (surtout) des affectations de S (successeur) ;
- le traitement des tournures (reconnaissance et fonctions ELIM et ELIT) ;
- le traitement des mots composés.

75 On remarquera que les filtres sont en fait également des démons, similaire au démon :*restriction* du GFP.

Chapitre 8

ROBRA - Un transducteur d'arbres

L'analyse et la réingénierie du LSPL ROBRA nous permet de généraliser ce qui a été fait pour ATEF. Une hiérarchie de classes d'objets unifiant les composants des deux langages est élaborée. Nous voyons ces classes comme les premiers éléments d'une boîte à outils pour le TALN. Cependant, certains des composants d'ATEF ne sont pas adaptés à ROBRA et ont été remplacés par d'autres composants. C'est le cas de la définition interne du moteur.

Le moteur de ROBRA ne se base plus sur une série de boucles inbriquées, mais sur un automate. Cet automate est la réalisation du graphe de contrôle défini par le linguiste pour une application donnée. C'est l'occasion d'étudier une nouvelle technique d'extensibilité par fusion de classes.

8.1. Modèle original

ROBRA est un langage d'écriture de systèmes transformationnels agissant sur des arborescences étiquetées. En ARIANE, ce LSPL est utilisé pour écrire les trois phases d'analyse structurale ("multiniveau"), de transfert structural et de génération structurale (voir figure 2.5).

Pour une description du langage ROBRA, on peut se référer à [Boitet & al. 1982] et à [Chappuy 1983].

On peut considérer ROBRA comme un langage algorithmique de très haut niveau, mais il s'agit bien d'un LSPL qui dispose de structures de données et de contrôle particulières. En plus d'un système de décorations similaire à celui d'ATEF, les structures de données sont augmentées d'arborescences étiquetées. Les structures de

contrôle comprennent des affectations conditionnelles (à non-déterminisme unaire en profondeur d'abord), le parallélisme dans l'application des règles d'une grammaire, l'itération contrôlée de grammaires, le non-déterminisme du parcours du graphe de contrôle (unaire en profondeur d'abord), et différents modes de récursivité.

Dans les configurations normales, le modèle est décidable. Il est possible de détecter statiquement les sources d'indécidabilité dans les autres configurations.

8.1.0. Présentation générale

Cette description est tirée de [Boitet & al. 1982].

Un système transformationel (ST) opère sur une arborescence objet (AO) étiquetée sur un ensemble dénombrable . En ROBRA, est l'ensemble des décorations possibles. Un ST est défini par un graphe de contrôle (GC), un ensemble de grammaires transformationnelles (GT) et un ensemble de règles (RP). Une GT est un ensemble ordonné de règles, et un GC est un graphe à nœuds étiquetés sur GT∪{∧} (∧ est la sortie), dont les arcs peuvent porter des conditions sur T , l'ensemble des AO possibles.

L'exécution d'un ST consiste à utiliser le GC comme structure de contrôle non-déterministe unaire. Depuis un nœud initial unique, on cherche le premier chemin qui mène à une sortie (∧). Sur ce chemin, on exécute les grammaires contenues dans les nœuds, et, pour pouvoir traverser un arc, il faut que l'AO courant vérifie la condition portée par cet arc.

8.1.1. Graphe de contrôle

Si une règle appliquée comporte un appel récursif à un sous-système transformationel, l'arborescence soumise à la récursion est traitée par le même GC en prenant le nœud correspondant à la grammaire comme nœud inital. S'il n'existe aucun chemin possible, l'arbre de sortie sera identique à l'arbre d'entrée.

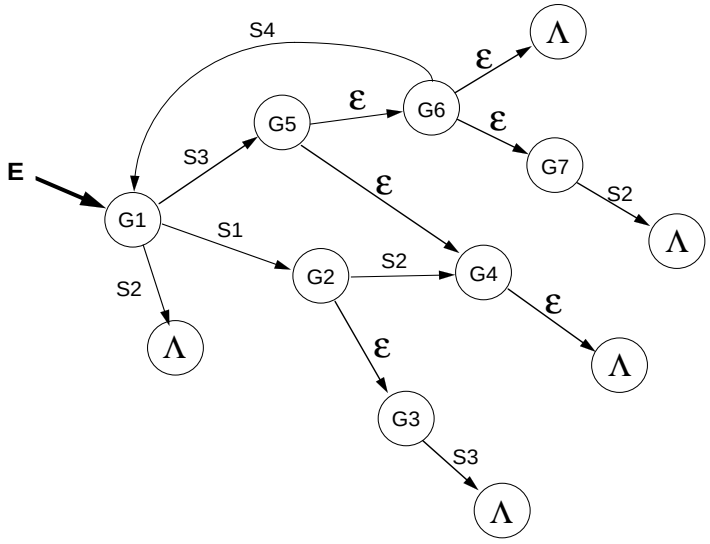


Figure 8.1 : Exemple de graphe de contrôle en ROBRA.

Si le schéma d'arbre S1 apparaît dans l'application de l'AO par G1 (notée G1(AO)), on calcule G3(G2(G1(AO))) = A3. Si S3 apparaît dans A3, le système s'arrête et produit A3. Sinon, il revient en arrière, à A2. Si alors, S2 apparaît dans A2, G4(A2) est produit, etc.

8.1.2. Grammaires transformationnelles

Une grammaire transformationnelle (GT) est un ensemble ordonné de RP. Cet ordre est propre à chaque grammaire et permet de résoudre des conflits entre les règles.

Règle

Une règle de production (RP) se compose d'un nom, d'une partie gauche (le schéma) et d'une partie droite (l'image).

Le schéma est la condition de la règle et l'image correspond à l'action. Le schéma décrit la famille de sous-arbres que la règle peut réécrire. Il se compose d'une description géométrique et d'une description algébrique (des conditions sur les décorations).

L'image se compose d'un sous-arbre image, d'une fonction de transfert qui indique où rattacher les dépendants des sommets qui n'apparaissent pas dans le schéma, et d'une partie affectation qui calcule les nouvelles valeurs des décorations des sommets images.

```
MDEVEN:   (*0,&NIV=1) (0(1($A,2,*),*($B,4),*,5(*V))/
            1: $MODAL: 4: UL-E-"BE"; 5: SUBV-INC-VEN; $A; ADV; $B;
            $ADV
        ==  0(5($A, 2, $B, $V)) / * <-- 1,3,4 /
            2: 2, SF:=AUX;
            5: 5,  K:=VCL, VOICE:=PAS, UL:= "*VCL", NEG:=NEG(1)
                   SUBV:=VB, NUM:=NUM(1)
                   -SI- UL(2) -E- "COULD" -OU- UL(2) -E- "MIGHT"
                   -ALORS- TENSE(5):=COND -U- PRES
                   -SINON TENSE(5) := TENSE(2)
                   -FSI-
```

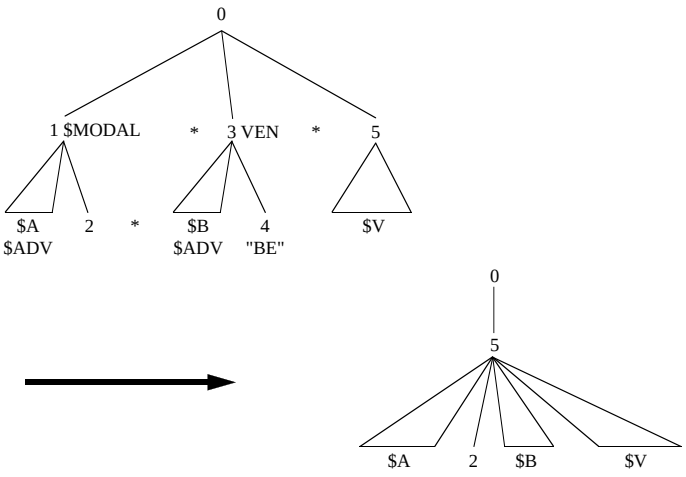


Figure 8.2 : Exemple de règle ROBRA.
(tiré de [Chappuy 1983])

Modes d'exécution

Une GT dispose de plusieurs *modes d'exécution* qui peuvent contrôler l'itération ou le mode de choix entre plusieurs occurrences (de règles) possibles :

- Le mode impératif (I) ou le mode facultatif (F - mode par défaut).

Le mode I demande qu'au moins une règle ait été appliquée lors d'un passage par une GT. Cette règle peut être l'identité. Sinon, le chemin dans le GC est bloqué et l'automate revient en arrière. En mode F, la grammaire peut ne réaliser aucune transformation ;

- Le mode zéro (Z) ou normal (N - mode par défaut)

  Dans le mode Z, l'automate ne peut pas revenir en arrière. Le travail effectué précédemment ne peut être détruit. Le nœud correspondant du GC devient donc déterministe ;

- Le mode unitaire (U - mode par défaut) ou exhaustif (E).

  En mode E, on réitère les applications élémentaires jusqu'à ce qu'il ne reste plus aucune règle applicable. En mode U, on effectue une seule application (toujours parallèle) ;

- Le mode libre (L) ou gardé (G - mode par défaut).

  En mode gardé, si une règle est appliquée sur un nœud, ce nœud est marqué et cette règle ne peut plus s'appliquer sur ou sous ce nœud. En mode libre, les marques ne sont pas considérées comme des interdictions ;

- Le mode bas (B - mode par défaut) ou haut (H).

  En mode B, la préférence est donnée aux occurrences de règles les plus profonde dans l'arbre, par opposition au mode H (préférence aux occurrences les plus "hautes") ;

- Le mode coupe (C - mode par défaut) ou total (T).

  En mode C, aucune occurrence de règle ne peut en dominer une autre, alors qu'en mode T les occurrences peuvent être "imbriquées" (dépendre les unes des autres) ;

- Le mode dessous (D - mode par défaut) ou ponctuel (P).

  En mode dessous, la règle appliquée, et toutes les règles de rang inférieur sont marquées sur la racine de l'image. En mode P, seule la règle appliquée est marquée sur la racine de l'image ;

D'autre modes (dits "de trace") existent et facilitent la mise au point d'une application écrite en ROBRA par le développeur linguiste.

Application élémentaire simple

Une application élémentaire simple consiste à exécuter une seule fois en parallèle plusieurs règles de la GT. Il faut préciser la forme générale de la famille d'occurrences souhaitée (mode coupe ou total) et les règles de résolution de conflits. Pour appliquer simultanément deux occurrences de règle, il faut que les occurrences de leurs schémas n'aient pas de points actifs communs. Dans ce cas, on dira que les occurrences sont indépendantes.

Trois étapes sont nécessaires à une application simple :

- Recherche des occurrences des règles de la GT dans l'arborescence objet (AO).

Une occurrence d'une règle R est un sous arbre de AO qui correspond au schéma de RP. R peut avoir plusieurs occurrences distinctes dans AO.

- Sélection de la sous-famille d'occurrences applicable.

La sous-famille est composée d'occurrences indépendantes et répond aux critères liés aux modes. En mode total, une occurrence peut en dominer une autre. Ce n'est pas le cas en mode coupe. Un parcours canonique associé à une GT est utilisé pour cette sélection (mode haut ou bas).

- Application en parallèle et marquage

Les racines de transformation (RT) images par une ou plusieurs règles sont marquées lors de l'application. Chaque point de l'AO est muni d'un ensemble, initialement vide, de "marques" de règles. Après l'exécution d'une occurrence O_k de la règle R_i, on ajoute à l'ensemble de marques de l'occurrence de la racine de transformation $O_k(RT_i)$, la (ou les) marques :

R_i si la GT est en mode ponctuel (P) ;

les (marques) R_j telle que "$j \leq i$ dans la GT" si la GT est en mode dessous (D).

Les marques sont effacées quand on passe d'un nœud à l'autre du graphe de contrôle. Lors d'un appel récursif, on efface sur la racine de l'arbre soumis à la récursion les marques des règles qui interviennent dans cette récursion. Au retour de la récursivité, les nouvelles marques seront ajoutées aux anciennes.

Application élémentaire itérée

L'application itérée d'une GT dépend principalement du mode "libre" ou "gardé".

- en mode libre et exhaustif (L et E) :

Les applications élémentaires sont réitérées jusqu'à ce qu'il ne reste plus aucune règle applicable.

- en mode gardé (G) :

Si R_k est marquée sur un nœud, R_k ne peut être utilisée sur ce nœud ou sous ce nœud. Le nombre d'applications potentielles diminue, et l'itération s'arrête nécessairement.

Application élémentaire récursive

Dans une grammaire G, une règle R peut appeler récursivement :

- R(G/<l>/sommets validés)

  la sous-grammaire d'une grammaire G', définie par une sous-liste de règles <l> et par les mêmes modes d'exécution ;

- R(G'/*/sommets validés)

  une grammaire G' (cas particulier du précédent) ;

- R(*G'/*/sommets validés)

le sous-système transformationnel dont le nœud inital est G'.

Ce type d'appel récursif permet d'utiliser différentes stratégies sur diverses parties d'une unité de traduction.

Une règle peut avoir un appel récursif et un seul. Pour l'exécution d'un appel récursif, le système "détache" le sous-arbre dominé par la racine de transformation R_t et le traite comme un nouvel arbre objet. les sommets images non-validés ne peuvent participer activement à aucune transformation durant l'appel (le nombre des occurrences potentielles diminue donc à chaque appel). Le résultat de l'appel remplace le sous-arbre d'entrée dans l'arbre origine.

8.2. Recherche de l'extensibilité et de la généricité : ATEF comme base de départ

L'analyse et l'implémentation de ROBRA sont parties des acquis du travail réalisé avec ATEF. Nous avons procédé en deux étapes. Le première a consisté à généraliser ce qui avait été fait pour ATEF, et la seconde à définir des spécialisations adéquates pour ROBRA.

Certaines parties d'ATEF ne nous semblaient pas directement applicables à ROBRA (la structure des règles, par exemple). Nous avons donc décidé de rendre modulaires des composantes d'ATEF qui ne l'étaient pas et semblaient généralisables à ROBRA. Le travail fait sur ROBRA a consisté à trouver des "pièces de remplacement" et à définir un protocole pour rendre effective la composition.

8.2.1. Objets remplaçables

Le fait que le modèle classique agisse sur des arborescences étiquetées ne constitue en rien un aspect obligatoire des systèmes transformationnels. Cependant, la forme des règles ainsi que certains des modes d'application d'une grammaire transformationnelle de ROBRA sont dépendent du choix d'arbres comme structures de données de base.

La définition d'un nouveau LSPL, MATCHES, est en cours. Ce langage permet de reconnaître des schémas de chaînes de caractères, de graphes ou d'arbres. Les graphes et les arbres pouvant être décorés, il est nécessaire que MATCHES puisse avoir accès à DÉCOR. La composition de ces deux langages passe, encore une fois, par la définition d'un protocole.

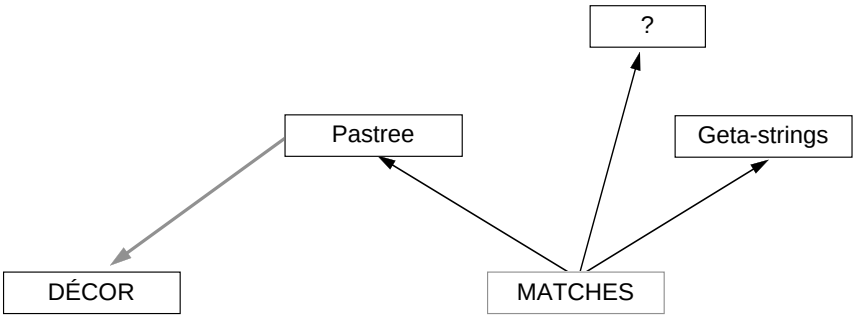


Figure 8.3 : Vers des combinaisons de langages génériques.

Un langage de filtrage, MATCHES, conçu dans le même esprit que DÉCOR, vise à fournir un noyau qui peut être dérivé vers plusieurs langages travaillant sur des structures de données particulières. Des travaux ont déjà été effectués sur les chaînes avec [Lafourcade 1992a]. Un langage de filtrage sur les arbres, PASTREE, est également l'objet d'une spécification qui lui permet d'être facilement combinable avec DÉCOR.

8.2.2. Classes de base

L'ensemble des protocoles et des classes de base de l'outil ATEF$_{lisp}$ a été étendu à ROBRA$_{lisp}$.

La décomposition des composants de ROBRA est similaire à celle d'ATEF. Les règles de ROBRA n'utilisent pas directement une spécialisation de DÉCOR. Le langage de filtrage sur les arbres, PASTREE est une spécialisation de MATCHES qui est un langage de filtrage général[76]. PASTREE utilise DÉCOR pour les conditions sur les décorations. Un autre langage, de manipulation d'arbres cette fois, est nécessaire. Nous avons choisi d'utiliser dans un premier temps, TTEDIT [Durand 1988] qui est également l'objet d'une expérience de réingéniérie.

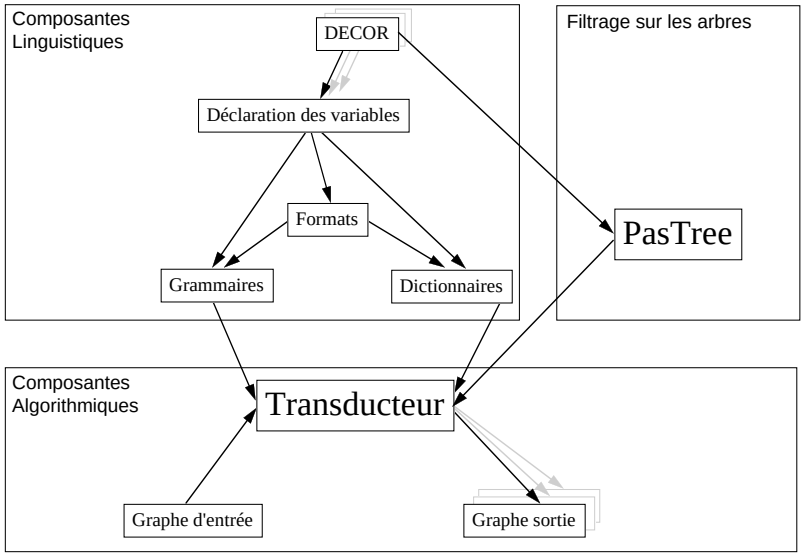


Figure 8.4 : Décomposition des éléments linguistiques et algorithmiques dans ROBRA$_{lisp}$.

La structure du moteur de ROBRA est relativement éloignée de celle d'ATEF. Le moteur répond toutefois au même protocole externe. Le moteur de ROBRA est basé sur un automate qui correspond à la définition donnée par l'utilisateur pour le graphe de contrôle.

Ce moteur utilise les mêmes protocoles qu'ATEF pour l'accès aux règles.

8.2.3. Généralisation du contrôle

Dans ROBRA, le contrôle s'exprime principalement lors du parcours du graphe de contrôle, et lors de l'exécution d'une grammaire transformationnelle.

Graphe de contrôle

Un nœud en mode zéro est déterministe. Le mode par défaut est un non-déterminisme unaire. On pourrait aussi souhaiter un non-déterminisme n-aire. Nous décidons de renommer ces modes d'exécution du graphe de contrôle respectivement 0, 1 et n. Plusieurs approches sont possibles pour modifier les classes du gestionnaire d'automate.

---

76 Il s'agit plutôt d'un protocole général sur le filtrage.

La première solution consiste à ajouter une option définissant un mode de fonctionnement non-déterministe dans la classe d'automates. Cette idée ne nous semble pas satisfaisante dans la mesure où nous souhaitons garder des comportements simples pour chaque classe. De plus, il est généralement admis que si le comportement d'une classe dépend d'un paramètre, mieux vaut définir deux classes distinctes. Enfin, ce type d'option pose des problèmes d'extensibilité.

La seconde solution consiste à créer une nouvelle classe d'automates non-déterministes. Le gestionnaire d'automates dispose alors de deux classes d'automates, une déterministe et l'autre non. Les deux classes peuvent hériter d'une même classe abstraite qui factorise le code nécessaire. Selon les options demandées, la bonne classe peut être sélectionnée automatiquement au moment de l'instanciation. Nous pouvons par exemple avoir les deux classes d'automates :

```
automata-class                                              [classe]
1-nd-automata-class                                         [classe]
n-nd-automata-class                                         [classe]
```

et la fonction:

```
select-automata-class (&key nd &allow-other-keys)          [fonction]
```

Cette méthode retourne le symbole qui désigne la classe d'automates à instancier en fonction de la valeur du mot-clé *nd*. *0* signifie que l'automate est déterministe. *1* signifie que l'automate est déterministe unaire et le symbole *'n* déterministe n-aire.

La définition de *select-automata-class* est alors une analyse par cas :

```
(defmethod select-automata-class (&key (nd 0) &allow-other-keys)
  (cond
    ((= nd 0)     'automata-class)
    ((= nd 1)     '1-nd-automata-class)
    ((eql nd 'n)  'n-nd-automata-class)
    (t (error …)))
 )
```

Cette technique pose au moins deux problèmes. Le premier est qu'il faut modifier la méthode de sélection de la classe dès qu'un nouveau comportement est identifié. Cette modification affecte non seulement la signature de la méthode (ajout de nouveaux mots-clés) mais aussi sa définition (ajout de nouveaux cas). Le second problème réside dans la définition des nouvelles classes qui correspondent à de nouveaux comportements. Si, en plus du non-déterminisme, on considère la récursivité, on obtient la conbinatoire de classes suivante:

```
automata-class                                              [classe]
1-nd-automata-class                                         [classe]
n-nd-automata-class                                         [classe]
rec-automata-class                                          [classe]
rec-1-nd-automata-class                                     [classe]
rec-n-nd-automata-class                                     [classe]
```

Nous n'indiquons pas quel est l'héritage entre ces classes. Du point de vue du partage du code, il n'est pas du tout évident que, par exemple, la classe *rec-1-nd-automata-class* hérite à la fois de *rec-automata-class* et *1-nd-automata-class*. Il

est même peu probable que cela soit le cas si cette hiérarchie de classes n'a pas été pensée comme telle dès le début (il ne s'agit pas alors d'un développement incrémental).

La troisième solution que nous envisageons (et adoptons) consiste à créer des classes fusionnantes qui *rajoutent* du comportement aux classes de base (*automaton-class*). La hiérarchie de classes envisagée pour le non-déterminisme est la suivante :

```
automata-class                                          [classe]
1-nd-automaton-mixin-class                  [classe fusionnante]
n-nd-automaton-mixin-class                  [classe fusionnante]
rec-automaton-mixin-class                   [classe fusionnante]
```

Pour chaque propriété de l'automate, une fonction de sélection indique si une classe fusionnante doit être rajoutée et laquelle. Cette fonction générique a la signature suivante :

```
select-mixin-class classe-de-base propriété valeur
                                                [fonction générique]
```

Cette fonction rend un symbole dénotant une classe fusionnante ou NIL.

classe-de-base est le symbole indiquant la classe de base à laquelle sont rajoutées les classes fusionnantes ;

propriété est le symbole dénontant la propriété concernée ;

valeur est la valeur pour la propriété.

Par exemple, pour notre automate, nous avons spécialisé la méthode de la façon suivante :

```
(defmethod select-mixin-class
                   (class-de-base (eql 'automaton-class))
                   (propriété (eql :nd))
                   (valeur (eql 1))
 '1-nd-automaton-mixin-class)

(defmethod select-mixin-class
                   (class-de-base (eql 'automaton-class))
                   (propriété (eql :nd))
                   (valeur (eql 'n))
 'n-nd-automaton-mixin-class)

(defmethod select-mixin-class
                   (class-de-base (eql 'automaton-class))
                   (propriété (eql :recurive))
                   (valeur (eql t))
 'rec-automaton-mixin-class)
```

Le cas (spécialisé sur aucun argument) par défaut rend la valeur NIL.

La fonction *compute-mixin-class-list* détermine la liste ordonnée des classes fusionnantes qui doivent être rajoutées à la classe de base. Cette fonction est appelée lors de la création de l'automate avec *make-object*.

```
(defmethod compute-mixin-class-list (base-class property-value-list)
 (let (result class-symbol)
  (dolist (pv (reverse property-value-list) result)
    (setf class-symbol (select-mixin-class base-class (first pv) (second pv)))
    (when   (and (symbolp class-symbol) (find-class class-symbol nil))
            (push result class-symbol))
  )
  result))
```

Par sécurité, on teste que la valeur rendue par *select-mixin-class* est bien un symbole correspondant à une classe.

Nous devons maintenant définir dynamiquement une nouvelle classe à partir de l'héritage multiple de la classe de base et des classes fusionnantes.

`make-class-with-mixin (base-class mixin-list)` `[fonction]`

```
(defun make-class-with-mixin (base-class mixin-list)
 (let ((name (gentemp name)))
  (eval
    `(defclass ,name (,base-class ,@mixin-list)))
 ))
```

make-class-with-mixin doit faire appel à *eval* dans la mesure où *defclass* est une macrofonction et où son appel dépend de paramètres variables.

Nous avons défini un mécanisme général qui nous permet de définir dynamiquement de nouvelles classes par combinaison d'une classe de base et de classes fusionnantes. La définition des classes fusionnantes pour l'automate implémentant le graphe de contrôle de ROBRA est présentée au §8.3.

Grammaire transformationelle

Contrairement à ATEF où une grammaire est un objet passif qui se contente de fournir des règles à des clients, ROBRA applique une GT à un arbre objet. Cette application revient à sélectionner les bonnes règles puis à les exécuter en parallèle selon différents modes.

Nous avons expérimenté l'approche par classes fusionnantes pour implémenter les différents modes d'exécution d'une grammaire transformationnelle. Tout d'abord, nous décidons que la classe de base doit implémenter un comportement correspondant à toutes les options par défaut (identifiées jusqu'à présent).

`transformational-grammar-class` `[classe abstraite]`

Puis nous définissons la sous-classe des grammaires transformationnelles qui s'appliquent sur des structures arborescentes

`tree-transformational-grammar-class` `[classe]`

Les classes fusionnantes que nous définissons sont les suivantes (*ttg* est une abréviation de *tree-transformational-grammar)* :

`imperative-execution-mode-ttg-mixin-class` `[classe fusionnante]`

`zero-execution-mode-ttg-mixin-class` `[classe fusionnante]`

`exaustive-execution-mode-ttg-mixin-class` `[classe fusionnante]`

`free-execution-mode-ttg-mixin-class` `[classe fusionnante]`

```
top-execution-mode-ttg-mixin-class              [classe fusionnante]
total-execution-mode-ttg-mixin-class            [classe fusionnante]
single-execution-mode-ttg-mixin-class           [classe fusionnante]
```

Il est maintenant nécessaire de déterminer les points d'entrée de ces classes fusionnantes, c'est-à-dire les méthodes qu'elles vont spécialiser. Pour cela, il est nécessaire de définir un protocole pour l'exécution de la classe abstraite *transformational-grammar* et de la classe *tree-transformational-grammar* et ensuite de raffiner ce protocole pour les classes fusionnantes.

Une grammaire transformationnelle s'évalue sur une donnée avec la méthode suivante :

```
tg-execute ((gt transformational-grammar-class)
            data data-kind &key &allow-other-keys)          [méthode]
```

Cette méthode retourne un code résulat de l'exécution de *gt* sur data. *data-kind* permet de spécialiser le type de données si celui-ci est implémenté sous forme de type générique[77]. Si aucune transformation n'a été effectuée, *data* est retournée inchangée.

Le résultat est une instance de la classe suivante :

```
tg-result-code                                                  [classe]
```

Le protocole *object-data* s'applique sur cette classe avec les sélecteurs suivants :

- :data la donnée après transformation ;
- :touched-p vrai si une règle s'est appliquée sur la donnée.

Notons que si *touched-p* vaut vrai, cela ne veut pas dire nécessairement que la donnée a été changée (la règle identité peut s'être appliquée).

La méthode *tg-execute* est implémentée comme suit :

```
(defmethod tg-execute ((self transformational-grammar-class)
                       data data-kind &key &allow-other-keys)
  (let ((rule-list (tg-select-applicable-rules self data data-kind)))
   (tg-apply-rules self rule-list data data-kind)
 ))
```

Avec les signatures des deux méthodes suivantes :

```
tg-select-applicable-rules ((gt transformational-grammar-class)
            data data-kind &key &allow-other-keys)          [méthode]
```

Cette méthode retourne la liste des règles de *gt* applicables sur data.

```
tg-apply-rules ((gt transformational-grammar-class)
            rule-list data data-kind &key &allow-other-keys)[Méthode]
```

---

77 Ce que nous entendons par type générique est un type d'objet pouvant être utilisé pour plusieurs codages : par exemple, les listes. Si les arbres et les graphes sont deux structures codées sous forme de listes, il n'est pas possible de les différencier dans la spécialisation d'une méthode. L'ajout d'un argument permet la spécialisation dans tous les cas.

Cette méthode applique les règles de la liste *rule-list* sur *data* . Si la liste de règles est vide, *data* est retournée inchangée.

Dans le graphe de contrôle, l'action d'un état est un appel à *tg-execute* avec comme paramètre : la grammaire transformationnelle de cet état, la donnée courante (par défaut l'arbre courant contenu dans l'automate), et l'indicateur sur la donnée.

8.3. Protocoles et classes

Nous avons défini des classes fusionnantes et modifié certains protocoles de façon à obtenir des automates non-déterministes et récursifs et implémenter les différentes modes d'exécution des grammaires transformationnelle utilisées dans ROBRA. Auparavant certaines classes d'ATEF ont été généralisées à ROBRA.

8.3.0. Principes généraux

La généralisation des classes nous a amené à définir une classe abstraite, *nlp-object-class,* qui chapeaute l'ensemble des classes. De plus, nous définissons des classes abstraites pour les outils, les moteurs, et les environnements. Les environnements sont des zones de stockage d'informations pour les outils ou les moteurs.

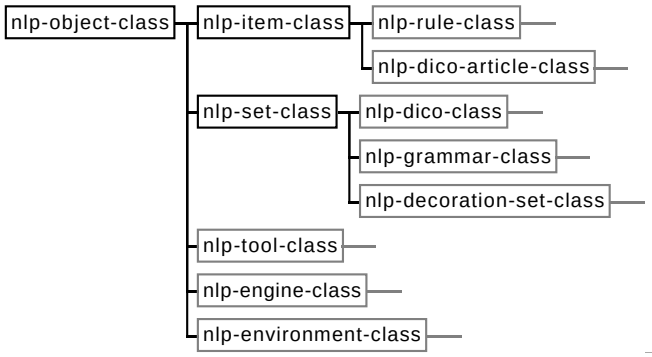


Figure 8.5 : Hiérarchie de classes abstraites pour les objets linguistiques et les composants de moteurs.

Enfin, nous avons défini les classes propres à ATEFlisp et à ROBRAlisp comme sous-classes de ces classes abstraites. La conception d'un nouveau LSPL qui correspond à ce modèle définira des classes membres de cette hiérarchie.

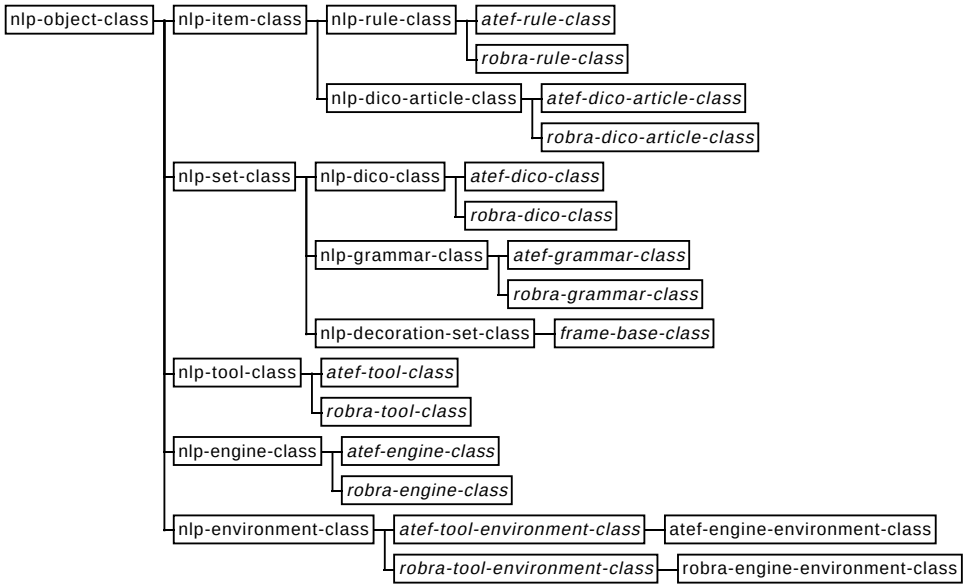


Figure 8.6 : Hiérarchie de classes d'ATEF et ROBRA vue depuis nlp-object-class.

Il s'agit en fait d'une *hétérarchie,* car certaines classes héritent de plusieurs classes. Par exemple, toutes les classes d'ATEF héritent, en plus de leur classe concrète, d'une classe abstraite *atef*. Il en est de même pour les classes de ROBRA avec une classe abstraite *robra* .

8.3.1. Automate

Le graphe de contrôle est un automate généré par la définition fournie par le linguiste. Cet automate est exprimé selon les termes du gestionnaire d'automates GAM (qui à déjà été utilisé pour LT, voir §3.1.2.). Il a fallu modifier légèrement ce gestionnaire afin de pourvoir définir des automates non-déterministes et autoriser la récursivité.

Ces fonctionnalités sont indépendantes entre elles et doivent pouvoir être combinées lors de la définition d'un nouvel automate. C'est pourquoi nous ne définissons pas une classe d'automates non-déterministe ou une classe d'automates récursifs mais plutôt des extensions d'automates qui leur confèrent ces propriétés.

Ajout de la récursivité

Au niveau de l'automate, la récursivité consiste à pouvoir le relancer sur un état quelconque à partir d'une action d'un état. Pour cela, nous avons modifié la méthode principale de l'automate, *automata-run* , en lui ajoutant un paramètre optionnel *state*.

```
(defmethod automata-run ((self automata-class) &optional state)
 (if (and state (typep state 'state-class))
     (setf (next-state self) state)
     (setf (next-state self) (initial-state self))
 )
 (do () ((not (next-state self)))
  (state-run (next-state self)))
 (variable-list self)
)
```

Si le paramétre *state* n'est pas fourni, nous supposons qu'il faut lancer l'automate à partir de l'état initial.

Une fois l'appel récursif terminé (que ce soit par un échec ou une réussite), il est nécessaire de mettre à jour l'état de l'automate avant l'appel récursif, sauf les variables (qui ont pu être modifiées). Nous ajoutons donc une structure de pile (plusieurs appels récursifs peuvent s'imbriquer) à la classe d'automates par l'intermédiaire de la classe fusionnante *rec-automaton-mixin-class*:

`rec-automaton-mixin-class` [classe]

```
(defclass rec-automaton-mixin-class (automaton-class)
   ((rec-automaton-mixin-stack :initform (make-instance 'generic-stack-class)
                     :accessor rec-automaton-mixin-stack
                     :documentation "pile de recursivité"))
 )
```

L'information passée sur la pile est l'état courant de l'automate (champ *current-state*). Comme c'est au niveau d'un état que se fait l'appel récursif, nous avons choisi de définir une classe fusionnante pour la classe d'état :

`rec-state-mixin-class` [classe]

```
(defclass rec-state-mixin-class (state-class)
   ((rec-call :initform nil
              :accessor rec-all
              :documentation "appel recursif sur un état de l'automate"))
 )
```

Nous avons spécialisé la méthode *state-run* pour cette classe fusionnante :

```
(defmethod state-run :around ((self rec-state-mixin-class))
   (cond
      ((rec-call self)
       (stack-push  (rec-automaton-mixin-stack (owner self))
                    (next-state (owner self)))
       (execute-duty (action self))
       (automata-run (owner self) (rec-call self))
       )
      (t
       (when (next-method-p) (call-next-method))
       (when
         (and  (not (next-state (owner self)))
               (get-object-data (rec-automaton-mixin-class (owner self)) :top))
         (automata-run (owner self)
                       (stack-pop! (rec-automaton-mixin-class (owner self))))))
    ))
```

Si le champ *rec-call* contient un état (dans le code ci-dessus, on ne vérifie pas le type de la valeur mais cela doit être fait), on empile sur la pile de récursivité l'état suivant précédemment calculé. L'appel récursif est ensuite effectué.

S'il n'y a pas d'appel récursif, on effectue le traitement normal. Si, ensuite, il n'y a pas d'état suivant (l'automate va s'arrêter, le cas d'erreur s'étant déjà produit si nécessaire) alors nous sommes peut-être à la fin d'un appel récursif. Si la pile de récursivité n'est pas vide, c'est bien le cas. Nous relançons alors l'automate sur l'état dépilé.

Ajout du non-déterministe

Pour rendre l'automate non-déterministe, nous l'avons modifié de la façon suivante :

- une pile est rajoutée à l'automate ;
- avant d'emprunter une transition, certaines informations sont poussées sur la pile : transition empruntée, état des variables de l'automate avant le passage par la transition ;
- lors de l'arrivée dans l'état d'erreur, on relance l'automate sur l'état de départ de la transition en sommet de pile. Les valeurs des variables doivent être remises à jour. Seules les transitions après la transition en sommet de pile doivent être considérées. Le sommet de pile doit être dépilé.

Nous définissons la classe fusionnante *1-nd-automaton-mixin-class :*

`1-nd-automaton-mixin-class` [classe]

```
(defclass 1-nd-automaton-mixin-class (automaton-manager-class)
   ((automaton-stack   :initform (make-instance 'automaton-stack-class)
                       :accessor automaton-stack
                       :documentation "pile de non-déterminisme unaire"))
 )
```

L'élément de pile (*item-automaton-stack*) répond au protocole *object-data* sur les sélecteurs suivants :

- :state sauvegarde de l'état de l'automate ;
- :taken-transition transition empruntée, on peut en déduire l'état précédant à partir duquel l'automate doit redémarrer.

Nous devons également définir une classe alternative à la classe standard pour l'état d'erreur qui relance l'automate.

`1-nd-error-state-class` [classe]

```
(defclass nd-error-state-class (error-state-class))
   () )

(defmethod state-run ((self 1-nd-error-state-class))
   (when (next-method-p) (call-next-method))
   (let ((stack-item (get-object-data (automaton-stack (owner self)) :top)) )
      (if stack-item
         (progn
          (set-automata-next-state
            (get-object-data    (get-object-data stack-item :taken-transdition)
                                :begin-state))
            (restore-automata-state (get-object-data stack-item :state))
          )
         (set-automata-no-next-state self)
         )))
```

On fixe l'état suivant à l'état de départ de la transition que l'on vient de prendre et on restaure l'état des variables de l'automate.

Ces modifications sont insuffisantes. Un état standard doit marquer les transitions déjà empruntées (pour ne pas les emprunter à nouveau). Si aucune autre transition ne peut être prise, on se trouve alors dans deux cas :

- si la pile est vide, c'est un cas d'erreur et l'automate s'arrête ;
- sinon, on effectue un retour arrière.

Il faut donc spécialiser l'état standard afin d'avoir ce comportement :

`1-nd-standard-state-class` [classe]

```
(defclass 1-nd-standard-state-class (standard-state)
   ((taken-transition-list   :initform nil
                             :accessor taken-transition-list
                             :documentation "liste des transitions prises"))

(defmethod choose-transition ((self 1-nd-standard-state-class) transition-list)
   (let ((new-transition-list (difference transition-list (taken-transition self)))
         )
      (find-if #'applicable-p new-transition-list)) )

(defmethod state-run ((self 1-nd-standard-state-class))
   (when (next-method-p) (call-next-method))
   (push (next-transition (owner self)) (taken-transition-list list)) )
```

Ainsi spécialisée, la méthode *choose-transition* ne choisit une transition que parmi les transitions restantes.

Le non-déterminisme implémenté ci-dessus marche seulement quand un état (non final) n'arrive pas à trouver une transition. Cependant, si on considère le graphe de contrôle de ROBRA, on souhaite faire en sorte qu'un état "échoue". Pour cela, il suffit d'associer à un état une méthode déterminant l'échec ou la réussite. En cas d'échec, *taken-transition-list* doit être rempli avec toutes les transitions possibles. On simule ainsi un échec sur toutes les transitions.

Nous avons modifié *state-run* de façon à récupérer le résulat de l'évaluation de l'action de l'état :

```
(defmethod state-run ((self 1-nd-standard-state-class))
   (let ((result (when (next-method-p) (call-next-method))))
      (if (state-fail-p (owner self) self result)
         (setf (taken-transition-list list) (out-transitions self))
         (push (next-transition (owner self)) (taken-transition-list list))
      )
 ))
```

L'évaluation est effectuée lors de *call-next-method*. Si l'évaluation est un échec, toutes les transitions sont déclarées comme ayant été prises.

`state-fail-p (automaton state result)` `[fonction générique]`

Cette fonction rend T si, pour l'état *state* de *automaton*, l'action de *state* échoue. Dans le cas de ROBRA, on ne fait échouer la grammaire transformationnelle que si, en mode impératif, aucune règle ne s'est appliquée.

La spécialisation par défaut est :

```
(defmethod state-fail-p ((self automata-class) state result)
   (declare (ignore state result))
   (when (next-method-p) (call-next-method)) )
```

S'il n'y a pas de méthode suivante, la fonction rend NIL (l'état n'échoue pas).

Modifications du GAM

Pour implémenter les classes fusionnantes ci-dessus, nous avons dû modifier l'implémentation originale du GAM. Les modifications ont consisté à ajouter des méthodes spécialisables à des endroits où un comportement particulier était directement codé.

Par exemple, le choix de la transition à appliquer entraînait immédiatement l'évaluation de son action (avec des effets de bord sur les variables de l'automate). Nous avons retardé cette évaluation. Pour cela, nous avons rajouté un champ *next-transition* à la classe d'automates et modifié la méthode *automata-run* de la façon suivante :

```
(defmethod automata-run ((self automata-class))
   (setf (next-state self) (initial-state self))
   (do () (not (next-state self))
      (when (next-transition self) (transition-run (next-transition self)))
      (state-run (next-state self))
   (variable-list self))
```

Nous avons donc dû augmenter le protocole interne du GAM avec la méthode *transition-run* qui évalue l'action d'une transition. Cette modification va dans le sens d'une plus grande facilité de spécialisation.

8.3.2. Règles et grammaires

Comme l'exécution des règles se fait en parallèle, nous en avons laissé le contrôle au niveau de la grammaire. Le statut des règles dans ROBRA est donc inversé par rapport à ATEF où les grammaires sont passives (elles peuvent être considérées comme des dictionnaires de règles) et les règles actives (elles disposent d'une méthode pour évaluer leur corps).

Règles

La classe des règles utilisée dans ROBRA répond au protocole *object-data* sur les sélecteurs suivants :

- :name
- :schema
- :image
- :schema-geometric-descriptor
- :schema-algebric-descriptor
- :image-geometric-descriptor
- :image-algebric-descriptor

Le *schéma* est une liste contenant le *descripteur géométrique* et le *descripteur algébrique*. L'image contient deux descripteurs équivalents. La recherche d'une famille d'occurrences se fait sous contrôle de la grammaire à l'aide du langage de filtrage sur les arbres, PASTREE. Un sous ensemble des transformations sur les arbres prévues dans le modèle original se font à l'aide d'un langage inspiré de TTEDIT[78].

Grammaire

Certains des modes d'exécution de grammaire transformationnelles dépendent de la structure d'arbre. Il en sera de même pour l'implémentation des classes fusionnantes réalisant ces modes.

- *robra-grammar-class*

Cette classe est la classe de base d'une grammaire transformationnelle de ROBRA. C'est à elle que peuvent être rajoutées les classes fusionnantes. Cette classe doit implémenter tous les modes par défaut.

---

78 Il s'agit d'une réimplémentation non complètement finalisée.

Nous avons les modes par défaut suivants : facultatif, normal, unitaire, gardé, bas, coupe et dessous.

Le mode gardé demande qu'une table soit associée à l'AO. Cette table contient la liste des nœuds marqués. Les marques sont des listes de règles. Inversement, une règle de ROBRA contient une liste de nœuds. Comme les structures de données employées pour décrire les arbres peuvent varier (codage ou forme de liste, instances de classes), les informations contenues sur la table sont des descripteurs permettant de retrouver le nœud.

Un état du graphe de contrôle contient une référence sur une grammaire transformationnelle. Plusieurs états peuvent partager la même grammaire transformationnelle. Le champ *owner* d'une grammaire fait donc référence à l'automate (et non à un état). Si la grammaire est active, alors l'état courant de l'automate contient cette grammaire. Une grammaire, peut connaitre son état courant en utilisant le protocole *object-data* sur le sélecteur *current-state* :

:current-state retourne NIL si la grammaire n'est pas active (l'état courant ne contient pas cette grammaire).

- *imperative-execution-mode-ttg-mixin-class*

Ce mode d'exécution est indépendant de la structure de données sur laquelle travaille la grammaire transformationnelle.

Les règles de la liste retournée par *tg-select-applicable-rules* sont toujours appliquées. Donc si cette liste n'est pas vide, au moins une règle s'est appliquée. Il suffit dans ce cas de fixer le champ *touched-p* du code résultat à vrai et à faux sinon.

On peut pour cela donc spécialiser *tg-apply-rule* :

```
(defmethod tg-apply-rules :after ((self imperative-execution-mode-ttg-mixin-class)
                                  rule-list data data-kind &key &allow-other-keys)
 (unless rule-list (setf-object-data (result-code self) :touched-p nil))
 )
```

Il faut aussi surcharger *state-fail-p* :

```
(defmethod state-fail-p ((self imperative-execution-mode-ttg-mixin-class)
                         state result)
  (declare (ignore state))
  (not (get-object-data result :touched-p))
 )
```

L'évaluation de l'action de l'état a réussi si une règle au moins a été appliquée.

- *zero-execution-mode-ttg-mixin-class*

Ce mode d'exécution est indépendant de la structure de données sur laquelle travaille la grammaire transformationnelle.

Pour rendre le nœud portant I déterministe, il est possible de "purger" la pile de non-déterminisme après le passage sur un état. Il ne faut purger que localement à l'intérieur du sous-système transformationel courant. Pour cela, nous spécialisons *tg-execute* :

```
(defmethod tg-execute :after ((self zero-execution-mode-ttg-mixin-class)
                              data data-kind)
  (declare (ignore data data-kind))
  (stack-purge! (automaton-stack (owner self))
                :condition predicat   ; à vrai si le niveau de récursivité
                                      ; est le même que ceux en sommet de pile

 ))
```

- *exhaustive-execution-mode-ttg-mixin-class*

Ce mode d'exécution est indépendant de la structure de données sur laquelle travaille la grammaire transformationnelle.

Ici, nous modifions la méthode *tg-apply-rules*. En effet, par défaut, une seule application est faite. Les modifications sur *tg-apply-rules* consistent à introduire une boucle pour l'application des règles et à marquer les règles. Pour rendre ce mode indépendant de la structure d'arbre, les règles utilisées ne sont pas marquées directement sur l'arbre, mais dans une table avec comme information complémentaire l'objet (lui-même ou des informations permettant de le retrouver — un descripteur) sur lequel la règle s'applique. Cet objet peut-être un arbre mais aussi bien une chaîne de caractères ou une partie de graphe.

```
(defmethod tg-execute ((self transformational-grammar-class)
                       data data-kind &key &allow-other-keys)
  (let (end data)
   (do () (end)
    (let ((rule-node-list (tg-select-applicable-rules self data data-kind)))
       (setf new-data (tg-apply-rules self rule-node-list data data-kind))
       (setf end (tg-compute-execution-end self new-data data data-kind))
     ))
 ))
```

rule-node-list est une liste de paires règle/nœud. Les règles vont être appliquées sur les nœuds.

Par défaut (mode unitaire), la fin de l'exécution est immédiate après la première application.

```
(defmethod tg-compute-execution-end ((self transformational-grammar-class)
                                     original-data data data-kind &key &allow-other-keys)
  (declare (ignore original-data data data-kind))
  (if (next-method-p) (call-next-method)
     t))
```

En mode exhaustif, on répète l'application jusqu'à ce qu'il n'y ait plus de règle applicable.

```
(defmethod tg-compute-execution-end
                    ((self exhaustive-execution-mode-ttg-mixin-class)
                      original-data data data-kind &key &allow-other-keys)
  (declare (ignore original-data data data-kind))
  (tg-all-rules-marked-p self))
```

Le prédicat *tg-all-rules-marked-p* retourne vrai si toutes les règles sont marquées. La méthode *tg-compute-execution-end* spécialisée sur la classe fusionnante est celle qui retourne le résultat à la méthode appelante (au vu de l'ordre topologique des classes).

- *free-execution-mode-ttg-mixin-class*

Ce mode d'exécution est dépendant des arbres comme structures de données sur lesquelles travaille la grammaire transformationnelle.

Nous pouvons spécialiser *tg-rule-applicable-p* de façon à ce que cette méthode ne tienne pas compte des marques. *tg-rule-applicable-p* est appelée par *tg-select-applicable-rules* pour chaque règle de la grammaire sur tous les racines de transformation potentielles. Par défaut nous avons, pour *tg-select-applicable-rules* :

```
(defmethod tg-select-applicable-rules (grammar data data-kind &key
                                       &allow-other-keys)
   (let ((scan-mode (tg-select-scan-mode grammar data data-kind))
         (rule-list (get-object-data grammar :rules))
         rule-node-list)
      (scan-data (item data data-kind scan-mode)
         (scan-data (rule rule-list :list)
            (if  (rule-applicable-p grammar rule node)
                 (push (cons rule node) rule-node-list))))
      (setf rule-list (handled-marked-rules grammar data data-kind rule-node-list))
      (setf rule-list (resolve-conflicts grammar data data-kind rule-node-list))
 ))
```

scan-data est une fonction de parcours générique semblable à la fonction prédéfinie *dolist*. Le parcours est effectué sur *data* avec un indicateur *data-kind*. Chaque élément du parcours est localement lié à *item*. Enfin, un mode de parcours peut être spécifié.

Par exemple, si *data* est une chaîne, chaque caracatère de la chaîne sera énuméré selon deux parcours possibles, gauche-droite et droite-gauche. Le mode de parcours est optionnel, un mode par défaut étant toujours spécifié.

Pour chaque nœud, on cherche toutes les règles pouvant s'appliquer. Ensuite *resolve-conflicts* permet de ne garder qu'un sous-ensemble des occurrences chaque de règle. On peut vouloir soit une occurrence de chaque règle soit plusieurs occurences de la même règle en parallèle.

handle-marked-rules supprime de la liste des règles, les règles marquées.

Pour le mode libre, nous devons surcharger *handle-marked-rules* de façon à ne pas tenir compte du marquage

```
(defmethod handle-marked-rules :around (grammar data data-kind rule-list)
   (declare (ignore grammar data data-kind rule-list))
   )
```

La méthode suivante n'est pas appelée (pas de *call-next-method*), on ne supprime donc aucune règle.

- *top-execution-mode-ttg-mixin-class*

Ce mode d'exécution est dépendant des arbres comme structures de données sur lesquelles travaille la grammaire transformationnelle.

Le mode de parcours des arbres peut être spécialisé pour cette classes fusionnates :

```
(defmethod tg-select-scan-mode ((self top-execution-mode-ttg-mixin-class)
                                data data-kind)
   (declare (ignore data data-kind))
   :top)
```

Par défaut le mode de parcours est bas.

```
(defmethod tg-select-scan-mode (grammar data data-kind)
  (declare (ignore data data-kind))
  (if  (next-method-p) (call-next-method)
       :bottom) )
```

Si la méthode suivante existe, la valeur rendue sera celle rendue par cette fonction, sinon c'est la valeur *:bottom.*

Nous pouvons expliciter le schéma (dit "d'ordre inverse") que nous employons depuis quelques spécialisations :

```
(if  (next-method-p) (call-next-method)
     action)
```

pour privilégier la méthode la plus éloignée dans l'ordre topologique[79].

- *total-execution-mode-ttg-mixin-class*

Ce mode d'exécution est dépendant des arbres comme structures de données sur lesquelles travaille la grammaire transformationnelle.

La résolution de conflits tient compte, entre autres, des dominances entre les nœuds. Il est nécessaire de surcharger *resolve-conflicts* pour cela.

- *single-execution-mode-ttg-mixin-class*

Ce mode d'exécution est dépendant des arbres comme structures de données sur laquelle travaille la grammaire transformationnelle.

tg-apply-rule effectue en parallèle l'application des règles. Pour l'implémentation, les occurrences de règles sont en pratique évaluées dans une ordre quelconque. Une fonction *tg-mark-rule* est appelée par *tg-apply-rule*, une fois l'occurrence de règle appliquée.

```
tg-mark-rule (grammar rule node data data-kind) [fonction générique]
```

La définition par défaut de *tg-mark-rule* est un schéma inverse :

```
(defmethod tg-mark-rule (grammar rule node data data-kind)
  (if  (next-method-p) (call-next-method)
       …
       ;; met la marque dans la table pour le nœud node et tous les
       ;; nœuds dessous
 ))
```

La spécialisation de *tg-mark-rule* est la suivante :

---

79 Nous avons défini d'autres schémas :
- Schéma de "surcharge" : *action*

La première méthode trouvée rend son résultat. On n'appelle donc jamais les méthodes suivantes.
- Schéma "dedans" :
```
(let (next-result)
 action1
 (setf next-result (if (next-method-p) (call-next-method)))
 action2)
```

Le méthode suivante, dont on peut exploiter le résultat, est appelée dans la méthode.

- Schéma "Autour" avec utilisation de *:after* pour les méthodes des classes fusionnantes. C'est l'inverse du précédent.

```
(defmethod tg-mark-rule ((grammar single-execution-mode-ttg-mixin-class)
                         rule node data data-kind)
  (if  (next-method-p) (call-next-method)
          …
          ;; met la marque dans la table seulement pour le nœud node
 ))
```

L'implémentation de ROBRA a jusqu'ici principalement concerné ce qui été présenté ici. À savoir, la modification du GAM pour le graphe de contrôle ainsi que la classe de grammaire transformationnelle et ses modes d'exécution. Doivent être terminés ou étendus :

- le(s) langage(s) de description de schémas d'arbres ;
- le(s) langage(s) de transformations d'arbres (inclusion des dépendants généralisées et des contextes).

Chapitre 9

Vers des protocoles extensibles

Dans la réingéniérie des LSPL ATEF et ROBRA, nous avons introduit l'extensibilité sour la forme de spécialisation de fonctions génériques et de composition de classes fusionnantes. Ces deux techniques permettent des extensions de nature différentes.

L'extensibilité peut concerner non seulement les noyaux des outils, mais aussi l'interface utilisateur ou les syntaxes externes. Avec un outillage adéquat, on peut fournir des possibilités d'extention au développeur linguiste. Ces extensions étaient auparavant réservées au développeur informaticien.

Tout comme la généricité, l'extensibilité bute sur des difficultés. Une fois encore, l'outillage constitue une bonne solution à certains problèmes, mais au prix d'un accroissement de la complexité du protocole.

9.1. Leçons de deux premières implémentations

Les protocoles peuvent être conçus de façon à être facilement étendus à l'aide de nouvelles spécialisations sur des fonctions génériques. Plus un protocole sera stratifié, plus une nouvelle extension sera facile à réaliser. Par contre, le protocole est plus difficile à concevoir.

Les classes fusionnantes permettent de composer dynamiquement de nouvelles classes à partir de "briques élémentaires" implémentant chacune un comportement particulier. On peut ainsi éviter de développer une hiérarchie de classes traduisant la combinatoire de tous les comportements identifiés.

9.1.1. Extensions par fonctions génériques

Les fonctions génériques peuvent être spécialisées de deux manières.

- sur la classe de leur argument

Cette technique est utilisée avec la dérivation de classes et dans la mesure où un objet dispose en paramètres des classes d'objets dont il a la charge.

- sur une valeur de leur argument

Comme valeurs, nous utilisons généralement des symboles. De nouveaux comportements peuvent ainsi librement être créés avec des symboles inutilisés.

Production des formes de sortie dans ATEF

Ce protocole (voir 7.2.2.) est particulièrement simple, car la création d'une nouvelle forme de sortie ne nécessite la spécialisation que d'une seule méthode (*output-form-name*) sur l'argument *output-form-name*.

Cependant, le travail pour produire une forme de sortie à partir de la forme interne peut être important. Le protocole n'a pas été stratifié, car nous n'avons pas vu comment un code de base pourrait être factorisé.

Distribution de règles dans ATEF

Le protocole de distribution des règles (*rule-dispatcher*) d'ATEF est plus complexe. Quatre handlers doivent être spécialisés et les états inital et final de chaque méthode ne sont pas quelconques.

En général, la spécialisation se limite à peu de lignes de code (pour le mode *:or* seule *dispatch-handler-rule-completed* fait plus de cinq lignes). Parfois, la spécialisation d'un handler se limite à écrire une fonction qui ne fait rien (cas de *dispatch-hanlder-subrule-completed* spécialisé sur *:or*).

Dans la conception de ce protocole, nous avons trouvé difficile de déterminer de façon générale quelle est la part des mécanismes qui aurait avantage à être gérés par le système et quels sont ceux à laisser à la charge du développeur. Par exemple, dans *rule-dispatcher*, les sous-règles doivent être mises sur la pile dans la spécialisation. C'est-à-dire que l'on a délibérement choisi de ne pas le faire automatiquement.

Les modes de distribution peuvent être changés dynamiquement (à l'exécution). Cela peut être particulièrement utile à un développeur linguiste lors de la mise au point d'une application linguistique. Par exemple, le développeur linguiste peut souhaiter disposer d'un mode où les sous-règles ne sont pas exécutées. Il s'agit de "mettre provisoirement entre commentaires" les sous-règles. Ce mécanisme est plus simple que celui consistant à redéfinir provisoirement la même règle en ôtant ses sous-règles[80].

La spécialisation des handlers est immédiate :

---

80 De plus, on peut fournir au développeur linguiste des fonctions qui permettent de changer dans les règles les modes de distribution d'autres règles. Nous pensons que de telles fontionnalités sont nécessaires lors de la phase de développement linguistique, mais qu'elles ne doivent pas être utilisées une fois l'application mise au point. Il s'agit selon nous d'un outillage.

```
(defmethod dispatch-hanlder-body-success ((self rule-dispatcher)
                                          item (mode (eql :null)))
  (declare (ignore item))
 )
```

Si le corps de la règle réussit (c'est-à-dire que *(get-object-data item 'body-result)* vaut T), on n'empile pas les sous-règles. On n'a donc rien à faire.

```
(defmethod dispatch-hanlder-body-failure ((self rule-dispatcher)
                                          item (mode (eql :null)))
  (declare (ignore item))
  (engine-restore-state (owner self)
                        (get-object-data (s-top (rule-stack self)) 'state))
 )
```

Si la règle échoue (c'est-à-dire que *(get-object-data item 'body-result)* vaut NIL), on restaure l'ancien état courant.

```
(defmethod dispatch-handler-rule-completed ((self rule-dispatcher)
                                            item (mode (eql :null)))
  (declare (ignore mode))
  (cond
   ((= (get-object-data item 'level) 0)
      (set-object-data item 'rule-result t)
      (engine-gather-result (owner self) :success t)
    )
   ((> (get-object-data item 'level) 0)
    (set-object-data item 'rule-result t)
    )
   (t (error "level ~s is not correct." (get-object-data item 'level))))
  ;; don't forget to call the next-method (restore the state)
  (when (next-method-p) (call-next-method))
  )
```

Ici, on ne tient pas compte du résultat des éventuelles sous-règles (elle n'ont jamais été appelées).

```
(defmethod dispatch-handler-subrule-completed ((self rule-dispatcher)
                                               item (mode (eql :null)))
  (declare (ignore item)))
```

Cette spécialisation est inutile, car elle ne sera pas appelée (on peut en faire un cas d'erreur).

Nous constatons qu'un certains nombre de mécanismes auraient pu être gérés automatiquement. Par exemple, l'exploitation du résultat de l'évaluation d'une règle de niveau 0 et peut-être la gestion de l'échec du corps d'une règle. Si tel avait été le cas, le protocole et les spécialisations auraient été plus simples. Cependant, l'ensemble du protocole aurait été moins puissant. Il aurait été par exemple impossible de réaliser un mode inverse qui fait réussir une règle qui échoue :

```
(defmethod dispatch-hanlder-body-success ((self rule-dispatcher)
                                          item (mode (eql :inverse)))
  (declare (ignore item))
  (engine-restore-state (owner self)
                        (get-object-data (s-top (rule-stack self)) 'state))
 )
```

```
(defmethod dispatch-hanlder-body-failure ((self rule-dispatcher)
                                          item (mode (eql :inverse)))
   (declare (ignore item))
   (if   (atef-rule-has-subrules? (get-object-data item 'rule))
         (push-subrules self))
 )
…
```

Nous avons donc volontairement défini un protocole très souple qui se révèle puissant, mais qui favorise peu la factorisation du code et avec lequel il est facile de "faire n'importe quoi". Il est probable que, dans certains cas, moins de puissance et plus de sécurité seraient désirables.

LT et les classes paramétrées

Un programme noyau LT est compilé selon les primitives du GAM (voir §3.1.2.). Cependant, nous avons défini une classe d'automates, *lt-transcriptor-class*, dérivée de la classes de base, *automata-class.*

La macro *defautomata* dispose d'un mot-clé *class* qui permet de préciser la classe de l'automate à instancier. La méthode de CLOS *initialize-instance* appelle elle-même *choose-input-tape-class* et *choose-output-tape-class* pour déterminer les classes des bandes d'entrée et de sortie.

Définir un outil dérivé de LT peut donc se faire en définissant une nouvelle classe d'automates (peut être dérivée de *lt-transcriptor-class*) et en fournissant le nom de cette classe au mot-clé *class.*

9.1.2. Extensions par composition de classes

L'intérêt des extensions par composition de classes fusionnantes est qu'elles permettent au développeur linguiste de composer lui-même (de manière cachée) ses classes, en sélectionnant des options. Dans [Habert 1991], cette approche a été adoptée pour la spécialisation de règles syntaxique de l'analyseur OLMES.

Extension du GAM dans ROBRA

Nous avons décidé d'étendre le GAM au non-déterminisme et à la récursivité en essayant de définir les classes fusionnantes qui surchargent les méthodes de base afin d'obtenir ces comportements. Nous n'avons pas précisé comment le non-déterminisme n-aire pouvait être implémenté.

Avec le non-déterminisme unaire, seule l'arrivée dans l'état d'erreur active le retour arrière. Avec le non-déterminisme n-aire, l'arrivée dans un état final doit aussi activer le retour arrière avec conservation du résultat. La classe fusionnante :

`n-nd-automaton-mixin-class` `[classe fusionnante]`

doit avoir le même comportement que *1-nd-automaton-mixin-class* mais relancer l'automate à l'état final après avoir stocké le résultat dans une liste.

```
(defclass n-nd-automaton-mixin-class (automaton-manager-class)
   ((automaton-stack   :initform nil
                       :accessor automaton-stack)
    (automaton-result-list   :initform nil
                             :accesor automaton-result-list)
 )
```

Nous en profitons pour rendre spécialisable la classe de pile automate (automaton-stack) et également pour rendre plus générique ce type de spécialisation avec la fonction générique :

```
choose-object-slot-class (object slot)          [fonction générique]
```

Pour une instance *object*, cette méthode retourne le symbole de la classe qui doit être instanciée pour *slot*[81].

Il est de plus nécessaire de définir une classe dérivée pour l'état final et de surcharger la méthode sélectionnant la classe d'état final.

Mode d'exécution dans ROBRA

Après analyse des différents modes d'exécution des grammaires transformationnelles de ROBRA, nous avons conçu une architecture générale permettant d'implémenter les modes à l'aide de classes fusionnantes.

L'idée était de pouvoir spécialiser avec un "schéma inverse" une méthode afin d'implémenter le mode. Cela a de fortes implications :

- deux classes fusionnantes ne peuvent pas spécialiser la même méthode ;
- Les modes *:before, :after* ou *:around* nous permettent dans une certaine mesure de contourner la limitation ci-dessus, mais au risque de rendre incompatibles certains classes fusionnantes ;
- Il est possible de définir des modes non binaires.

  Par exemple, le mode unitaire fait place à une option application-number pouvant avoir les valeurs *unitaire*, *exhaustif*, ou un entier *n*.

Pour chaque méthode spécialisable, on peut définir autant de classes fusionnantes la spécialisant. L'extension de ROBRA à de nouvelles valeurs de mode ne pose donc pas de problème. Par contre, la définition d'un nouveau mode peut présenter quelques difficultés. Il faudra en effet déterminer quelle méthode est spécialisable. Si une telle méthode n'est pas trouvée, il faudra modifier le code existant pour la créer.

Ce processus peut être coûteux.

---

81 Ce protocole rend obsoletes les méthodes *choose-input-tape-class* et *choose-output-tape-class*. Il suffit de les remplacer respectivement par :

```
(defmethod select-object-slot-class (  (self LT-transcriptor-class) (slot (eql 'input-tape))
    'double-headed-tape)
(defmethod select-object-slot-class (  (self LT-transcriptor-class) (slot (eql 'output-tape))
    'single-headed-tape)
```

Il faut aussi mettre à jour *initialise-instance* :

```
(defmethod initialize-instance ((self LT-transcriptor-class) &key)
    (call-next-method)
    (setf (input-tape self) (select-object-slot-class self 'input-tape))
    (setf (output-tape self) (select-object-slot-class self 'output-tape)) )
```

9.2. Vers quelle extensibilité ?

Nous pensons que l'extensibilité doit faciliter le travail de prototypage du développeur. L'idée d'un développement de système évolutif[82] de [Budde, Kautz, Kuhlenkamp & al. 1991] mérite selon nous d'être appliquée au TALN.

Pour définir des outils évolutifs, nous avons choisi de nous baser sur la notion de protocole. Dans [Cutting, Pedersen & Halvorsen 1991], l'accent est également mis sur la programmation par objets et sur les protocoles[83] afin de permettre de construire rapidement de nouveaux outils.

Pour cela, nous distinguons plusieurs facettes à l'extensibilité qui, ensemble, permettent d'aboutir à des familles d'outils.

9.2.1. Plusieurs niveaux d'extensibilité

Nous distinguons plusieurs niveaux d'extensibilité : celle pour le développeur et pour les noyaux (sur lesquelles nous avons surtout insisté jusqu'à maintenant) et celle pour l'utilisateur (le développeur linguiste, dans le cas particulier qui nous intéresse) et les environnements.

Extension du noyau par le développeur

L'ensemble des protocoles que nous avons définis s'adresse au développeur informaticien. Une première approche du développement de protocoles en CLOS peut être trouvée dans [Keene 1989]. Les travaux sur le MOP [Kiczales & al. 1991] se sont concentrés, par exemple, sur la définition de nouvelles combinaisons de méthodes.

Pour être facilement utilisable par le développeur, un protocole doit être stratifié et outillé :

- la stratification permet de réduire le travail nécessaire à une nouvelle spécialisation.

La stratification fait partie des techniques réflexives. Ces techniques permettent d'ouvrir l'implémentation d'un langage (ou d'un outil), sans en révéler des détails inutiles, et sans compromettre la portabilité ([Kiczales & al. 1991]). Comme introduire de la stratification simplifie la spécialisation mais rend l'ensemble du système plus complexe, il est important de déterminer où il est utile de le faire.

- l'outillage consiste à développer des fonctions (voire des protocoles) annexes permettant une bonne gestion d'un protocole.

Nous n'avons qu'évoqué cet aspect avec la gestion des formes d'entrée et de sortie acceptables pour ATEFlisp. Nous revenons, de façon générale, sur cette question de la gestion des protcoles au §9.3.1.

Extension par l'utilisateur

L'utilisateur ne peut (et ne veut) pas définir lui-même de nouvelles extensions en CLOS. Il est, par contre, possible de lui fournir un menu ou un formulaire, dans lequel on lui propose de choisir entre différentes options.

---

82 "Evolutionary System Development" (ESD)

83 En CLOS également.

Un mécanisme d'extension activable/désactivable par l'utilisateur peut être défini[84].

Extension de l'environnement

Nous n'avons que très peu abordé le problème de la conception de l'environnement de développement pour linguiste. Nous avons toutefois présenté l'éditeur multi-dialecte (voir §3.3.3). De nouvelles syntaxes peuvent être facilement développées et intégrées dans l'éditeur.

L'intégration peut être faite par l'utilisateur par un mécanisme similaire à celui présenté ci-dessus. Un fichier de "nouvelle syntaxe" contient la définition d'une syntaxe externe, c'est-à-dire une réalisation particulière des grammaires d'"analyse" et de "génération".

9.2.2. Familles d'outils

Du point de vue du développeur, il est intéressant d'augmenter la réutilisabilité en étant capable, à partir d'un outil suffisamment extensible, d'en dériver toute une famille. Nous avons constaté deux choses : non seulement les mécanismes d'extension se basent sur des mécanismes génériques, mais, réciproquement, l'extensibilité rend générique.

L' idée de famille d'outils a été également développée dans [Hüe & Jayez 1991] où est présenté un atelier de génie logiciel composé de trois générateurs d'analyseurs syntaxico-sémantiques qui permettent d'utiliser des grammaires à contraintes lexicales, des grammaires contextuelles ou des grammaires lexicales inverses.

Familles de moteurs

Nous pouvons catégoriser l'extensibilité apportée aux moteurs d'ATEF et de ROBRA selon trois niveaux : la *paramétrisation* , le *réglage* et la *dérivation*. Nous pouvons à chaque fois, utiliser une analogie avec un moteur de voiture (voir [Lafourcade 1994d])

- Paramétrisation

La paramétrisation est une modification du comportement d'un outil pouvant être effectuée pendant son fonctionnement. Le starter d'une voiture peut être activé par le conducteur pour modifier la carburation.

Dans ATEFlisp le sens de l'analyse peut être changé durant l'analyse. Le contrôle se fait au niveau des règles. Cela peut être utile dans le cas où l'analyse par défaut a échoué et où un processus de *mot inconnu* essaye de tirer le plus d'informations possible de la forme à analyser.

- Réglage

Le réglage est une modification ne pouvant pas être effectuée pendant le fonctionnement de l'outil. Le réglage d'un moteur doit être effectué à l'arrêt.

Certaines options globales de fonctionnement pour le graphe de contrôle de ROBRA comme le déterminisme n-aire doivent être fixées avant l'exécution.

---

84 De la même manière que sous Macintosh des "Extensions" peuvent être "lachées" dans le "Dossier Système" et être activées au redémarrage. Ces extensions ajoute de nouvelles fonctionnalités au système d'exploitation.

- Dérivation

La dérivation est une modification en profondeur de un ou de plusieurs composants d'un outil. Elle peut être est effectué par changement de certaines "pièces". La modification du moteur d'une voiture peut consister à rajouter un compresseur ou à changer le nombre de cylindres.

Pour ROBRA, la dérivation vers un système transformationnel sur les chaînes de caractères nécessite les changements suivants : remplacement du langage de filtrage sur les arbres par un équivalent sur les chaînes, remplacement du langage de manipulation d'arbres par un équivalent sur les chaînes, suppression des modes d'exécution dépendant de la structure d'arbre et définition (éventuelle) de nouveaux modes dépendant de la structure de chaînes.

Familles de syntaxes externes

Dans quelle mesure les syntaxes externes (qui cachent la syntaxe noyau) proposées aux développeurs linguistes peuvent-elles être, elles aussi, intégrables, génériques et extensibles ?

L'intégration d'une nouvelle syntaxe ne pose pas de problème. Par contre, la question de la généricité et de l'extensibilité reste, ici, largement ouverte. Cependant, nous pensons que l'extensibilité peut porter sur les points suivants :

- Qualité du traitement des erreurs.

  Ajouter des traitements ou des messages spécifiques en cas de détection statique d'une erreur.

- Extensions de syntaxes à géométrie variable

  Ajouter des équivalents pour des mots-clés. Par exemple, pour LT, **`trans`** pourrait être un nouvelle équivalent de **`transcripteur`** en syntaxe française.

  Rendre certains mots-clés optionnels. Par exemple, **`le transcripteur`** pourrait être équivalent à **`transcripteur`**.

9.3. Difficultés et limites de l'extensibilité

Nous pensons qu'il est intéressant d'essayer de fixer les limites de l'extensibilité au-delà desquelles on complique trop une implémentation. Celle-ci peut devenir inefficace, peu compréhensible et donc difficilement maintenable et extensible

Nous avons montré qu'un outillage s'imposait souvent pour rendre un protocole utilisable. Cela peut aussi se payer par des développements importants.

9.3.1. Gestion des protocoles

Une des difficultés que posent les protocoles extensibles est de connaître les extensions qui ont été faites. C'est particulièrement vrai si les extensions n'ont pas été effectuées par les développeurs qui ont conçu le protocole.

Avec le protocole sur les formes d'entrée d'ATEF, nous avons défini une variable **atef-tool-acceptable-input-form** qui contient la liste par défaut des formes d'entrée acceptables.

Devons-nous systématiser une approche consistant à enregistrer chaque nouvelle spécialisation ? Ou encore est-il nécessaire d'implémenter l'idée de protocole sous forme d'objets CLOS et de définir un meta-protocole (pas nécessairement extensible) permettant la gestion des protocoles ?

Nous pensons que la conception d'un mécanisme général de gestion des protocoles pose plusieurs problèmes :

- Difficulté à concevoir et implémenter

  La spécialisation d'une méthode peut être faite sur tous ses arguments. La combinatoire des spécialisations peut ainsi devenir importante, et il n'est pas facile de les gérer.

  De plus, l'implémentation doit faire appel au MOP qui est encore incomplétement finalisé.

- Complication de l'idée de protocoles

  En effet, la gestion par défaut pourra ne pas convenir à tous. Nous nous dirigeons alors vers la pente, dangereuse selon nous, menant à des meta-protocoles génériques et extensibles permettant d'instancier des protocoles particuliers.

Nous pensons que l'accent doit plutôt être mis sur une documentation efficace et complète, où la stratification doit être mise en évidence.

9.3.2. Gestion de la fusion de classes

Jusqu'ici, nous avons délibérément passé sous silence la gestion délicate des classes fusionnantes.

Problèmes

Par exemple, si nous voulons créer facilement un objet dont les propriétés vont induire une combinatoire de classes fusionnantes, nous pouvons utiliser la fonction générique *make-object* avec un mot-clé *properties* :

Ce mot clé contient une liste de paires propriétés/valeur. Pour chaque propriété est déterminée la classe fusionnante appropriée (ou NIL s'il n'en existe aucune).

```
select-mixin-classes (base-class property-list) [fonction générique]
```

La définition de *select-mixin-classes* est :

```
(defmethod select-mixin-classes (base-class property-list)
  (let (result mixin-class)
    (dolist (pair property-list (reverse result))
      (setf   mixin-class
              (select-mixin-class base-class (first pair) (second pair)))
      (push mixin-class result)
     )
    result) )
```

La création effective de l'instance doit d'abord être précédée de la création de la classe, nous avons donc dans *make-object*, le code suivant :

```
(let* ( (class (make-class-with-mixin
                  base-class
                  (select-mixin-classes base-class properties)))
        (instance (make-instance class))
      )
  … )
```

make-instance est polymorphique et accepte d'avoir comme paramètre l'objet correspondant à la classe (rendu par *make-class-with-mixin*).

Le problème avec ce code est que la même classe est dynamiquement créée pour chaque nouvelle instance. Si deux instances sont créées, deux classes strictement équivalentes sont créées. Non seulement ce processus est coûteux en mémoire, mais les deux instances ne sont pas de la même classe, alors qu'elles ont été créées avec la même classe de base et les mêmes propriétés. Nous nous attendons donc à ce qu'elles soient de la même classe.

Stockage des classes fusionnantes

Le problème présenté ci-dessus peut être évité par un stockage, dans une table, des classes fusionnantes. En effet, il suffit dans la méthode *make-class-with-mixin* d'insérer la classe créée dans une table de hachage. L'accès se fait par la liste composée de la classe de base et des classes fusionnantes.

`*dynamicaly-defined-class-repository*` `[variable]`

Cette variable contient la table de hachage. Le test de comparaison est l'égalité (au sens *equal*).

Nous devons donc modifier *make-class-with-mixin* en reportant son travail sur *ensure-mixin-class*:

```
(defun make-class-with-mixin (base-class mixin-list)
  (ensure-mixin-class (cons base-class mixin-list)))
```

La méthode *ensure-mixin-class* cherche dans la table si une classe appropriée existe et la retourne si tel est le cas. Sinon, la classe est créée puis insérée dans la table.

La définition de *ensure-mixin-class* est la suivante:

```
(defmethod ensure-mixin-class (class-list)
  (let* (  (class (gethash class-list *dynamicaly-defined-class-repository*)) )
    (if (not class)
     (eval
       `(let (class)
            (setf class (defclass ,name (,base-class ,@mixin-list)))
            (setf (gethash   (cons base-class mixin-list)
                             *dynamicaly-defined-class-repository*)
                  class)
          ))
     class)) )
```

La fonction *(setf gethash)* est la fonction standard pour écrire dans une table de hachage.

L'ensemble de ce protocole fonctionne dans la limite où les propriétés sont triées (par ordre lexicographique). Ce tri permet de s'assurer que deux classes ayant les mêmes proriétés disposeront d'une liste de classes fusionnantes triée dans le même ordre. Dans le cas contraire, *ensure-mixin-class* pourrait échouer alors que la classe correspondant à la même composition (dans un ordre différent) se trouve bien dans la table.

Conclusion de la troisième partie

Les LSPL peuvent être rendus extensibles à l'aide de protocoles. Ces protocoles permettent de combiner de nouveaux types d'objets manipulés par les primitives du langage, et aussi de définir de nouveaux types de contrôle. Le partage de code amène à la définition de sous-protocoles très fins qu'il est aisé de modifier. Cependant, cela se paye par une augmentation de l'abstraction des primitives de base et un accroissement de la complexité du système.

La définition de classes fusionnantes permet, pour une classe d'objets, de modulariser des comportements et de les composer dynamiquement. Seuls les langages à objets offrant l'héritage multiple et le changement dynamique de classes permettent une telle approche.

Cependant, la gestion d'un ensemble cohérent de classes fusionnantes est particulièrement délicate. La mise au point est difficile car le contrôle sur la distribution des méthodes n'est pas explicitement géré par le développeur. Un tel contrôle explicite peut s'avérer utile, mais peut aussi être la conséquence d'une analyse à objets non pertinente.

Conclusion

Nous avons identifié l'intégrabilité, la généricité et l'extensibilité comme trois propriétés essentielles à la définition et à l'implémentation des LSPL.

Le modèle à objets LEAF que nous avons défini permet de décrire les composants qui interviennent dans la plupart des systèmes de TALN. LEAF définit trois grandes classes : les treillis servent à représenter des informations géométriques, les décorations sont liées aux informations algébriques, et les moteurs modifient les treillis et les décorations. Nous nous sommes intéressé aux LSPL qui servent à décrire les décorations et les moteurs.

Après une première expérience concrète sur *l'intégrabilité,* faite sur l'outil ODILE, nous avons conclu qu'il est nécessaire d'élargir le concept d'interface logicielle à celui de protocole. Pour des systèmes de grande taille et constitués de nombreux composants, l'architecture de "tableau blanc" nous semble satisfaisante. Cette architecture favorise une approche modulaire et incrémentale, et permet aussi de fournir au développeur linguiste une visualisation centralisée.

La généricité passe aussi par la définition de protocoles. Nous avons réalisé une première expérience avec la réingénierie du LSPL LT. Ce Langage de Transcription est basé sur un gestionnaire d'automates, GAM, que nous avons implémenté. La syntaxe "noyau" de LT est traduite dans les termes du langage constitué par le GAM dont certaines des classes ont été spécialisées. Comme il est difficile de choisir une syntaxe externe satisfaisante pour tous les types d'utilisateurs, nous avons défini un éditeur "multidialectal". Cette expérience a été

l'occasion de définir une bonne modularité entre le noyau et l'interface utilisateur.

Une seconde expérience sur la généricité a été menée avec le langage noyau DÉCOR. Ce langage, basé sur des "décorations", a été défini de façon à pouvoir servir de base d'implémentation à de nombreux types de représentation utilisés dans le TALN. Nous avons pu constater que la généricité passe souvent par des protocoles qui peuvent eux-mêmes être étendus. Les protocoles nous semblent incontournables pour la réalisation d'une collection d'outils formant un ensemble cohérent.

L'extensibilité a été l'objet de deux expérimentations avec les moteurs des LSPL ATEF et ROBRA. Une fois encore, nous avons opté pour la définition de protocoles pour rendre extensibles le contrôle et les différentes fonctionnalités de ces moteurs. Nous avons alors constaté que cette extensibilité nécessite la réalisation de mécanismes de plus en plus généraux et éloignés de la conception originale. Ces mécanismes généraux (souvent abstraits au sens de la programmation à objets) sont ensuite spécialisés et instanciés.

Par exemple, le protocole basé sur les "piles à filtres dynamiques" utilisé dans ATEF permet une bonne généralisation des fonctions de contrôle du non déterminisme. Cette généralisation permet de définir de nouvelles fonctions plus facilement qu'auparavant. Cependant, ce mécanisme est relativement abstrait et rend finalement plus difficile la compréhension générale du système.

La définition de protocoles à l'aide de classes fusionnantes, expérimentée dans ROBRA avec l'extension du GAM, permet d'isoler certains comportements et d'en faire des briques de base réutilisables et composables. Cependant, en pratique, une telle approche demande un outillage important. De plus, cette augmentation de la modularité bute, elle aussi, sur une "barrière de compréhensibilité".

Ces problèmes de compréhensibilité du code nous ont incité à ne pas aller trop loin dans la définition de protocoles génériques et extensibles. Dans les expérimentations que nous avons entreprises, nous avons pu déterminer certaines limites à ne pas dépasser. Dans le futur, nous espérons trouver des règles générales permettant de déterminer où se trouve cette limite.

Certains de nos outils, protocoles et modules, ont été rendus accessibles par ftp[85] (cambridge.apple.com). Les réactions d'autres développeurs sur la nature de ces outils, aussi bien que sur leur conception, ont été tout à fait enrichissantes. Elles nous ont permis d'améliorer notre code, et de vérifier qu'il était bien utilisable et compréhensible par d'autres que nous. Les échanges avec les linguistes ont également été très importants. En effet, les LSPL ne peuvent plus être considérés comme des prototypes de laboratoire, dès lors qu'ils doivent être utilisés par des linguistes.

Tous nos outils ont été développés à l'aide de langages à objets. ODILE a été implémenté en Object Pascal, et les autres outils et protocoles en CLOS avec Macintosh Common Lisp (MCL).

On peut se demander si le choix de CLOS est pertinent. L'avenir qu'Apple réserve à MCL semble incertain. Cependant, la portabilité de Common Lisp/CLOS avait été un facteur déterminant dans le choix de MCL/CLOS, et, sauf peut-être en ce

85 File Transfert Protocol

qui concerne les éléments d'interface utilisateurs (soigneusement séparés des "noyaux"), l'adaptation de nos travaux à d'autres Common Lisp/CLOS, sur d'autres plates-formes, ne devrait pas rencontrer de grande difficulté.

D'autre part, il ne semble pas y avoir de choix qui s'impose vraiment. Ainsi, les outils du projet MU (1982-86) ont été réalisés en ZetaLisp, abandonné depuis, et ne sont plus exploitables aujourd'hui. Tout a dû être réécrit en C pour le système Majestic du JICST. Plus récemment, le projet Eureka EUROLANG a choisi C++ comme langage de développement. Ce standard offre des performances d'exécution supérieures à CLOS, mais reste de bien plus bas niveau en termes de développement. La pérennité du choix du langage de développement dépend de facteurs peu ou très peu maîtrisables par les laboratoires de recherche (et même par beaucoup d'industriels). Utiliser l'approche du "tableau blanc" permettrait d'augmenter l'espérance de vie des outils.

Les mêmes questions peuvent être posées sur le choix du matériel. Le Macintosh constitue-t-il la base idéale de développement ? Sa facilité d'utilisation en fait une bonne alternative aux stations de travail Unix pour les développeurs linguistes (qui, rappelons-le, ne sont pas des informaticiens). Cependant, l'usage du Macintosh pour le développement reste limité dans le domaine de la recherche.

Dans quelle direction aller maintenant ? Nous pensons qu'il serait intéressant de poursuivre ces travaux dans trois directions complémentaires. Il nous semble d'abord nécessaire de valider nos méthodes sur un plus grand nombre de moteurs. Ensuite, un outil d'interface comme l'éditeur multidialectal nous permettrait de valider plus facilement de nouvelles syntaxes de LSPL. Enfin, il nous semble nécessaire d'étendre les techniques de génie logiciel aux environnements de développement de linguiciels.

Au total, cette étude, fondée aussi bien sur des considérations théoriques que sur des implémentations complètes, permet de cerner plus précisément les possibilités et les limites de la quête de l'intégrabilité, de la généricité, et de l'extensibilité dans le "génie logiciel pour le génie linguiciel".

Bibliographie

[Abeillé 1993] **Abeillé A. (1993)** *Les nouvelles syntaxes - Grammaires d'unification et analyse du français.* Armand Colin, Paris, 1 vol., 327 p.

[Aho & al. 1977] **Aho A. V. & Ullman J. D. (1977)** *Principles of Compiler Design.* Addison-Wesley series in Computer Science and Information Processing, Addison-Wesley, 604 p.

[Ahrenberg & al. 1991] **Ahrenberg L., Jönsson A. & Dahlbäck N. (1991)** *Discourse Representation and Discourse Management for a Natural Language Dialogue System.* LiTH-IDA-R-91-21, NLPLAB, August 1991, 14 p.

[Aït-Kaci & al. 1989] **Aït-Kaci H. & Lincoln P. (1989)** *LIFE : a Natural Language for Natural Language.* T.A. information, **30**/1-2: pp. 37-67.

[Aït-Kaci & al. 1992] **Aït-Kaci H., Meyer R. & Roy P. V. (1992)** *Wild LIFE - A User Manual.* 81 p.

[Aït-Kaci & al. 1986] **Aït-Kaci H. & Nasr R. (1986)** *LOGIN : a Logic Programming Language with Built-in Inheritance.* Journal of Logic Programming, **3**/1: pp. 185-215.

[Akasaka & al. 1989] **Akasaka K., Kubo Y., Fukumoto F., Fukushima H. & Hagiwara K. (1989)** *Language ToolBox (LTB) - A Program Library of NLP Tools.* Technical Report, TR-521, ICOT, November 1989, 29 p.

[Andreoli & al. 1991] **Andreoli J.-M. & Pareschi R. (1991)** *Communication as Fair Distribution of Knowledge.* Proc. OOPSLA'91, 1991, ACM Press ed., vol. 1/1: pp. 212-229.

[Apple Computer 1993] **Apple Computer (1993)** *AppleScript Language Guide.* Apple ed., Developer Technical Publications, Apple, Cupertino, 280 p.

[Apple Computer Inc. 1992] **Apple Computer Inc. (1992)** *Dylan - An object-oriented dynamic language.* Apple Eastern Research and Technology, 167 p.

[Assimi 1994] **Assimi A. B. (1994)** *Méthodes statistiques de traitement de corpus textuels, et étude détaillée d'un analyseur morpho-syntaxique probabiliste en vue de son intégration dans un système de TAO fondée sur le dialogue.* Rapport de DEA, GETA-IMAG, septembre 1994, 80 p.

[Bachut 1991] **Bachut D. (1991)** *Industrialisation d'un système de TAO français-anglais pour la documentation technique.* Proc. Génie Linguistique 91, Versailles, 1991, vol. 1/1: pp. 221-227.

[Backofen & al. 1992] **Backofen R. & Smolka G. (1992)** *A Complete and Recursive Feature Theory.* Research Report, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, September 1992, 32 p.

[Balkan 1992] **Balkan L. (1992)** *Translation Tools.* Meta, **37**/3: pp. 408-420.

[Bar-Hillel 1960] **Bar-Hillel Y. (1960)** *The Present Status of Automatic Translation of Languages.* In "Advances in Computers", Academic Press ed., New-York, pp. 91-163.

[Barnett & al. 1990] **Barnett J., Knight K., Mani I. & Rich E. (1990)** *Knowledge and Natural Language Processing.* CACM, **33**/8: pp. 50-71.

[Bennet & al. 1985] **Bennet W. S. & Slocum J. (1985)** *The LRC Machine Translation System.* Computational Linguistics, **11**/2-3, April-September 1985: pp. 111-121.

[Bergstein 1991] **Bergstein P. L. (1991)** *Object-Preserving Class Transformations.* Proc. OOPSLA'91, Vancouver - Canada, 1991, ACM Press ed., vol. 1/1: pp. 299-313.

[Blanchon 1990a] **Blanchon H. (1990a)** *HyperAriane : architecture de la partie "micro" d'un système de TAO personnelle.* Rapport de DEA, Institut National Polytechnique de Grenoble (ENSIMAG) & Université Joseph Fourier (UFR IMA), 26 juin 1990, 63 p.

[Blanchon 1990b] **Blanchon H. (1990b)** *LIDIA-1 : Un prototype de TAO personnelle pour rédacteur unilingue.* Proc. Avignon-90, conférence spécialisée : Le Traitement Automatique des Langues Naturelles et ses Applications, Avignon, France, 28 mai-1 juin 1990, EC2 ed., vol. 1/2: pp. 51-60.

[Blanchon 1994] **Blanchon H. (1994)** *LIDIA-1 : une première maquette vers la TA interactive "pour tous".* Nouvelle Thèse, Université Joseph Fourier - Grenoble 1, 321 p.

[Boitet 1982] **Boitet C. Réd. (1982)** *"DSE-1"— Le point sur ARIANE-78 début 1982.* Contrat ADI/CAP-Sogeti/Champollion (3 vol.), GETA, Grenoble, avril 1982, 368 p.

[Boitet 1988] **Boitet C. (1988)** *Representation and computation of units of translation for Machine Interpretation of spoken texts.* GETA - ATR, 1988, 20 p.

[Boitet 1991] **Boitet C. (1991)** *Twelve Problems for Machine Translation.* Proc. CICL - 91 (International Conference on Current Issues in Computational Linguistics), Penang, June 1991, vol. 1/1: pp. 45-56.

[Boitet 1993a] **Boitet C. (1993a)** *Crucial open problems in Machine Translation & Interpretation.* Proc. BKK'93, Bangkok, Thailand, 17-20 March 1993, vol. 1/1.

[Boitet 1993b] **Boitet C. (1993b)** *Integration of Heterogeneous Components for Speech Translation: the "Whiteboard" Architecture and an Architectural Prototype.* ATR Interpreting Telecommunications Research Laboratories, 8/30/1993, 20 p.

[Boitet 1993c] **Boitet C. (1993c)** *TA et TAO à Grenoble… 32 ans déjà !* T.A.L. (revue semestrielle de l'ATALA), **33**/1—2, Spécial Trentenaire: pp. 45-84.

[Boitet & al. 1982] **Boitet C., Guillaume P. & Quézel-Ambrunaz M. (1982)** *ARIANE-78: an integrated environment for automatic translation and human revision.* Proc. COLING-82, Prague, July 5-10 1982, North-Holland, Ling. series 47 ed., vol. 1/1: pp. 19-27.

[Boitet & al. 1985] **Boitet C., Guillaume P. & Quézel-Ambrunaz M. (1985)** *A case study in software evolution: from ARIANE-78.4 to ARIANE-85.* Proc. International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Language, Colgate University, Hamilton, New York, 14-16 août 1985, vol. 1/1: pp. 27-58.

[Boitet & al. 1982] **Boitet C., Hue & Collomb Réd. (1982)** *"DSE-2" — Spécification du système Ariane-X.* Projet ESOPE Contrat ADI/CAP-Sogeti/Champollion, GETA- Champollion - Cap Sogeti France, 24 juin 1982, 50 p.

[Boitet & al. 1994] **Boitet C. & Seligman M. (1994)** *The "Whiteboard" Architecture: A Way to Integrate heterogeneous components of NLP Systems.* Proc. COLING-94, Kyoto, Japan, August 1994, Makoto Nagao & ICCL ed., vol. 1/2: pp. 426-430.

[Booch 1992] **Booch G. (1992)** *Conception orientée objets et applications.* Addison-Wesley ed., Addison-Wesley, 588 p.

[Bouchard & al. 1992a] **Bouchard L., Emirkanian L., Estival D., Fay-Varnier C., Fouqueré C., Prigent G. & al. (1992a)** *EGL: a French Linguistic Development Environment.* Proc. Avignon'92, Avignon, France, 1-6 June 1992, vol. 1/1: pp. 227-238.

[Bouchard & al. 1992b] **Bouchard L., Emirkanian L., Estival D., Fay-Varnier C., Fouqueré C., Prigent G. & al. (1992b)** *First Results of a French Linguistic Development Environment.* Proc. COLING-92, Nantes, France, 1992, vol. 4/4: pp. 1177-1181.

[Bouchard & al. 1990] **Bouchard L. H. & Emirkanian L. (1990)** *Développement d'un analyseur morpho-syntaxique pour le français.* Proc. Les industries de la langue (perspectives des années 1990), Montréal, canada, 1990, Société des Traducteurs de Québec ed., vol. 1/2: pp. 115-130.

[Bruno & al. 1992] **Bruno G. & Castella A. (1992)** *An Extended View of the Object-Oriented Paradigm.* Proc. Software Engineering & its Applications, Toulouse, France, vol. 1/1: pp. 713-722.

[Budde & al. 1991] **Budde R., Kautz K., Kuhlenkamp K. & Züllighoven H. (1991)** *Prototyping - An Approach to Evolutionary System Development.* Springer-Verlag, 203 p.

[Calder & al. 1992] **Calder P. & Linton M. (1992)** *The Object-Oriented Implementation of a Document Editor.* Proc. OOPSLA'92, Vancouver, Canada, 1992, ACM ed., vol. 1/1: pp. 164-165.

[Caroll & al. 1991] **Caroll J., Briscoe T. & Grover C. (1991)** *A Development Environment for Large Natural Language Grammars.* University of Cambridge Computer Laboratory, July 1991, 65 p.

[Carpenter 1992] **Carpenter B. (1992)** *The logic of Typed Feature Structures.* Cambridge University Press, 1 vol., 270 p.

[Cavazza & al. 1992] **Cavazza M. & Zweigenbaum P. (1992)** *Compréhension automatique du language naturel par construction de modèles.* Technique et Science Informatiques, **11**/4: pp. 119-138.

[Chappuy 1983] **Chappuy S. (1983)** *Formalisation de la description des niveaux d'interprétation des langues naturelles.* Nouvelle Thèse, UJF, Grenoble, 213 p.

[Chauché 1974] **Chauché J. (1974)** *Transducteurs et arborescences.* Thèse d'Etat, USMG - Grenoble I, 440 p.

[Chauché 1975] **Chauché J. (1975)** *Les langages ATEF et CETA.* AJCL, **microfiche 17**: pp. 21-39.

[Chikayama 1991] **Chikayama T. (1991)** *Introduction to KL1.* ICOT, August 1991, .

[Claris 1991] **Claris (1991)** *HyperCard Script Language Guide.* Claris, 583 p.

[Coad & al. 1992] **Coad P. & Yourdon E., ed. (1992)** *Analyse Orientée Objets.* MIPS ed., Prentice Hall - Masson, 194 p.

[Cointe 1993] **Cointe P. (1993)** *CLOS and Smalltalk.* In "Object-Oriented Programming - The CLOS Perspective", A. Paepcke ed., MIT Press, Cambridge, Massachusetts, pp. 215-250.

[Colmerauer 1970] **Colmerauer A. (1970)** *Les Système-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur.* Rapport Université de Montréal - groupe TAUM, septembre 1970, 43 p.

[Colmerauer 1985] **Colmerauer A. (1985)** *Prolog in 10 Figures.* **28**/12: pp. 1296-1310.

[Coulon & al. 1986] **Coulon D. & Kayser D. (1986)** *Informatique et Langage naturel: présentation générale des méthodes d'interprétation des textes écrits.* Technique et Science Informatiques, **5**/02: pp. 103-128.

[Courdier & al. 1993] **Courdier R., Hérin-Aine D. & Galéra R. (1993)** *Une démarche et un modèle de conception à base d'objets et de réseaux sémantiques.* Technique et Science Informatique, **12**/3: pp. 285-318.

[Coutaz 1988] **Coutaz J. (1988)** *Interface Homme-ordinateur : Conception et Réalisation.* Thèse d'Etat, Université Joseph Fourier, Grenoble, 402 p.

[Cutting & al. 1993] **Cutting D., Kupiec J., Pedersen J. & Sibun P. (1993)** *A practical Part-of-Speech Tagger.* Research Report, Xerox Palo Alto Research Center, 1993, 9 p.

[Cutting & al. 1991] **Cutting D. R., Pedersen J. & Halvorsen P.-K. (1991)** *An Object-Oriented Architecture for Text Retrieval.* Proc. RIAO-91, Barcelona, Spain, April 1991, vol. 1/1: pp. 285-298.

[Daelemans & al. 1992] **Daelemans W., Gazdar G. & Smedt K. D. (1992)** *Inheritance in Natural Language Processing.* Computational Linguistics, **18**/2: pp. 205-218.

[Delannoy 1990] **Delannoy J.-F. (1990)** *A Message Processing System with Object-Centered Semantics.* Proc. COLING-90, Helsinki, Finland, 20-25 August 1990, Hans Karlgren ed., vol. 3/3: pp. 333-335.

[Delannoy 1991] **Delannoy J.-F. (1991)** *Un système fondé sur les objets pour le suivi de situations a partir de textes en languages naturel.* Nouvelle thèse, Université d'Aix-Marseille III, 173 p.

[Divay 1985] **Divay M. (1985)** *Un système-expert de traitement des textes écrits.* Proc. 5ème Congrès de reconnaissance des formes et d'intelligence artificelle, Grenoble, France, 27-29 Novembre 1985, Afcet-Inria ed., vol. 1/2: pp. 473-487.

[Doedens & al. 1988] **Doedens C.-J. & Zuijlen J. v. (1988)** *AL - an ATN programming language.* BSO/Research, 28 January 1988, 79 p.

[Dony & al. 1992] **Dony C., Malenfant J. & Cointe P. (1992)** *Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and their Validations.* Proc. OOPSLA'92, Vancouver, Canada, 1992,: pp. 201-217.

[Dreyfus 1993] **Dreyfus H. L. (1993)** *What Computers still can't Do - A critique of Artificial Reason.* M. Press ed., MIT Press, Cambridge, 352 p.

[Ducournau & al. 1989] **Ducournau R. & Habib M. (1989)** *La multiplicité de l'héritage dans les langages à objets.* Techniques & Science Informatiques, **8**/1: pp. 41-62.

[Durand 1988] **Durand J.-C. (1988)** *TTEDIT - un système interactif et de transformation d'arbres décorés.* Nouvelle thèse, Université Joseph Fourier, 84 p.

[Dvorak & al. 1993] **Dvorak J. & Bunke H. (1993)** *Using CLOS to Implement a Hybrid Knowledge Representation Tool.* In "Object-Oriented Programming — The CLOS Perspective", A. Paepcke ed., The MIT Press, Cambridge, pp. 195-320.

[Earley 1970] **Earley J. (1970)** *An-Efficient Context-Free Parsing Algorithm.* CACM, **13**/2: pp. 94-102.

[EDR 1988] **EDR (1988)** *Electronic Dictionary Project.* Technical Guide, EDR, November 1988, 68 p.

[EDR 1990] **EDR (1990)** *EDR Technical Reports, an Overview of the Electronic Dictionaries.* Technical reports, TR-024, TR-025, TR-026, TR-027, TR-029, EDR, 1990, 176 p.

[Edvins 1993] **Edvins M. (1993)** *Objects Without Classes.* Frameworks, **7**/6: pp. 34-39.

[Emele & al. 1990] **Emele M. & Zajac R. (1990)** *Typed Unification Grammars.* Proc. COLING-90, Helsinki, Finland, 20-25 August 1990, Hans Karlgren ed., vol. 1/3: pp. 293-298.

[Erman & al. 1980] **Erman L. D. & Lesser V. R. (1980)** *The Hearsay-II Speech Understanding System: A Tutorial.* In "Trends in Speech Recognition", W. A. Lea ed., Prentice Hall, pp. 361-381.

[Evins 1994] **Evins M. (1994)** *Protocols.* FrameWorks, **8**/2: pp. 24-27.

[Friedman & al. 1971] **Friedman J., Bredt T. H., Doran R. W., Pollack B. W. & Martner T. S., ed. (1971)** *A Computer Model of Transformational Grammar.* Mathematical Linguistics and Automatic Language Processing, D. G. Hays ed., American Elsevier Publishing Company, Inc, New York, 166 p.

[Gaschler & al. 1994] **Gaschler J. & Lafourcade M. (1994)** *Manipulating human-oriented dictionaries with very simple tools.* Proc. COLING-94, Kyoto, Japan, August 5-9 1994, Makoto Nagao & ICCL ed., vol. 1/2: pp. 283-286.

[Gazdar & al. 1985] **Gazdar G., Klein E., Pullum G. & Sag I. (1985)** *Generalized Phrase Structure Grammar.* Blackwell, 1 vol., 276 p.

[Gazdar & al. 1989] **Gazdar G. & Mellish C. (1989)** *Natural Language Processing in Lisp - An introduction to Computational Linguistics.* Addison-Wesley Publishing Company, 1 vol., 524 p.

[Genthial 1991] **Genthial D. (1991)** *Contribution à la construction d'un système robuste d'analyse du français.* Nouvelle thèse, Université Joseph Fourier, 236 p.

[Gerber 1984] **Gerber R. (1984)** *Etude des possibilités de coopération entre un système fondé sur des techniques de compréhension implicite (système logico-syntaxique) et un système fondé sur des techniques de compréhension explicite (système expert).* Nouvelle Thèse, UJF, Grenoble, 105 p.

[Gilloux 1990] **Gilloux M. (1990)** *Traitement automatique des langues naturelles.* Proc. 3ème Ecole d'été tenue à l'ENSSAT, Lannion,: pp. 53.

[Girod 1991] **Girod X. (1991)** *Conception par Objets (Mecano : une méthode et un environnement de construction d'applications par objets).* Nouvelle thèse, Université Joseph Fourier - Grenoble I, 250 p.

[Goodman & al. 1991] **Goodman K. & Nirenburg S. (1991)** *The KBMT Project: A case study in knowledge-based machine translation.* Morgan Kaufmann, San Mateo, California, 1 vol., 331 p.

[Grover & al. 1993] **Grover C., Caroll J. & Briscoe T. (1993)** *The Alvey Natural Language Tools Grammar (4th Release).* Research Report, University of Cambridge Computer Laboratory, 29 January 1993, 131 p.

[Guilbaud 1980] **Guilbaud J.-P. (1980)** *Analyse morphologique de l'allemand en vue de la traduction par ordinateur de textes techniques spécialisés.* Doctorat de troisième cycle, Univerité de la Sorbonne Nouvelle, Paris III, 242 p.

[Guilbaud 1990] **Guilbaud J. P. (1990)** *Méthodes et représentations linguistiques en ARIANE-G5. Etude de l'étape d'analyse, introduction à l'étape de transfert.* Research Report, GETA, IMAG (UJF & CNRS), Grenoble, 1990, 22 p.

[Guimarães 1991] **Guimarães N. (1991)** *Building Generic User Interface Tools: an Experience with Multiple Inheritance.* Proc. OOPSLA'91, 1991, vol. 1/1: pp. 89-96.

[Guise 1993] **Guise D. (1993)** *KR: Constraint-based Knowledge Representation.* Internal Report, Carnegie Mellon University, October 1993, 45 p.

[Habert 1991] **Habert B. (1991)** *Spécialiser des règles syntaxiques.* Proc. RFIA-91, Lyon - Villeurbanne, France, 25 - 29 novembre 1991, vol. 2/2: pp. 873-878.

[Habert 1992] **Habert B. (1992)** *Olmes: un analyseur à objets pour grammaires d'unification.* Proc. Douzièmes Journées Internationales - Le Traitement du Langage Natural & ses Applications, Avignon, France, 1-6 juin 1992, vol. 1/1: pp. 215-225.

[Habert & al. 1993a] **Habert B. & Fleury S. (1993a)** *Suivi fin de l'analyse automatique du language naturel basé sur l'héritage et la discrimination multiples.* Proc. Représentation par Objets, La Grande Motte, France, 17-18 juin 1993, EC2 ed., vol. 1/1: pp. 77-88.

[Habert & al. 1993b] **Habert B. & Fleury S. (1993b)** *Vers des analyseurs réflexifs.* T.A.L (revue de l'ATALA), **34**/1: pp. 35-60.

[Harrison 1978] **Harrison M. A. (1978)** *Introduction to Formal Language Theory.* Addison-Wesley series in Computer Science and Information Processing, Addison-Wesley, 594 p.

[Hayes & al. 1991] **Hayes F. & Coleman D. (1991)** *Coherent Models for Object-Oriented Analysis.* Proc. OOPSLA'91, 1991, ACM Press ed., vol. 1/1: pp. 171-183.

[Helm & al. 1991] **Helm R. & Maarek Y. (1991)** *Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries.* Proc. OOPSLA'91, 1991, ACM ed., vol. 1/1: pp. 47-61.

[Hüe & al. 1991] **Hüe J.-F. & Jayez J.-H. (1991)** *Une famille d'analyseurs contextuels.* Proc. Les industries de la langue (perspectives des années 1990), Montréal, Canada, 22-24 novembre 1991, Société des traducteurs du Québec ed., vol. 1/2: pp. 43-57.

[Hutchins 1986] **Hutchins W. J., ed. (1986)** *Machine Translation - Past, Present, Future.* Computers and their Applications, E. Horwood ed., Ellis Hordwood Limited, New York/Chichester/Brisbane/Toronto, 382 p.

[Hutchins & al. 1992] **Hutchins W. J. & Somers H. L. (1992)** *An introduction to Machine Translation.* A. Press ed., Harcourt Brace Jovanovich, 362 p.

[Iida & al. 1992] **Iida H. & Arita H. (1992)** *Natural Language Dialogue Understanding on a Four-Layer Plan Recognition Model.* Journal of Information Processing, **15**/1: pp. 96-107.

[Jacobson 1991] **Jacobson I. (1991)** *Re-engineering of old systems to an object-oriented architecture.* Proc. OOPSLA'91, 1991, ACM Press ed., vol. 1/1: pp. 340-350.

[Jensen & al. 1993] **Jensen K., Heidorn G. E. & Richardson S. D., ed. (1993)** *Natural Language Processing: The PLNLP Approach.* Kluwer Academic Publishers, Boston/Dordrecht/London, 324 p.

[Johnson 1991] **Johnson M. (1991)** *Features and Formulae.* Computational Linguistics, **17**/2: pp. 131-151.

[Karttunnen 1984] **Karttunnen L. (1984)** *Features and Values.* Proc. COLING-84, Stanford University, California, 2-6 July 1984, Association for Computational Linguistics ed., vol. 1/1: pp. 28-33.

[Karttunnen 1991] **Karttunnen L. (1991)** *Finite-state Constraints.* Proc. CICL-91, USM, Penang, Malaysia, vol. 1/1: pp. 1-18.

[Karttunnen 1993] **Karttunnen L. (1993)** *Finite-State Lexicon Compiler.* Research Report, ISTL-NLTT-1993-04-02, Xerox PARC, Avril 1993, 18 p.

[Karttunnen & al. 1992] **Karttunnen L. & Beesley K. R. (1992)** *Two-Level Rule Compiler.* Rsearch Report, ISTL-92-2, Xerox PARC, October 1992, 15 p.

[Kay 1973] **Kay M. (1973)** *The MIND system.* In "Courant Computer Science Symposium 8: Natural Language Processing", R. Rustin ed., Algorithmics Press, New York, pp. 155-188.

[Kay 1980] **Kay M. (1980)** *The Proper Place of Men and Machines in Language Translation.* Research Report, CSL-80-11, Xerox, Palo Alto Research Center, octobre 1980, .

[Keene 1989] **Keene S. E. (1989)** *Object-Oriented Programming in Common Lisp.* Addison-Wesley, 266 p.

[Kiczales & al. 1992] **Kiczales G. & Lamping J. (1992)** *Issues in the Design and Specification of Class Libraries.* Proc. OOPSLA'92, Vancouver, Canada, 1992, ACM ed., vol. 1/1: pp. 435-451.

[Kiczales & al. 1991] **Kiczales G., Rivières J. d. & Bobrow D. G. (1991)** *The Art of the Metaobject Protocol.* MIT Press, 335 p.

[Kita & al. 1992] **Kita K., Morimoto T. & Sagayama S. (1992)** *Two-Level LR Parsing with a Category Reachability Test and its Application to Speech Recognition.* Research Report, NLC92-12, ATR, 1992, 7 p.

[Konstan & al. 1991] **Konstan J. A. & Rowe L. A. (1991)** *Developing a GUIDE using Object-Oriented Programming.* Proc. OOPSLA, 1991, ACM Press ed., vol. 1/1: pp. 75-88.

[Korson & al. 1990] **Korson T. & McGregor J. D. (1990)** *Understanding Object-Oriented: a Unifying Paradigm.* **33**/9: pp. 41-60.

[Koskenniemmi 1984] **Koskenniemmi K. (1984)** *A General Computational Model for Word-Form Recognition and Production.* Proc. COLING-84, Stanford University, California, 2-6 July 1984, ICCL&ACL ed., vol. 1/1: pp. 181-187.

[Krief 1992] **Krief P. (1992)** *Utilisation des langages objets pour le prototypage.* Masson, 248 p.

[Lafourcade 1991a] **Lafourcade M. (1991a)** *ATEF.* rapport de DEA/TALN, GETA-IMAG, avril 1991, 20 p.

[Lafourcade 1991b] **Lafourcade M. (1991b)** *ODILE-2, un outil pour traducteurs occasionnels sur Macintosh.* Proc. L'environnement traductionnel. La station de travail du traducteur de l'an 2001, Mons, Belgique, 25-27 avril 1991, Presses de l'Université du Québec ed., vol. 1/1: pp. 3-19.

[Lafourcade 1991c] **Lafourcade M. (1991c)** *Outils micro-informatiques pour traducteurs occasionnels.* Rapport de DEA, GETA-IMAG, juin 1991, 86 p.

[Lafourcade 1992a] **Lafourcade M. (1992a)** *Geta-Strings.* Logiciel GETA, Grenoble, Common Lisp Object System (MCL - CLOS), Macintosh, version 1.0.

[Lafourcade 1992b] **Lafourcade M. (1992b)** *Le problème de l'accès au lexique dans les outils pour rédacteurs. ODILE, une approche.* Proc. Séminaire Lexique, Toulouse, Pôle langage naturel et parole du GDR-PRC CHM ed., vol. 1/1: pp. 81-89.

[Lafourcade 1993a] **Lafourcade M. (1993a)** *Geta-Browser.* GETA-IMAG, Grenoble, Common Lisp Object System (MCL - CLOS), Apple Macintosh, version 2.2.

[Lafourcade 1993b] **Lafourcade M. (1993b)** *Inside LT (LT Revisited but still a Language for Transcriptions).* Research Repport, GETA-IMAG, December 1993, 47 p.

[Lafourcade 1993c] **Lafourcade M. (1993c)** *LEAF ou comment garder l'origine de l'ambiguïté.* Proc. Troisième Journées Scientifiques "Traductique-TA-TAO", Montréal, Canada, septembre 1993,: pp. (à paraître).

[Lafourcade 1993d] **Lafourcade M. (1993d)** *Projet Sambre - Génie Logiciel pour le Génie Linguiciel.* Rapport de recherche, GETA-IMAG, décembre 1993, 6 p.

[Lafourcade 1994a] **Lafourcade M. (1994a)** *Applying Pivot MT Techniques to Multi-dialectal Programming Language Editors.* (internal report), GETA-IMAG, janvier 1994, 6 p.

[Lafourcade 1994b] **Lafourcade M. (1994b)** *Geta-Frames Protocols.* Research Repport, GETA-IMAG, March 1994, 23 p.

[Lafourcade 1994c] **Lafourcade M. (1994c)** *ODILE: un outil personnel d'aide à la traduction.* Turjuman, **3**/1: pp. 13-21.

[Lafourcade 1994d] **Lafourcade M. (1994d)** *Re-Engineering with added Genericity of Specialized Languages for Linguistic Programming - A case study with the ATEF & LT SLLPs.* Proc. IACL'94, Penang, Malaysia, July 26-28 1994,: pp. (to be published).

[Lafourcade & al. 1993a] **Lafourcade M. & Sérasset G. (1993a)** *DOP (Dictionary Object Protocol).* GETA-IMAG, Grenoble, Common Lisp Object System (MCL - CLOS), Apple Macintosh, version 2.0.

[Lafourcade & al. 1993b] **Lafourcade M. & Sérasset G. (1993b)** *Geta-Grapher.* GETA-IMAG, Grenoble, Common Lisp Object System (MCL-CLOS), Apple Macintosh, version 1.1.

[Lenat & al. 1990] **Lenat D. B., Guha R. V., Pittman K., Pratt D. & Shepherd M. (1990)** *CYC: Toward Programs with Common Sense.* CACM, **33**/8: pp. 30-49.

[Lepage 1985] **Lepage Y. (1985)** *Un language de transcription.* DEA d'informatique, USMG-INPG, 43 p.

[Lepage 1986] **Lepage Y. (1986)** *A language for transcriptions.* Proc. COLING-86, Bonn, 1986, IKS ed., vol. 1/1: pp. 402—404.

[Lewis & al. 1991] **Lewis J. A., Henry S. M., Kafura D. G. & Schulman R. S. (1991)** *An Empirical Study of the Object-Oriented Paradigm and Software Reuse.* Proc. OOPSLA, vol. 1/1: pp. 184-196.

[Lin & al. 1993] **Lin D. & Goebel R. (1993)** *Context-Free Grammar Parsing by Message Passing.* Proc. PACLing, Vancouver, Canada, April 21-24, 1993, vol. 1/1: pp. 203-211.

[Massini & al. 1991] **Massini G., Napoli A., Colnet D., Léonard D. & Tombre K., ed. (1991)** *Les langages à objets.* IIA, InterEditions ed., Jacky Akoka, Paris, 584 p.

[Matsumoto & al. 1987] **Matsumoto Y. & Sugimura R. (1987)** *A parsing system based on logic programming.* Proc. IJCAI 87,.

[Matsura & al. 1992] **Matsura S. & Ohbayashi M. (1992)** *The Software Development Environment: dmCASE.* Journal of Information Processing, **15**/1: pp. 116-128.

[Meyer 1990] **Meyer B. (1990)** *Lessons from the Design of the Eiffel Libraries.* CACM, **33**/9: pp. 69-88.

[Mizogushi & al. 1982] **Mizogushi F. & Kondo S. (1982)** *A Software Environment for Developing Natural Language Understanding System.* Proc. COLING-82, Prague, July 5-10 1982, North-Holland, Ling. series 47 ed., vol. 1/1: pp. 233-238.

[Morimoto & al. 1992] **Morimoto T., Takezawa T. & Yato F. (1992)** *ATR's Speech Translation System: ASURA.* Morimoto T., Takezawa T. & Yato F. ATR Interpreting Telephony Research Laboratories, .

[Morin 1991] **Morin J.-Y. (1991)** *Intégration des connaissances en génie linguistique : niveaux, dimensions, objets et contraintes.* Proc. L'environnement traductionnel - La station de travail du traducteur de l'an 2001, Mons, Belgique, 25-27 avril, AUPELF&UREF, Presses de l'Université de Montréal ed., vol. 1/1: pp. 109-133.

[Myers & al. 1992] **Myers B. A., Giuse D. A. & Zanden B. V. (1992)** *Declarative Programming in a Prototype-Instance System: Object-Oriented Programming without writing Methods.* Proc. OOPSLA, Vancouver, Canada, 1992, ACM ed., vol. 1/1: pp. 184-200.

[Myers 1991] **Myers J. K. (1991)** *NP: An ATMS-Based Plan Inference System that uses Feature Structures.* Research Report, TR-I-023, ATR, September 1991, 10 p.

[Nakamura 1988] **Nakamura J.-i. (1988)** *A Software System for Machine Translation.* Doctoral Thesis, Kyoto University, 181 p.

[Nerson 1992] **Nerson J.-M. (1992)** *Applying Object-Oriented Analysis and Design.* CACM, **35**/9: pp. 63-74.

[Nierstrasz & al. 1990] **Nierstrasz O. & Pintado X. (1990)** *Class Management for Software Communities.* CACM, **33**/9: pp. 91-103.

[Nirenburg 1989a] **Nirenburg S. (1989a)** *KBMT-89 Project Report.* Center for Machine Translation, Carnegie Mellon University, Pittsburg, avril 1989, .

[Nirenburg 1989b] **Nirenburg S. (1989b)** *Knowledge-Based Machine Translation.* Machine Translation, **4**: pp. 5-24.

[Nishida & al. 1993] **Nishida T. & Takeda H. (1993)** *Towards the Knowledgeable Community.* Proc. KB&KS'93, Tokyo, Japan, December 1993, JIPDEC ed., vol. 1/1: pp. 157-166.

[Norvig 1992] **Norvig P. (1992)** *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann Publishers, San Mateo - California, 948 p.

[Paepcke 1993] **Paepcke A., ed. (1993)** *Object-Oriented Programming - The CLOS Perspective.* A. Paepcke ed., MIT Press, Cambridge, Massachusetts, 352 p.

[Palmer 1990] **Palmer P. (1990)** *Etude d'un analyseur de surface de la langue naturelle - Application à l'indexation automatique de textes.* Nouvelle thèse, Université Joseph Fourier - Grenoble 1, 292 p.

[Pereira & al. 1986] **Pereira F. & Warren D. (1986)** *Definite Clause Grammars for Language Analysis.* In "Readings in Natural Language Processing", B. J. Grosz, K. Spark Jones & B. L. Webber ed., Morgan Kaufmann Publishers, Los Altos, California, pp. 101-124.

[Perry 1988] **Perry J. M. (1988)** *Perspective on Software Reuse.* Research Report, Software Engineering Institute - CMU, Pittsburg, September 1988, 10 p.

[Porter 1992] **Porter H. H. (1992)** *Separating the subtype hierarchy from the inheritance of implementation.* JOOP, **February 1992**: pp. 20-29.

[Pritchett & al. 1990] **Pritchett B. L. & Reitano J. W. (1990)** *Parsing with on-line principles: a psychologically plausible, object-oriented approach.* Proc. COLING-90, Helsinki, Finland, 20-25 August 1990, vol. 3/3: pp. 437-439.

[Quézel-Ambrunaz 1989] **Quézel-Ambrunaz M. (1989)** *Ariane-G5, le moniteur, manuel d'utilisation.* GETA, IMAG, Grenoble, mai 1989, .

[Quinton 1980] **Quinton P. (1980)** *Contribution à la reconnaissance de la parole. Utilisation de méthodes heuristiques pour la reconnaissance de phrases.* Thèse d'Etat, Université de Rennes, 239 p.

[Romanczuk 1990] **Romanczuk A. (1990)** *Towards a Methodological Guide for the Design of Object Oriented Programs.* Technical Report, TR-590, ICOT, August 1990, 32 p.

[Sabah 1988] **Sabah G. (1988)** *L'intelligence artificielle et le langage. Volume 1 : Représentation des connaissances.* M. Borillo & F. Nef ed., Langue, Raisonnement, Calcul, Hermès, Paris, 352 p.

[Sabah 1989] **Sabah G. (1989)** *L'intelligence artificielle et le langage. Volume 2 : Processus de compréhension*. M. Borillo & F. Nef ed., Langue, Raisonnement, Calcul, Hermès, Paris, 411 p.

[Sabah 1990] **Sabah G. (1990)** *CARAMEL: A flexible model for interaction between the cognitive processes underlying natural language understanding.* Proc. COLING-90, Helsinki, Finland, 20-25 August 1990, H. Karlgren ed., vol. 3/3: pp. 446-448.

[Sabah 1993] **Sabah G. (1993)** *Knowledge Representation and Natural Language Understanding.* **6**/3/4: pp. 155-186.

[Schneider 1989] **Schneider T. (1989)** *The METAL System. Status 1989.* Proc. MT Summit II, Munich, Germany, August 16-18, 1989, Christian Rohrer ed., vol. 1/1: pp. 128-136.

[Schubert 1988] **Schubert K. (1988)** *The Architecture of DLT - Interlingual or Double Direct?* Proc. New Directions in Machine Translation, Budapest, 18-19 August 1988, Floris Publications ed., vol. 1: pp. 131-143.

[Schütze & al. 1990] **Schütze C. T. & Reich P. A. (1990)** *Language Without a Central Pushdown Stack.* Proc. COLING-90, Helsinki, Finland, 20-25, Hans Karlgren ed., vol. 3/3: pp. 64-69.

[Seligman 1991] **Seligman M. (1991)** *Generating Discourses from Networks using an Inheritance-based Grammar.* PhD Thesis., 171 p.

[Seligman & al. 1994] **Seligman M. & Boitet C. (1994)** *A "whiteboard" architecture for automatic speech translation.* Proc. International Symposium on Spoken Dialogue, Waseda University, Tokyo, 1-12 November 1993,: pp. 4-8.

[Sérasset 1994a] **Sérasset G. (1994a)** *Approche œcuménique au problème du codage des structures linguistiques.* Proc. TALN-94, Marseille, France, 7-8 avril 1994, vol. 1/1: pp. 109-118.

[Sérasset 1994b] **Sérasset G. (1994b)** *Recent Trends of Electronic Dictionary research in Europe.* Technical Memorandum, TM-038, EDR, Tokyo, 1994, 90 p.

[Sérasset & al. 1993] **Sérasset G. & Blanc É. (1993)** *Une approche par acceptions pour les bases lexicales multilingues.* Proc. T–TA–TAO 93, Montréal, Canada (à paraître), 30 septembre-2 octobre 1993.

[Shieber 1986] **Shieber S. M. (1986)** *An Introduction to Unification-based Approaches to Grammar.* CSLI Lecture Note, CSLI, 105 p.

[St Clair 1991] **St Clair B. (1991)** *WOOD: a Persistent Object Database for MCL.* Apple, Avalaible in MCL CD-ROM & FTP (cambridge.apple.com), .

[Steele 1990] **Steele G. L. Jr. (1990)** *COMMON LISP. The Language.* Digital Press, 1030 p.

[Stefanini & al. 1992] **Stefanini M.-H., Berrendonner A., Lallich G. & Oquendo F. (1992)** *TALISMAN - Un système multi-agents gouverné par des lois linguistiques pour le traitement de la langue naturelle.* Proc. COLING-92, Nantes, France, 23-28 juillet 1992, Ch. Boitet ed., vol. 2/3: pp. 490-497.

[Stewart 1975] **Stewart G. (1975)** *Le langage de programmation REZO.* Rapport de Stage, TAUM, septembre 1975, 60 p.

[Stewart 1978] **Stewart G. (1978)** *Spécialisation et compilation des "Augmented Transition Networks": REZO.* Proc. COLING-78, Bergen, Norway, 14-18 August 1978, University of Bergen ed.,.

[Tan 1991] **Tan E. K. (1991)** *Generation of Lingware for Natural Language analysis based on the STCG (String-Tree Correspondence Grammar).* Research Report, Universiti Sains Malaysia, 80 p.

[Tanaka 1986] **Tanaka H. (1986)** *LangLAB.* Research Report, Departement of Computer, Science Tokyo Institute of Technology., November 1986, 6 p.

[Tanaka & al. 1993] **Tanaka H., Tokunaga T., Suresh K. G. & Inui K. (1993)** *Analysis and Generation Technologies.* Proc. KB&KS'93, Tokyo, Japan, December 1993, vol. 1/1: pp. 41-52.

[Tomasino 1990] **Tomasino I. (1990)** *ODILE, un Outil d'Intégration Extensible de Dictionnaires et de Lemmatiseurs.* Mémoire d'ingénieur CNAM, GETA-IMAG, 150 p.

[Tomita 1986] **Tomita M. (1986)** *Efficient Parsing for Natural Language. A fast algorithm for practical systems.* Kluwer, 201 p.

[Tomita 1990] **Tomita M. (1990)** *The Generalized LR Parser/Compiler V8-4 : A Software Package for Practical NL Projects.* Proc. COLING-90, Helsinki, Finland, 20-25 August 1990, Hans Karlgren ed., vol. 1/3: pp. 59-63.

[Uszkoreit 1989] **Uszkoreit H. (1989)** *Des faisceaux de traits aux types de données abstraits : nouvelles orientations des représentations et traitements linguistiques.* T.A. Information, **1-2**: pp. 11-35.

[Vasconcellos & al. 1985] **Vasconcellos M. & León M. (1985)** *SPANAM and ENGSPAN: Machine Translation at the Pan American Health Organization.* Computational Linguistics, **11**/2-3: pp. 122-136.

[Vauquois 1988] **Vauquois B. (1988)** *BERNARD VAUQUOIS et la TAO, vingt-cinq ans de Taduction Automatique, ANALECTES. BERNARD VAUQUOIS and MT, twenty-five years of MT.* Ch. Boitet (ed), Ass. Champollion & GETA, Grenoble, 700 p.

[Vauquois & al. 1985a] **Vauquois B. & Boitet C. (1985a)** *Automated Translation a Grenoble University.* Computational Linguistics, **11**/1: pp. 28-36.

[Vauquois & al. 1985b] **Vauquois B. & Chappuy S. (1985b)** *Static grammars : a formalism for the description of linguistics models.* Proc. Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages, Colgate University, Hamilton, N. Y., 14-16 August 1985, vol. 1/1: pp. 298-322.

[Vonk 1992] **Vonk R. (1992)** *Prototypage. L'utilisation efficace de la technologie CASE.* Masson, Prentice Hall, 163 p.

[Vosse & al. 1988] **Vosse T., Doedens C.-J., Zuijlen J. v. & Vendelmans R. (1988)** *ATN Parsing: Tools and Developments.* BSO/Research, 28 January 1988, 79 p.

[Wehrli 1990] **Wehrli É. (1990)** *STS: An Experimental Sentence Translation System.* Proc. COLING-90, Helsinki, Finland, 20-25 August 1990, H. Karlgren ed., vol. 1/3: pp. 76-78.

[Wehrli 1992] **Wehrli É. (1992)** *The IPS System.* Proc. COLING-92, Nantes, France, 23-28 juillet 1992, Ch. Boitet ed., vol. 3/4: pp. 870-874.

[White 1985] **White J. S. (1985)** *Characteristics of the METAL Machine Translation System at Production Stage.* Proc. Theoretical and Methodological Issues in Machine Translation of Natural Languages, Colgate University, Hamilton, New York, August 14-16 1985, vol. 1/1: pp. 359-369.

[Wirfs-Brock & al. 1990] **Wirfs-Brock R. J. & Johnson R. E. (1990)** *Surveying Current Research in Object-Oriented Design.* CACM, **33**/9: pp. 105-124.

[Woods 1970] **Woods W. A. (1970)** *Transition Network Grammars for Natural Language Analysis.* CACM, **13**/10: pp. 591-606.

[Yokota 1990] **Yokota E. (1990)** *How to Organize a Concept Hierarchy.* Proc. International workshop on electronic dictionaries, Oiso, Kanagawa, Japan, 1990,: pp. 50-57.

[Yokoyama 1989] **Yokoyama T. (1989)** *An Object-Oriented and Constraint-Based Knowledge Representation System for Design Object Modeling.* Technical Memorandum, TM-0809, ICOT, October 1989, 15 p.

[Zajac 1986] **Zajac R. (1986)** *SCSL: a linguistic specification language for MT.* Proc. COLING-86, Bonn, 25-29 Aug. 86, IKS ed.,: pp. 393—398.

[Zajac 1988] **Zajac R. (1988)** *Operations on Typed Feature Structures: Motivations and Definitions*. Internal Report, ATR Interpreting Telephony Research Laboratories, 1988, 32 p.

[Zajac 1992] **Zajac R. (1992)** *Toward Computer-Aided Linguistic Enginnering.* Proc. COLING-92, Nantes, France, 23-28 July 1992, Ch. Boitet ed., vol. 2/3: pp. 827-834.

Index

Annexes

Annexe A

Choix de CLOS

La ré-ingénierie des LSPL LT, ATEF et ROBRA ainsi que les réalisations des différents protocoles présentées dans les deuxième et troisième parties de ce document ont été implémentées en CLOS (Common Lisp Object System). Un certain nombre de caractéristiques ont motivé ce choix[86], nous mentionnons ci-dessous celles qui, à la fois, nous semblent les plus remarquables et nous ont été le plus utile.

COMMON LISP est un standard et de ce fait une bonne portabilité est théoriquement assurée. L'extension objet qu'est CLOS profite naturellement du dynamisme de LISP. Le développeur n'a pas à ce soucier de la gestion mémoire liée aux objets, ceux-ci sont sujets au ramasse-miettes quand c'est nécessaire.

Héritage multiple

Avec l'héritage multiple — une fonctionnalité relativement rare dans d'autres langages de programmation — une classes peut hériter du comportement de plus d'une classe.

La démarche classique d'analyse par objet quand on dispose de l'héritage multiple consiste à définir une classes de base combinée à des classes *fusionnantes* (*mixin-class*). Cette approche amène à la définition de protocoles de taille réduite liés au classes fusionnantes. De manière générale, les classes fusionnantes permettent l'héritage de leur protocole plutôt que de leurs attributs.

Des fonctionnalités intéressantes peuvent ainsi être développées en marge de l'application principale pour être introduites ensuite à la demande.

86 Rien n'empêche d'utiliser d'autres langages de programmation pour réimplémenter LT avec le même degré de généricité ou d'extensibilité. Cependant, les détails d'implémentation seront différents et parfois certaines approche devront être révisées. Par exemple, l'héritage multiple n'existe pas dans SmallTalk.

Méthodes multiples

La possibilité de sélectionner les méthodes sur plus d'un argument est quasiment inexistante parmi les autres langages de programmation. Pourtant, elle permet d'alléger la définition des protocoles.

Considérons le problème suivant. On dispose d'un certain nombre de classes d'imprimantes (laser, jet d'encre) et d'un certain nombre de types de documents (texte, image). Comment implémenter les méthodes permettant aux imprimantes de produire les différents types de documents.

Avec un langage de programmation classique — ne disposant pas de méthodes-multiples — la méthode *print* aura deux arguments (instance d'imprimante et de document) et spécialisera sur l'un ou l'autre de ces deux arguments. Si on choisit de faire la sélection sur la classe d'imprimante alors on écrira trois méthodes. Le corps de ces méthodes devront tester le type de l'argument document et agir de manière appropriée.

Par exemple, on pourrait écrire :

```
(defmethod print ((printer laser-printer) doc)
   …
   (document-print-laser doc printer)
   … )
(defmethod print ((printer jet-printer) doc)
   …
   (document-print-jet doc printer)
   … )
```

Il faut de plus définir les méthodes *document-print-laser* et *document-print-jet* :

```
(defmethod document-print-laser ((doc text-document) printer )
   …)
(defmethod document-print-jet   ((doc picture-document) printer )
   …)
```

Il faut donc définir un protocole inutilement compliqué pour chaque type d'impression L'ajout d'un nouveau type d'imprimante alourdira le protocole[87].

Avec les méthodes multiples, on définirait au plus autant de méthodes que la combinatoire entre les deux argument le permet :

```
(defmethod print ((printer laser-printer) (doc text-document))
   …)
(defmethod print ((printer laser-printer) (doc picture-document))
   …)
(defmethod print ((printer jet-printer) (doc text-document))
   …)
(defmethod print ((printer jet-printer) (doc picture-document))
   …)
```

87 Il existe une façon de faire plus directe consistant à faire, dans le corps de la méthode print une sélection selon le type du document et d'agir adéquatement sans passer par un sous-protocole. Cependant, cette méthode viole clairement les principes de la programmation à objets en rendant toute modularité impossible.

Le protocole se résume ici à la méthode *print.*

Spécialisation avec eql

CLOS peut distribuer les fonctions génériques non seulement sur la classe de ses arguments mais également si un des arguments est égal (*eql* au sens LISP) à une valeur particulière. En général, la valeur est un symbole LISP. Sans cette possibilité, un grand nombre de classes à instanciation unique devrait être créées afin de participer à la sélection des fonctions génériques.

Par exemple :

```
(defmethod printer-interpret ( (printer jet-printer) page-description
                               (language-type (eql 'ascii)))
  …)
(defmethod printer-interpret ( (printer jet-printer) page-description
                               (language-type (eql 'postscript)))
  …)
```

Cette fonctionnalité est donc appréciable aussi bien pour des raisons de lisibilité des programmes que de bonne gestion des ressources.

Changement de classe

En CLOS, il est possible de changer dynamiquement la classe d'un objet, avec la methode *change-class*. Il est possible de spécialiser des méthodes qui définissent comment les valeurs des champs doivent être mises à jour pour la nouvelle classe.

Pour plus de précisions sur CLOS nous proposons les lectures suivantes :

- [Steele 1990] pour COMMON LISP;
- [Keene 1989] pour la programmation objet en CLOS ;
- [Kiczales & al. 1991] pour le MOP;
- [Norvig 1992] pour l'application de COMMON LISP à l'intelligence Artificielle ;
- [Paepcke 1993] pour une compilation sur CLOS.

Annexe B

Compléments sur l'API d'ODILE 2

API pour les lemmatiseurs

Pour les lemmatiseurs, les points d'entrée étaient les suivants :

```
Procédure  Lem_ChargerDonnees;
```

La procédure Lem_ChargerDonnees charge les données nécessaires à l'analyse des occurrences fournies en entrée.

L'appel de cette procédure doit se faire une seule fois (avant de lancer la procédure Lem_Lemmatiser !).

```
Procédure  Lem_Lemmatiser (Chaîne : Chaîne; var StructSortieLem :
           Fichier texte);
```

La procédure Lem_Lemmatiser lemmatise le groupe d'occurrences *(Chaîne)* fourni en entrée. Elle construit la structure des résultats de lemmatisation *StructSortieLem* .

Chaîne est la chaîne de caractères à lemmatiser. Il s'agit d'une chaîne "normalisée", décrite par l'expression régulière : <chaîne sans blanc>(blanc <chaîne sans blanc>)*.

StructSortieLem est la structure de sortie, construite dans tous les cas

API pour les outils dictionnaires

Pour les dictionnaires, les points d'entrées suivants avaient été définis :

```
Fonction   LoadDT : Erreur;
```

Cette procédure permet de charger les données nécessaires pour le fonctionnement de l'outil dictionnaire

```
Fonction   UnloadDT : Erreur;
```

Cette procédure doit être appelée quand on ne veut plus utiliser l'outil dictionnaire.

```
Procédure  ControlDT (Event : Evènement);
```

On passe l'événement *(Event)* à l'outil dictionnaire.

```
Fonction   OpenDTBase (BaseName : chaîne;
           InfoNeeded : data;
           Lexique : Booléen) : Erreur;
```

Cette fonction ouvre une base de l'outil dictionnaire.

```
Fonction   CloseDTBase (BaseIndex : entier) : Erreur;
```

Cette fonction ferme une base de l'outil dictionnaire.

```
Function   FindArticle (Key : string;
           BaseIndex : integer;
           Seuil : integer): Erreur;
```

Cette procédure permet de rechercher des articles dont l'entrée est proche de *Key* dans la base *BaseIndex*. Proche signifie que la distance entre les articles trouvés et *Key* est inférieure à *Seuil.* Cette fonction écrit dans un fichier texte une liste de doublets. Chaque doublet représente la clé de l'article trouvé et la distance de cet article à *Key.* Le format du fichier est :

[(c,d)]* ()

c est une clé d'article et *d* est la distance de c par rapport à *Key.* Une liste vide (*()*) indique la fin de la liste.

On utilise cette procédure quant on veut juste s'assurer de l'existence d'un article dans une base, mais que l'on ne veut pas récupérer l'information associée.

```
Function   GetArticle (Key : chaîne;
           KeyNumber : integer;
           BaseIndex : entier;
           var Content : liste de données;
           Visible : boolean) : Erreur;
```

Cette procédure permet d'obtenir les informations associées à l'article repéré par la clé *Key* de la base repérée par *BaseIndex*. S'il existe plusieurs articles ayant la même clé, on obtiendra les *KeyNumber* premiers. On utilisera cette fonction pour récupérer les informations d'un article dont on sait qu'il existe dans la base.

```
Function   PutArticle (Key : string;
           Content : data;
           BaseIndex : integer) : Erreur;
```

Cette procédure crée un nouvel article de dictionnaire dans la base référencée par *BaseIndex*. Cet article a pour clé *Key* et pour contenu les informations (texte) référencées par *Content*.

Annexe C

Compléments sur le noyau LT4

Définition du langage LT4

Fonctions de base

Les fonctions de base permettent à l'utilisateur de définir et d'utiliser un nouveau trancripteur.

Fonctions de définitions

`transcriptor (name [variable-list] &key debug)` [macro]

Cette fonction définit un nouveau transcripteur nommé *name*, avec une liste optionnelle, *variable-list*, de paire <variable/valeur par défaut>.

Si *debug* vaut T, la définition du transcripteur en termes du GAM apparait dans une nouvelle fenêtre. Dans ce cas le transcripteur a déjà été évalué.

```
(transcriptor lg-example () :debug t)
```

`variable (<variable | (variable default-value)>+ [transcriptor])` [macro]

Cette fonction définit des variables pour le transcripteur. La valeur par défaut d'une variable est NIL si elle n'est pas fournie.

Si l'argument optionnelle *transcriptor* n'est pas fourni, le dernier transcripteur défini est considéré.

`initial-state (name [transcriptor])` [macro]

Cette fonction définit *name* comme l'état initial du transcripteur. L'utilisateur doit définir un et un seul état initial, sinon le comportement du transcripteur est indéterminé.

Si l'argument optionnelle*transcriptor* n'est pas fourni, le dernier transcripteur défini est considéré.

```
(initial-state (init lg-example))
```

`final-state (name+ [transcriptor])` `[macro]`

Cette fonction définit des états *name+* comme finals pour le transcripteur. Si aucun état n'est final, le trancripteur s'arrêtera en fin de bande avec un message d'erreur. Si l'état courant est un état final, le trancripteur s'arrête si la bande d'entrée est vide.

Si l'argument optionel *transcriptor* n'est pas fourni, le dernier transcripteur défini est considéré.

```
(final-states (f1 f2 lg-example))
```

```
transistions ((begin-state end-state [transcriptor]) transitions+)
  [Macro]
```

Cette fonction définit un ensemble de transitions entre les états *begin-state* et *end-state*. Si l'argument optionel *transcriptor* n'est pas fourni, le dernier transcripteur défini est considéré. Une transition doit prendre la forme suivante :

```
(input-string-expr condition action output-string-expr)
```

Une expression de chaîne d'entrée précédée par un *?* fait référence à la tête de lecture avant, sinon à la tête de lecture standard. Le détail de la syntaxe pour les expressions de chaînes d'entrée et de sortie est donnée plus avant.

Les conditions et les actions peuvent être n'importe quelle s-expression. Ces expressions peuvent faire référence aux variables du transcripteur.

```
(transitions (init path ex1)
        ((lg 1)
        ()  ()
        (string-upcase read-string))

        (? (lg 3)
        ()  ()
        read-string) )

     (transitions (init path ex3)
        ("A!3"
        ()  ()
        "à") )
```

Fonctions de transcription

`LT-transcribe (object <transcriptor | (transcriptor+)> keywords)` `[macro]`

Cette fonction applique un objet à la série de transcripteurs *transcriptor+* et retourne le contenu de la bande de sortie du dernier transcripteur sous forme de chaîne de caractères. *object* peut prendre plusieurs formes :

- *object* peut être une chaîne de caractères ;

```
(LT-transcribe lg-example "my-string")
```

- *object* peut être une chemin d'accès à un fichier. Le contenu du fichier sera transcrit.

```
(LT-transcribe-file lg-example
     "ccl:LT 2.0 Folder;Examples;File-to-Xcript.text")
```

- *object* peut être le symbole `:ask-file`. Une boite de dialogue demandera l'ouverture d'un fichier dont le contenu sera transcrit.

```
(LT-transcribe :ask-file lg-example)
```

- *object* peut être le symbole `:ask-dir`. Une boite de dialogue demandera l'ouverture d'un répertoire dont les fichiers seront transcrits.

```
(LT-transcribe :ask-dir lg-example)
```

Des mots-clés permettent certaines options :

`:show-result`

cet mot-clé permet d'afficher le résultat de la transcription d'un fichier dans une fenêtre. À T par défaut.

`:save-file`

Ce mot-clé permet d'enregistrer automatiquement de la transcription d'un fichier sous un autre ficher. Le nom du nouveau ficher est le nom du fichier d'origine suivit d'une extension ".lted". À NIL par défaut.

`:verbose`

Ce mot-clé permet l'affichage de certaines informations durant la transcription. À T par défaut.

`:extension`

Ce mot-clé permet de spécifier l'extension à rajouter au nom de fichier résultat. À ".lted" par défaut.

`:type`

Ce mot-clé permet de spécifier le type de fichier à créer. À :TEXT par défaut.

`:creator`

Ce mot-clé permet de spécifier le créateur du fichier à créer. À CCL2 par défaut.

```
(LT-transcribe :ask-file)
```

Fonctions spéciales

`(retract self)` `[méthode]`

Cette fonction fait reculer la tête de lecture avant jusqu'à la position de la tête de lecture standard.

`(stop self)` `[méthode]`

Cette fonction arrête immédiatement le transcripteur et retourne le résultat de la transcription.

`(suspend self)` `[méthode]`

Cette fonction arrête provisoirement le transcripteur. Le transcripteur peut reprendre son fonctionnment avec la fonction *resume*.

`(resume self)` `[méthode]`

Cette fonction relance le transcripteur après un appel à *suspend*.

Variables

Un transducteur a deux types de variables. Les variables externes sont généralement définies par l'utilisateur, mais certaines sont prédéfinies. Les variables internes qui sont prédéfinies et dont l'usage est moins courant. Ces variables sont utilisées dans les conditions et les actions.

Variables externes prédéfinies

Ces variables peuvent être librement accédées en lecture dans les actions et les conditions mais ne doivent pas être modifiées (le comportement du transcripteur serait alors indéterminé).

`read-string` `[variable externe prédéfinie]`

La chaîne lue est la sous-chaîne de la bande d'entrée venant d'être lue par la transition. Cette variable a été mise à jour pour la partie condition et peut être utilisée pour rendre invalide la transition courante. En cas d'échec de la lecture de la bande d'entrée, sa valeur n'est pas déterminée.

`lethal-string` `[variable externe prédéfinie]`

La chaîne létal est toujours égale à la sous-chaîne comprise entre la tête de lecture standard et la tête de lecture avant.

`self` `[variable externe prédéfinie]`

Cette variable contient l'objet correspondant à l'automate.

Variables internes

Les variables internes ne doivent pas être modifiées mais peuvent être utilisée en lecture. Ces variables sont accédés par l'intermédaire d'accesseurs.

`(input-tape self)` `[variable interne]`

Cette variable contient la bande d'entrée de l'automate. Cette bande n'est pas modifiée durant la transcription.

`(output-tape self)` `[variable interne]`

Cette variable contient la bande de sortie de l'automate. Cette bande est modifiée durant la transcription.

`(standard-pointer-index self)` `[variable interne]`

Cette variable contient la position courante de la tête de lecture standard.

`(forward-pointer-index self)` `[variable interne]`

Cette variable contient la position courante de la tête de lecture avant. Normallement, la tête de lecture avant est toujours positionnée après (ou à la même position que) la tête de lecture standard.

Expressions de chaînes (ise et ose)

Les expressions de chaînes sont des s-expressions qui sont interprétées comme des chaînes de caractères dans le contexte d'un transition LT. Il y a deux sortes d'expressions de chaînes correspondant à l'entrée (ce qui est lu) et à la sortie (ce qui est écrit).

Ici, nous n'explictions que les effets de bors des expressions de chaînes sur la variable *read-string* et sur la position des têtes de lecture.

La notation utilisée a la forme *se*: $\frac{< t,\ \sigma >\ \&\ conditions}{< t',\ \sigma' >}$ avec:

- *se* est une expression de chaînes (contenant éventuellement des variables);
- *object* est l'évaluation de l'expression de chaînes ;
- *conditions* sont les conditions devant être vérifées pour que la transition soit choisie;
- l'environnement (noté σ) est la liste des variables.

Expressions de chaînes d'entrée (ise)

`constant-string` `[ise]`

Lit une chaîne constante.

cs:

$$\frac{< cs.t,\ \sigma >}{< t,\ \sigma' \mid read - string = cs >} \qquad \frac{t}{failure}$$

```
"abc"
```

La chaîne d'entrée peut aussi être un caractère ou un nombre.

```
#\a, #\space, #\newline
7, 17, 23
```

`s-expression` `[ise]`

Lit une chaîne résultat de l'évaluation de la s-expression.

sexp:

$$\frac{< sexp.t,\ \sigma >\ \&\ (\text{eval sexp}) \text{ is a string}}{< t,\ \sigma' \mid read - string = (\text{eval sexp}) >}$$

$$\frac{sexp.s\ \&\ (\text{eval sexp}) \text{ is not a string}}{failure}$$

```
(concatenate 'string "abc" a-string)
```

avec *a-string* étant une variable du transcripteur contenant une chaîne de caractères.

`(lg n)` `[ise]`

Lit *n* caractères.

(lg n):

$$\frac{< e.t,\ \sigma > \ \&\ (\text{length e}) = n}{< t,\ \sigma' \mid \text{read} - \text{string} = e >} \qquad \frac{t\ \&\ (\text{length s}) < n}{\textit{failure}}$$

```
(lg 1)
```

Bien que cela ne soit pas permis par le modèle, *n* peut être une valeur négative permettant ainsi au pointeur de reculer. De cette manière, il est possible d'avoir un automate bidirectionnel. Il est également possible de donner une valeur négative en ce qui concerne la tête de lecture avant (par exemple *? (lg -1)*) mais dans ce cas, si sa position finale précéde celle de la tête de lecture standard, elle sera redéplacée à la position de la tête de lecture standard.

`(until s [:included])` `[ise]`

Lit la bande d'entrée jusquà ce que *s* soit rencontrée. *s* est une chaîne constante ou une s-expression. Si s n'existe pas c'est un cas d'échec. Si l'option *:included* est spécifiée, la tête de lecture se positionne après *s*, sinon avant.

(until e):

$$\frac{< a.e.t,\ \sigma >}{< e.t,\ \sigma' \mid \text{read} - \text{string} = a >} \qquad \frac{< e.t,\ \sigma >}{< e.t,\ \sigma' \mid \text{read} - \text{string} = \varepsilon >}$$

$$\frac{t}{\textit{failure}}$$

Avec *:included* :

(until e :included):

$$\frac{< a.e.t,\ \sigma >}{< t,\ \sigma' \mid \text{read} - \text{string} = a.e >} \qquad \frac{< e.t,\ \sigma >}{< e.t,\ \sigma' \mid \text{read} - \text{string} = e >}$$

$$\frac{t}{\textit{failure}}$$

```
(until "abc")
```

`(set s+)` `[ise]`

s+ est une suite arbitaire de chaînes (ou de caractères, nombres ou s-expressions). Cete fonction lit la bande d'entrée avec une chaîne *s* si *s* est le premier élement de *s+* contenu sur la bande d'entrée.

(set s):

$$\frac{< a.t,\ \sigma > \ \&\ a \in s}{< t,\ \sigma' \mid \text{read} - \text{string} = a >} \qquad \frac{< a.t,\ \sigma > \ \&\ a \notin s}{\textit{failure}}$$

```
(set "a" "e" "i" "o" "u" "y")
```

`rest` `[ise]`

Lit le reste de la bande d'entrée. Cette fonction n'échoue jamais.

(set s):

$$\frac{< t,\ \sigma >}{< \varepsilon,\ \sigma' \mid \text{read} - \text{string} = t >}$$

`null-string` [ise]

Lit la chaîne vide (""). N'échoue jamais. Il est plus efficace d'utiliser cette expression que la chaine vide "".

null-string:

$$\frac{< t,\ \sigma >}{< t,\ \sigma' \mid \text{read} - \text{string} = \varepsilon >}$$

`empty` [ise]

Lit la chaîne vide si la bande d'entrée est complétement lue (ce qui reste est vide), sinon échoue.

empty:

$$\frac{< \varepsilon,\ \sigma >}{< \varepsilon,\ \sigma' \mid \text{read} - \text{string} = \varepsilon >} \qquad \frac{< s,\ \sigma >}{\textit{failure}}$$

Cette expression n'est pas très orthodoxe car elle comporte une condition, mais son usage s'est révélé particulièrement utile.

Expressions de chaînes de sortie (ose)

Ces expressions ne peuvent pas échouer.

`constant-string` [ose]

Concatenante une chaîne constante à la bande de sortie.

```
output-string → output-string.constant-string
  "hello"
```

`s-expression` [ose]

Concatène la chaîne résultat de l'évaluation de la s-expresssion à la bande de sortie.

```
output-string → output-string.(eval s-expression)
```

La s-expression est évaluée dans le contexte de l'automate et peut donc contenir les variable du transcripteur.

```
  (concatenate 'string "hello" read-string)
```

`character` [ose]

```
  #\space, #\newline, #\a
```

```
output-string → output-string.(string c)
```

`number` [ose]

```
output-string → output-string.(string n)
  42, (cos (+ 555 111))

read-string                                                    [ose]

output-string → output-string.read-string

lethal-string                                                  [ose]
```

Correspond à la variable de même nom.

```
output-string → output-string.lethal-string

null-string                                                    [ose]

output-string → output-string
```

Exemple d'utilisation de *lg*

Cette fonction met un caractère sur 4 en majuscules. Le reste est en minuscules.

```
(transcriptor lg-example () :debug t)
(initial-state (init lg-example))

(transitions (init path lg-example)
  ((lg 1)
    ()  ()
    (string-upcase read-string)) )

(transitions (path init lg-example)
  ((lg 3)
    () ()
    (string-downcase read-string) )

   (rest    ;; in case there is less than 3 characters left
    () ()
    (string-downcase read-string))
    )
```

```
(LT-transcript lg-example "abcdefghij")
>"AbcdEfghIj"
```

Dans ce transcripteur, un caractère est lu puis mis en majuscules. Un changement d'état permet ensuite de lire trois caractères, de les passer en minuscules et de retourner à l'état d'origine. *rest* est utilisé pour terminer correctement la trasncription pour un nombre quelconque de caractères.

Exemple d'utilisation de *until*

La fonction *until* permet de lire le contenu de la bande d'entrée jusqu'à une chaîne donnée. Il faut savoir que si l'option *:included* n'est pas spécifiée il est possible de partir en boucle. En effet, s'il n'y a pas changement d'état et si la chaîne se trouve immédiatement en début de bande d'entrée, la chaîne vide sera constamment lue. C'est pourquoi, il est toujours préférable d'"avaler" la chaîne recherchée.

Le transcripteur suivant met en majuscules une chaîne de caractères (il est vrai qu'il existe des fonctions prédéfinies qui font cela très efficacement, mais il s'agit juste d'un exemple d'utilisation) :

```
(transcriptor until-example () :debug t)
(initial-state (init until-example))

(transitions (init s1 until-example)
   ( ? " "  ()  ()  "")

   ( (lg 1) ()  ()
    (string-upcase read-string)))

(transitions (s1 s2 until-example)
   ((until " ")   ;; read anything until the first " " (not included)
    () () read-string)

   (rest            ;; get the rest of the string (cannot fail)
    () () read-string
     )        )

(transitions (s2 s2 until-example)
  (" " () () read-string ))

(transitions (s2 s1 until-example)
    ((lg 1) () () (string-upcase read-string) ))


(LT-transcript until-example "bc d efa g hnj kk pl "))

> "Bc D Efa G Hnj Kk Pl "
```

Exemples de transcripteurs réels

Des applications de LT4 ont été réalisées pour des transcriptions du GETA

Dans les linguiciels développés avec le système ARIANE, des transcriptions particulières sont utilisées pour la représentations des chaînes. Deux transcripteurs pour convertir une chaîne de caractères standard en une chaîne ARIANE et vice-versa ont été définis et réalisés. Pour plus de détails sur la transcription, se référer aux exemples.

Ces transcripteurs ont été utilisés dans le projet LIDIA [Blanchon 1994].

Quelques fonctions utilitaires sont tout d'abord définies.

```
;; -*- Package: LT -*-
;; Transcripteur LT pour transcrire le Francais Mac en Francais Ariane.
;; Gilles Serasset & Mathieu Lafourcade

(IN-PACKAGE :LT)

;; Utilities used by the transcriptor.
;; ----------------------------------

(defvar chiffres '("1" "2" "3" "4" "5" "6" "7" "8" "9" "0"))

(defvar majuscules '("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"))

(defvar minuscules '("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"))

(defvar majuscules-diacritees '("Á" "À" "Â" "Ã" "Ç" "É"
                                "È" "Ê" "Ë" "Î" "Ï" "Ô"
                                "Ù" "Û" "Œ"))
```

```
(defvar minuscules-diacritees '("á" "à" "â" "ã" "ç" "é"
                                "è" "ê" "ë" "î" "ï" "ô"
                                "ù" "û" "œ"))

(defvar ponctuations '(" " "." ";" "," "¡" "?" "-" "/" ":" "'" "`" "\""))

(defun chiff-p (chaine)
  (member chaine chiffres :test #'equal))

(defun ponct-p (chaine)
  (member chaine ponctuations :test #'equal))

(defun maj-p (chaine)
  (member chaine majuscules :test #'equal))

(defun maj-diacr-p (chaine)
  (member chaine majuscules-diacritees :test #'equal))

(defun min-p (chaine)
  (member chaine minuscules :test #'equal))

(defun min-diacr-p (chaine)
  (member chaine minuscules-diacritees :test #'equal))
```

Mac-to-Ariane

```
;; -*- Package: LT -*-
;; Transcripteur LT pour transcrire le Francais
Macintosh en Francais Ariane.
;; Gilles Serasset & Mathieu Lafourcade

(IN-PACKAGE :LT)

;; The Mac to Ariane transcriptor
;; ------------------------------

(TRANSCRIPTOR    MAC-TO-ARIANE
                 NIL :debug nil)

(INITIAL-STATE (INIT MAC-TO-ARIANE))

(TRANSITIONS (INIT INIT MAC-TO-ARIANE)
  ((LG 1) (CHIFF-P READ-STRING) nil READ-STRING)
  ((LG 1) (PONCT-P READ-STRING) nil READ-STRING)
  (#\Newline NIL NIL READ-STRING)
 )

(TRANSITIONS (INIT TEST MAC-TO-ARIANE)
  (? (LG 1) (MAJ-P READ-STRING) nil "")
  (? (LG 1) (MAJ-DIACR-P READ-STRING) nil "")
 )

(TRANSITIONS (TEST MOT-MAJ MAC-TO-ARIANE)
  (? (LG 1) (MAJ-P READ-STRING) (retract self) "**")
  (? (LG 1) (MAJ-DIACR-P READ-STRING)
            (retract self) "**")
 )

(TRANSITIONS (TEST MOT-CAP MAC-TO-ARIANE)
  (? (LG 1) (MIN-P READ-STRING) (retract self) "*")
  (? (LG 1) (MIN-DIACR-P READ-STRING)
            (retract self) "*")
  (? (LG 1) (CHIFF-P READ-STRING) (retract self) "*")
  (? (LG 1) (PONCT-P READ-STRING) (retract self) "*")
  (? #\Newline NIL (retract self) "*")
  (? null-string nil (retract self) "*")
 )

(TRANSITIONS (INIT MOT-MIN MAC-TO-ARIANE)
  (? (LG 1) (MIN-P READ-STRING) nil "")
  (? (LG 1) (MIN-DIACR-P READ-STRING) nil ""))

(TRANSITIONS (MOT-CAP MOT-MIN MAC-TO-ARIANE)
  ("Á" NIL NIL "A!1")
  ("À" NIL NIL "A!2")
  ("Â" NIL NIL "A!3")
  ("Ä" NIL NIL "A!4")
  ("Ã" NIL NIL "A!6")
  ("Ç" NIL NIL "C!5")
  ("É" NIL NIL "E!1")
  ("È" NIL NIL "E!2")
  ("Ê" NIL NIL "E!3")
  ("Ë" NIL NIL "E!4")
  ("Î" NIL NIL "I!3")
  ("Ï" NIL NIL "I!4")
  ("Ô" NIL NIL "O!3")
  ("Ö" NIL NIL "O!4")
  ("Ù" NIL NIL "U!2")
  ("Û" NIL NIL "U!3")
  ("Ü" NIL NIL "U!4")
  ("Œ" NIL NIL "E!13")
  ((LG 1) nil NIL read-string)
 )

(TRANSITIONS (MOT-MIN INIT MAC-TO-ARIANE)
  ((LG 1) (PONCT-P READ-STRING) NIL READ-STRING)
  (#\Newline NIL NIL READ-STRING)
 )

(TRANSITIONS (MOT-MIN TEST MAC-TO-ARIANE)
  (? (LG 1) (MAJ-P READ-STRING) nil "")
  (? (LG 1) (MAJ-DIACR-P READ-STRING) nil "")
 )

(TRANSITIONS (MOT-MIN MOT-MIN MAC-TO-ARIANE)
  ("á" NIL NIL "A!1")
  ("à" NIL NIL "A!2")
  ("â" NIL NIL "A!3")
  ("ä" NIL NIL "A!4")
```

```
  ("ã" NIL NIL "A!6")
  ("ç" NIL NIL "C!5")
  ("é" NIL NIL "E!1")
  ("è" NIL NIL "E!2")
  ("ê" NIL NIL "E!3")
  ("ë" NIL NIL "E!4")
  ("î" NIL NIL "I!3")
  ("ï" NIL NIL "I!4")
  ("ô" NIL NIL "O!3")
  ("ö" NIL NIL "O!4")
  ("ù" NIL NIL "U!2")
  ("û" NIL NIL "U!3")
  ("ü" NIL NIL "U!4")
  ("œ" NIL NIL "E!13")
  ((LG 1) (MIN-P READ-STRING)
          NIL  (STRING-UPCASE READ-STRING))
 )

(TRANSITIONS (MOT-MAJ INIT MAC-TO-ARIANE)
  ((LG 1) (PONCT-P READ-STRING) NIL READ-STRING)
  (#\Newline NIL NIL READ-STRING)
 )

(TRANSITIONS (MOT-MAJ MAJ-TEST MAC-TO-ARIANE)
  (? (LG 1) (MIN-P READ-STRING) NIL "")
  (? (LG 1) (MIN-DIACR-P READ-STRING) NIL "")
 )

(TRANSITIONS (MOT-MAJ MOT-MAJ MAC-TO-ARIANE)
  ("Á" NIL NIL "A!1")
  ("À" NIL NIL "A!2")
  ("Â" NIL NIL "A!3")
  ("Ä" NIL NIL "A!4")
  ("Ã" NIL NIL "A!6")
  ("Ç" NIL NIL "C!5")
  ("É" NIL NIL "E!1")
  ("È" NIL NIL "E!2")
  ("Ê" NIL NIL "E!3")
  ("Ë" NIL NIL "E!4")
  ("Î" NIL NIL "I!3")
  ("Ï" NIL NIL "I!4")
  ("Ô" NIL NIL "O!3")
  ("Ö" NIL NIL "O!4")
  ("Ù" NIL NIL "U!2")
  ("Û" NIL NIL "U!3")
  ("Ü" NIL NIL "U!4")
  ("Œ" NIL NIL "E!13")
  ((LG 1) NIL NIL (STRING-UPCASE READ-STRING))
 )

(TRANSITIONS (MAJ-TEST MOT-MIN MAC-TO-ARIANE)
  (? (LG 1) (MIN-P READ-STRING) (retract self) "**")
  (? (LG 1) (MIN-DIACR-P READ-STRING)
            (retract self) "**")
 )

(TRANSITIONS (MAJ-TEST MOT-UNCAP MAC-TO-ARIANE)
  (? (LG 1) (MAJ-P READ-STRING) (retract self) "*")
  (? (LG 1) (MAJ-DIACR-P READ-STRING)
            (retract self) "*")
  (? (LG 1) (CHIFF-P READ-STRING)
            (retract self) "*")
  (? (LG 1) (PONCT-P READ-STRING) (retract self) "*")
  (? #\Newline NIL (retract self) "*")
  (? null-string nil (retract self) "*")
 )

(TRANSITIONS (MOT-UNCAP MOT-MAJ MAC-TO-ARIANE)
  ("á" NIL NIL "A!1")
  ("à" NIL NIL "A!2")
  ("â" NIL NIL "A!3")
  ("ä" NIL NIL "A!4")
  ("ã" NIL NIL "A!6")
  ("ç" NIL NIL "C!5")
  ("é" NIL NIL "E!1")
  ("è" NIL NIL "E!2")
  ("ê" NIL NIL "E!3")
  ("ë" NIL NIL "E!4")
  ("î" NIL NIL "I!3")
  ("ï" NIL NIL "I!4")
  ("ô" NIL NIL "O!3")
  ("ö" NIL NIL "O!4")
  ("ù" NIL NIL "U!2")
  ("û" NIL NIL "U!3")
  ("ü" NIL NIL "U!4")
  ("œ" NIL NIL "E!13")
  ((LG 1) nil NIL (string-upcase read-string))
 )
```

Tests sur le transcripteur MAC-TO-ARIANE

```
(LT-transcribe "anticonstitutionnellement" MAC-TO-ARIANE) ->
"ANTICONSTITUTIONNELLEMENT"

(LT-transcribe "Ariane" MAC-TO-ARIANE) -> "*ARIANE"

(LT-transcribe "ARIANE" MAC-TO-ARIANE) -> "**ARIANE"

(LT-transcribe "ARIANE a LES Yeux bleus" MAC-TO-ARIANE) ->
   "**ARIANE A **LES *YEUX BLEUS"

(LT-transcribe "génération à Àbbatre" MAC-TO-ARIANE) ->
   "GE!1NE!1RATION A!2 *A!2BBATRE"

(LT-transcribe "Être ou ne pas Être" MAC-TO-ARIANE) ->
   "*E!3TRE OU NE PAS *E!3TRE"

(LT-transcribe "LATeX" MAC-TO-ARIANE) -> "**LAT*EX"

(LT-transcribe "LaTeX" MAC-TO-ARIANE) -> "*LA*TE*X"

(LT-transcribe "LaTEX" MAC-TO-ARIANE) -> "*LA**TEX"
```

```
(LT-transcribe "TEST, de ponc.tu,a;Tion" MAC-TO-ARIANE) ->
   "**TEST, DE PONC.TU,A;*TION"

(LT-transcribe "Ô ô ÔÛÂ" MAC-TO-ARIANE) -> "*O!3 O!3 **O!3U!3A!3"
```

Ariane-to-Mac

```
;; -*- Package: LT -*-
;; Transcripteur LT pour transcrire le Francais
Ariane en Francais Macintosh.
;; Gilles Serasset & Mathieu Lafourcade

(IN-PACKAGE :LT)

;; The Ariane to Mac transcriptor
;; ------------------------------

(TRANSCRIPTOR ARIANE-TO-MAC
              NIL :debug NIL)

(INITIAL-STATE (INIT ARIANE-TO-MAC))

(TRANSITIONS (INIT INIT ARIANE-TO-MAC)
  ((LG 1) (CHIFF-P READ-STRING) nil READ-STRING)
  ((LG 1) (PONCT-P READ-STRING) nil READ-STRING)
  (#\Newline NIL NIL READ-STRING))

(TRANSITIONS (INIT TEST ARIANE-TO-MAC)
  ("*" NIL NIL "")
 )

(TRANSITIONS (INIT MOT-MIN ARIANE-TO-MAC)
  (empty NIL NIL "")
 )

(TRANSITIONS (TEST MOT-MAJ ARIANE-TO-MAC)
  ("*" NIL NIL "")
 )

(TRANSITIONS (TEST MOT-CAP ARIANE-TO-MAC)
  (empty NIL NIL "")
 )

(TRANSITIONS (MOT-CAP MOT-MIN ARIANE-TO-MAC)
  ("A!1" NIL NIL "Á")
  ("A!2" NIL NIL "À")
  ("A!3" NIL NIL "Â")
  ("A!4" NIL NIL "Ä")
  ("A!6" NIL NIL "Ã")
  ("C!5" NIL NIL "Ç")
  ("E!1" NIL NIL "É")
  ("E!2" NIL NIL "È")
  ("E!3" NIL NIL "Ê")
  ("E!4" NIL NIL "Ë")
  ("I!3" NIL NIL "Î")
  ("I!4" NIL NIL "Ï")
  ("O!3" NIL NIL "Ô")
  ("O!4" NIL NIL "Ö")
  ("U!2" NIL NIL "Ù")
  ("U!3" NIL NIL "Û")
  ("U!4" NIL NIL "Ü")
  ("E!13" NIL NIL "Œ")
  ((LG 1) nil NIL read-string)
 )

(TRANSITIONS (MOT-MIN INIT ARIANE-TO-MAC)
  ((LG 1) (PONCT-P READ-STRING) NIL READ-STRING)
  (#\Newline NIL NIL READ-STRING)
 )

(TRANSITIONS (MOT-MIN TEST ARIANE-TO-MAC)
  ("*" NIL NIL "")
 )

(TRANSITIONS (MOT-MIN MOT-MIN ARIANE-TO-MAC)
  ("A!1" NIL NIL "á")
  ("A!2" NIL NIL "à")
  ("A!3" NIL NIL "â")
  ("A!4" NIL NIL "ä")
  ("A!6" NIL NIL "ã")
  ("C!5" NIL NIL "ç")
  ("E!1" NIL NIL "é")
  ("E!2" NIL NIL "è")
  ("E!3" NIL NIL "ê")
  ("E!4" NIL NIL "ë")
  ("I!3" NIL NIL "î")
  ("I!4" NIL NIL "ï")
  ("O!3" NIL NIL "ô")
  ("O!4" NIL NIL "ö")
  ("U!2" NIL NIL "ù")
  ("U!3" NIL NIL "û")
  ("U!4" NIL NIL "ü")
  ("E!13" NIL NIL "œ")
  ((LG 1) NIL
          NIL (STRING-downcase READ-STRING))
 )

(TRANSITIONS (MOT-MAJ INIT ARIANE-TO-MAC)
  ((LG 1) (PONCT-P READ-STRING) NIL READ-STRING)
  (#\Newline NIL NIL READ-STRING)
 )

(TRANSITIONS (MOT-MAJ MAJ-TEST ARIANE-TO-MAC)
  ( "*" nil NIL "")
 )

(TRANSITIONS (MOT-MAJ MOT-MAJ ARIANE-TO-MAC)
  ("A!1" NIL NIL "Á")
  ("A!2" NIL NIL "À")
  ("A!3" NIL NIL "Â")
  ("A!4" NIL NIL "Ä")
  ("A!6" NIL NIL "Ã")
  ("C!5" NIL NIL "Ç")
  ("E!1" NIL NIL "É")
  ("E!2" NIL NIL "È")
  ("E!3" NIL NIL "Ê")
  ("E!4" NIL NIL "Ë")
  ("I!3" NIL NIL "Î")
  ("I!4" NIL NIL "Ï")
  ("O!3" NIL NIL "Ô")
  ("O!4" NIL NIL "Ö")
  ("U!2" NIL NIL "Ù")
  ("U!3" NIL NIL "Û")
  ("U!4" NIL NIL "Ü")
  ("E!13" NIL NIL "Œ")
```

```
  ((LG 1) NIL NIL read-string)
 )

(TRANSITIONS (MAJ-TEST MOT-MIN ARIANE-TO-MAC)
  ("*" nil nil "")
 )

(TRANSITIONS (MAJ-TEST MOT-UNCAP ARIANE-TO-MAC)
  (empty NIL nil "")
 )

(TRANSITIONS (MOT-UNCAP MOT-MAJ ARIANE-TO-MAC)
  ("A!1" NIL NIL "á")
  ("A!2" NIL NIL "à")
  ("A!3" NIL NIL "â")
  ("A!4" NIL NIL "ä")
  ("A!6" NIL NIL "ã")
  ("C!5" NIL NIL "ç")
  ("E!1" NIL NIL "é")
  ("E!2" NIL NIL "è")
  ("E!3" NIL NIL "ê")
  ("E!4" NIL NIL "ë")
  ("I!3" NIL NIL "î")
  ("I!4" NIL NIL "ï")
  ("O!3" NIL NIL "ô")
  ("O!4" NIL NIL "ö")
  ("U!2" NIL NIL "ù")
  ("U!3" NIL NIL "û")
  ("U!4" NIL NIL "ü")
  ("E!13" NIL NIL "œ")
  ((LG 1) nil NIL (string-downcase read-string))
 )
```

Tests pour le transcripteur ARIANE-TO-MAC

```
(LT-transcribe "ANTICONSTITUTIONNELLEMENT" ARIANE-TO-MAC) ->
   "anticonstitutionnellement"

(LT-transcribe "*ARIANE" ARIANE-TO-MAC) -> "Ariane"

(LT-transcribe "**ARIANE" ARIANE-TO-MAC) -> "ARIANE"

(LT-transcribe "**ARIANE A **LES *YEUX BLEUS" ARIANE-TO-MAC) ->
   "ARIANE a LES Yeux bleus"

(time (LT-transcribe "GE!1NE!1RATION A!2 *A!2BBATRE" ARIANE-TO-MAC)) ->
   "génération à Àbbatre"

(time (LT-transcribe "*E!3TRE OU NE PAS *E!3TRE" ARIANE-TO-MAC)) ->
   "Être ou ne pas Être"

(LT-transcribe "**LAT*EX" ARIANE-TO-MAC) -> "LATeX"

(LT-transcribe "*LA*TE*X" ARIANE-TO-MAC) -> "LaTeX"

(LT-transcribe "*LA**TEX" ARIANE-TO-MAC) -> "LaTEX"

(time (LT-transcribe "**TEST, DE PONC.TU,A;*TION" ARIANE-TO-MAC)) ->
   "TEST, de ponc.tu,a;Tion"

(LT-transcribe "*O!3 O!3 **O!3U!3A!3" ARIANE-TO-MAC) -> "Ô ô ÔÛÂ"
```

Annexe D

Proposition d'un langage externe à partir de DÉCOR

Dans la seconde partie nous avons proposé un langage noyau générique adapté à la représentation de décorations linguistiques. L'implémentation de DÉCOR est en LISP et sa syntaxe lispienne n'est pas adaptée au développeur linguiste.

Nous proposons, ici, la syntaxe externe d'un langage dérivé de DÉCOR, CALIBAN. L'objectif de CALIBAN est de fournir un langage basé sur les décorations avec une syntaxe utilisable par le développeur linguiste.

Ce langage n'est pas à proprement parler une syntaxe externe de DÉCOR car certaines des fonctionnalités (par exemple, celle liées au dynamisme ou à l'extensibilité) n'apparaissent pas dans la syntaxe. Les possibilités offertes par le dynamisme existent mais sont implémentées au niveau de l'environnement de programmation. Ce langage est fermé, contrairement à DÉCOR qui est ouvert sur LISP.

La syntaxe qui suit a été partiellement implémentée sur l'éditeur multi-dialectes. Ce qui suit est extrait de [Lafourcade 1993d], un document de travail sur la définition de ce langage (il s'agit d'une introduction à CALIBAN destinée au linguiste).

The major goal of CALIBAN is to provide a typed language with the most straightforward syntax. It is why CALIBAN has a syntax very similar to programming (or scripting) languages like AppleScript or HyperTalk. The scripts are written in an "English like" way. Some features have been added from the original AppleScript, others have been deleted but if you are familiar with AppleScript, CALIBAN should not be a big trial to learn.

Decorations

Decorations in CALIBAN are like variables in other classical programming languages or scripting languages. They are called decorations mainly for historical reasons. The term "variables" and decorations can be used to denote the same kind of object.

Decorations can be typed or not. A type restricts the set of values the decoration can take. The semantic of operators sometimes relies on the type.

If a decoration is not typed then it can hold any kind of value. If the decoration is typed, it can hold only a value compatible with its type.

For instance, the following script:

```
copy 3 to x
5 + x
```

creates a variable x without any type and places 3 as its value. The second expression will return 8 as result.

It is possible to declare a decoration before using it. In that case the declared decoration is known globally and can be used inside other scripts. The previous script creates x locally (x is not declared) which can be done only with typeless decorations. For example:

```
decor x
copy 3 to x
5 + x
```

is equivalent to the previous script. Here x has been declared before 3 has been copied into x. You will get an error if you type the following script:

```
x + 3
```

In that case x has been declared neither globally or locally.

When you declare a typed decoration, it takes the form: **decor** *decorName* **is a** *typeExpression* where *decorName* is an identifier (we call identifiers symbols) and *typeExpression* is either the name of a declared type (a symbol) or an expression evaluating to a type. The "a" is optional and can be omitted or remplaced by "an".

The declaration can also take the two other equivalent forms:

```
decor myDecor is any type
decor myDecor
```

In these cases the decoration is declared without a specific type. It is the same as an "on the fly" decoration except that the decoration is known globally and can be used by other scripts.

To summarize, you can create decorations either locally ot globally (in that latter case we say it is declared). Local decorations are never typed and thus can have any kind of value. Global decorations may be typed.

Copy and Set

There are two basic operators allowing to place a value in a variable. Values are copied by the "copy" operator. Values are shared if "set" is used.

Copy

This operator copies data to a variable. Its syntax is:

```
copy data to variableName
copy data to objectPath
```

variableName is a symbol denoting a variable. *objectPath* is a sequence denoting a subpart of a variable (for aggregated variables).

The data is also put into the special variable result (cf Specials Variables).

If the variableName refers to no declared variable then a new local variable is created. For example:

```
copy 3 to x
```

Set

This operator sets the value of a variable. Its syntax is:

```
set variableName to data
set objectPath to data
```

variableName and *objectPath* denote respectively the same objects as in copy.

The data is also put into the special variable result (cf Special Variables).

The main difference between *set* and *copy* is in the possibility to do "data sharing". If data is a symbol of an existing variable then the value of this variable is placed in *variableName* (as *copy*) but if this value changes then the value of *variableName* is updated (which is not the case with *copy*). For example:

```
set y to {a, b, c, d}
set x to y
set item 1 of y to "hello"
x
⇒ {"hello", b, c, d}
```

There is a link established between the variables x and y. When y changes, x changes.

Data types

CALIBAN provides different kinds of types. The different types are: *text*, *number*, *boolean*, *symbol*, *list*, *enumeration* and *aggregate.*

Texts

A text decoration can hold any text you wish, for example "hello text".

```
copy "Caliban: the scripting language" to someText
copy "Caliban:" & "something new" to someText
```

It is possible to define a type which is equivalent to the type text (althought it is not very interesting):

```
type myTextType is a text
decor someText is a myTextType
copy "Hello" to someText
```

Numbers

CALIBAN provides the ability to define mathematical expressions with numbers. There is no distinction between real or integer. For example:

```
decor myNumber is a number
```

CALIBAN provides the possibility to constrain the type number with a range. The value of the decoration must fall inside the range.

```
type negNumber is a number up to 0
type posNumber is a number from 0
```

```
type grade is a number from 0 up to 10
```

If the first number is greater than the second, an error should be signalled.

Booleans

A boolean is a special kind of variable which can be true or false. The boolean type cannot be constrained.

```
decor flag is a boolean
copy (5 < 9) to flag
```

Symbols

A symbol is equivalent to an identifier in classical programming languages. A decoration can hold a symbol, but beware that some operations may fail if the value of a such decoration is a reserved symbol.

The quote (') is used to evaluate symbols to themselves, otherwise they are considered as variables (or types) containing values and will evaluate to the value(s) they hold.

```
decor name is a symbol
copy 'toto to name
```

The following script will produce an error as for CALIBAN, the symbol "hi" is not a created or declared variable.

```
copy hi to symbol
error
```

The *null* symbol is a valid value for any type. null corresponds to the empty value. The *indef* symbol is returned by some function when the result is undefined.

Lists

It is possible to put multiple pieces of information into a single variable. List values are enclosed by within braces, {…}. Each item in the list is separated from the other with a comma and a space.

Some list values:

```
{0, "Hello", {A, c}}
{1, 2, "World", true}
{}
null
```

The empty list is noted {}. For example:

```
decor myList is a list
copy {1, 2, 3} to myList
```

It is possible to constraint the list so that a decoration will only accept item of certain types (or refuse certain types in the list). For example:

```
type numberList is a list of number
type butSymbolList is a list without symbol
decor myList is a list of number and text
copy {1, 4,"Hello"} to myList
```

Enumerations

An enumeration is defined by a list and some constraints. The following type accepts only one item of the list as value. An enumeration can be empty.

```
type tGN is one of {Masc, Fem, Neut}
decor d is a tGN
copy Masc to d
```

Any other value will produce an error.

The following type accepts any subset of the list as value. Its value must be a list (potentially empty).

```
type tNex is any of {A, B, C, D}
decor d is a tNex
copy {A, C, D} to d
```

The items of the list can be any kind of value.

Aggregates

When working with large lists - especially those with different types of data, it is difficult to keep track of the type of each piece of data. Aggregates are like records (or typed frames) in other classical languages. An aggregate can be empty.

An aggregate is like a list with one significant difference. Instead of using a list such as {22, "Derrick", "Schneider"} and trying to remember which item contains which piece of information, you could use an aggregate:

```
copy {age:22, firstName:"Derrick", lastName:"Schneider"} to person
type personInfo is a {age:number, firstName:text, lastName:text}
decor otherPerson is a personInfo
copy {age:45, firstName:"John", lastName:"Avanzi"} to person
```

The item *age:22* (for example) is called an "a feature". *age* is the name of the feature (or attribute) and *22* is the value of the feature.

The list should could be empty, then the decoration is said to be an empty aggregate. Each item is composed of two symbols separated by a colon (:). The second symbol, denoting a type name, can be replaced by a type expression or any type. For example:

```
type morePersonInfo is {age:number, name:{firstName:text,
                            lastName:text}}
type stillMoreInfo is {age:number, name:any type}
```

The type can be a type name or a type declaration, or any-expression evaluating to a type.

It is possible to constrain aggregates the same way enumerations are constrained. The first constraint (one of) allows to force the choice between one of the items:

```
type vehicle is a one of {car:text, plane:text, boat:text}
decor deuxCV is a vehicle
copy {car:"Citroën"} to deuxCV
```

The second constraint allows any subsets of the list to be an acceptable value:

```
type completeName is any of {firstName:text, lastName:text,
medName:text}
decor personOne  is a completeName
copy {firstName:"Juan", medName:"Sebastian"}
```

It is possible to modify only parts of an aggregated decoration. For that, CALIBAN provide an access to decoration components. Here are two equivalent forms:

```
copy "Mateus" to medName of personOne
copy "Mateus" to personOne.merName
```

An aggregate type can be closed. Then it is not possible to add a new feature to the decoration of this type.

```
type vehicle is a closed one of {car:text, plane:text, boat:text}
```

All or some features can declared as mandatory:

```
type rectangle-1 is a obligatory of {top:integer, left:integer,
                                     bottom:integer, right:integer}
type rectangle-1 is a of {obl top:integer, obl left:integer,
                          obl bottom:integer, obl right:integer}
```

The two above types are equivalent. *obl* is an abbreviation of *obligatory*.

Type manipulation

A type can be manipulated as a decoration and some functions are dedicated to this purposes. These functions are not detailled here.

Tell statement

It is possible to use the "tell" command to let CALIBAN know that the included statements are directed to one object, instead of specfying this object in each line of the script. The "tell" command binds a context to a set of statements.

For example, in the following (quite complicated) script

```
type person is any of { name:text,
                        address:text,
                        birthDate:{d:number, m:number, y:number} }
type car is any of {owner:person, brand:text}
decor myCar is a car
copy {owner:{name:"Mathieu"}}, brand:"Corvette"} to myCar
copy {owner:{name:"Mathieu"}} & myCar to myCar
copy {address:"somewhere"} to owner of myCar
copy {birthDate:null} & (owner of myCar) to owner of myCar
copy {d:04, m:06} to birthdate of owner of myCar
```

The four lines above can be replaced by:

```
tell myCar
  copy {owner:{name:"Mathieu"}} & myCar
  copy {address:"somewhere"} to owner
  copy {birthDate:null} & owner to owner
  copy {d:04, m:06} to birthdate of owner
end tell
```

Several tell commands can be embedded:

```
tell myCar
  copy {owner:{name:"Mathieu"}} & myCar
  tell owner
```

```
    copy {address:"somewhere"} & owner
    copy {birthDate:null}
    copy {d:04, m:06} to birthdate
  end tell
end tell
```

Special variables

CALIBAN provides special variables which it fills and allows you to use.

result

contains the result of the last command.

result-code

contains a complementary information for the result. Generally contains an error code, if the result is not valid.

return, space and tab

contain respectively the characters return, space and tabulation.

```
copy "hello" & space & "world!" to myString
```

Coercing variables

You can tell CALIBAN how to interpret one piece of data as a different type. For example, if you write:

```
1 & 2
```

you will get the list {1, 2}

If you want to get the string "12", we will have to write:

```
1 as text & 2 as text
```

The following table shows the different coercions that can be used:

Data Type	Coercion	Result
text	`3 as text`	`"3"`
	`"hello" as text`	`"hello"`
number	`"3.5" as number`	`3.5`
list	`3.5 as list`	`{3.5}`

Generally, you would not worry about coercing variable as CALIBAN handles it for you.

Operators

Each data type has its own operators (although some are common between different data types). What follows is a review of all existing operators and their functions. Some of these operators have already been informally introduced.

boolean

These are the classical boolean operators.

```
copy 4 to x
```

```
if not (x is 5) then commands
```

This script will evaluates to true and executes the commands

```
copy {1, 2, 3, 4} to myList
if (x starts with 1) and not (x contains 5) then
  add 5 to myList
  if the number of items in x > 3 then
    commands
  end if
end if
```

text

Operators for text are concatenations, length, etc. Only concatenation is described here.

- & simple concatenation

```
copy "This" & "is" & "a" & "word" to x
⇒ "Thisisaword"
```

- && concatenation with space

This operation is like the & operator with a space added between the two arguments.

```
copy "This" && "is" && "a" && "sentence" to x
⇒ "This is a sentence"
```

symbols

Symbols can be converted to texts and to numbers.

numbers

- +, -, ÷ and /, - (unary), mod, div are the basic operators for numbers

Operator	Description	Example
*	multiplication	`4 * 3 = 12`
+	addition	`4 + 3 = 7`
-	substraction	`4 - 3 = 1`
÷ or /	division	`4 / 4 = 1`
div	division withoutremainder	`10 div 3 = 3`
mod	division returning remainder	`10 div 3 = 1`
-	negation	`-4`

Lists

- concatenation of two lists: & or union or +

This operator returns the non discriminative union of two lists.

```
{A, B, C} & {1, 2}
{A, B, C} union {1, 2}
```

```
{A, B, C} + {1, 2}
⇒ {A, B, C, 1, 2}
```

```
{A, B, C} & {A, B, D}
⇒ {A, B, C, A, B, D}
```

For the non discriminative counterpart of union, it is possible to nest the expression inside an "ignoring" expression (look at the Considering and Ignoring section, for more details):

```
considering unicity
  {A, B, C} & {A, B, D}
end considering
⇒ {A, B, C, D}
```

Union works as well embedded lists:

```
{A, {B, C}} & {{B, {D, E, {F}}}}
⇒ {A, {B, C}, {B, {D, E, {F}}}}
```

As embedded list without repetition can be considered as trees, a special considering statement can be used:

```
considering tree
  {A, {B, C}} & {{B, {D, E, {F}}}}
end considering
⇒ {A, {B, C, {D, E, {F}}}}
```

The same tree considering apply for intersection, difference, add and remove.

- intersection: inter or /

This operator returns the intersection of two lists.

```
{1, 2, 3} inter {3, 4}
{1, 2, 3} / {3, 4}
⇒ {3}
```

- difference: diff or -

This operator returns the discriminative union minus the intersection.

```
{1, 2, 3} diff {3, 4}
{1, 2, 3} - {3, 4}
⇒ {1, 2, 4}
```

• add and remove

```
add 3 to {1, 2}
⇒ {1, 2, 3}
remove 3 from {1, 2, 3}
⇒ {1, 2}
remove 3 from {1, 2}
⇒ {1, 2}
```

For the last statement, the result is remained inchanged becaused the item to be removed does not belong to the list. There is an error code avalaible in the *result-code* variable.

Enumerations

Exclusive enumerations (defined with *one of*) support operators for the type of their items. It is possible to copy only item (not list) into exclusive enumerations. Examples:

```
decor name is one of {"bob", "tom", "jenny", "lolita"}
copy "tom" to name
⇒ "tom"
copy "lolita" to name
⇒ "lolita"
```

Non-exclusive enumerations (defined with *any of*) support operators for the type of their items and the list operators. Examples:

```
decor group is any of {"bob", "tom", "jenny", "lolita"}
copy {"tom", "jenny"} to group
copy {"lolita"} & group to group
⇒ {"tom", "jenny", "lolita"}
```

we can write the script's last line in a simpler way:

```
add "lolita" to group
⇒ {"tom", "jenny", "lolita"}
```

If an already existing item is added then the enumeration remains unchanged (which is not the case for standard lists). The order of the enumeration can be changed by sorts. So with the previous example, the following expression doesn't change the value of the decoration group (because in already in group):

```
copy {"tom"} & group to group
⇒ {"tom", "jenny", "lolita"}
```

Aggregates

Aggregate decorations (with one of) support operators for the type of their items. Moreover, for non-exclusive aggregate decoration list functions aresupported. Aggregated are always considered with unicity and as trees. Here are some examples:

```
{owner:{name:"Mathieu"}, age:25} & {brand:"Corvette"}
⇒ {owner:{name:"Mathieu"}, age:25, brand:"Corvette"}
{owner:{name:"Mathieu"}, age:25} & {age:22}
⇒ indef
```

```
{owner:{name:"Mathieu"}, age:25} / {age:22}
⇒ null
{owner:{name:"Mathieu"}, age:25} / {age:null}
⇒ {age:25}
```

```
{owner:{name:"Mathieu"}, age:25} - {age:null}
⇒ {owner:{name:"Mathieu"}}
{owner:{name:"Mathieu"}, age:25} - {age:22}
⇒ {owner:{name:"Mathieu"}, age:25}
```

```
{owner:{name:"Mathieu"}, age:25} - {owner:{name:"Mathieu"}, age:25}
⇒ null
```

Conditionals

Conditionals gives the ability of deciding between several commands to execute based on the evaluation of an expression. The expression evaluates to a boolean value. If it is true then commands will executed, otherwise (false) they will be skiped.

Its syntax is the following:

```
if test then
   commands
end if
```

For example:

```
copy 3 to x
if x is 3 then
   copy "inside then" to myDecor
end if
```

An else part can be added to the standard "if… then" conditional. If the test expression evaluates to false then the command of the "else" part will be executed.

```
if test then
   commands
else
   commands
end if
```

For example:

```
if x ≠ 3 then
   copy "in then" to decor
else
   copy "in else" to decor
```

Comparisons

Comparisons are expressions that return boolean values (true or false) and are generally used in the test part of conditionals.

To verify if a piece of data is equal to another, we can write two equivalent expressions using "=" or "is". For example:

```
if x = 3 then commands
if x is 3 then commands
```

To verify if two pieces of data are different, we can write two equivalent expressions using "≠" or "is not". For example:

```
if x ≠ 3 then commands
if x is not 3 then commands
```

Text

You may want to evaluate if the text meets certain criteria. It is possible to test if a text begins with, ends with or contains a particular string. It is also possible to compare strings according the standard alphanumerical order. So you can do:

```
if "The tao of Caliban" starts with "Ca" then commands
if "The tao of Caliban" ends with "ban" then commands
if "The tao of Caliban" contains "tao" then commands
if "The tao of Caliban" comes before "Zz" then commands
if "The tao of Caliban" comes after "Aa" then commands
```

Numbers

You can use: <, >, ≤ and ≥ and their "English like" respective commands "is less than", "is more than", is less or equal to" and "is greater or equal to".

Booleans

Booleans are limited to the "equal to" and "not equal to" comparisons (the operators are and, or, not, implies, …). The next expressions returns true:

```
copy false to theFlag
if theFlag is false then commands
if theFlag is not true then commands
```

Lists

With a list you can use "starts with", "ends with" and "contains" as just with text. All of the following expressions will evaluate to true:

```
if {"The", "tao", "of", "Caliban"} starts with "The" then commands
if {"The", "tao", "of", "Caliban"} ends with "Caliban" then
commands
if {"The", "tao", "of", "Caliban"} contains "tao" then commands
```

The following expression will evaluate to false:

```
if {"The", "tao", "of", "Caliban"} contains {"tao", "Caliban"}
then commands
```

because the two items "tao" and "Caliban" are not contiguous in the list. A considering case can change this default behavior.

Enumerations

Exclusive enumerations support comparison of the type of their values. Non exclusive enumerations supports list comparisons:

Aggregates

When comparing aggregates, both the label and the data with that label must be equal. The items must be included but must not be in the same order (as in list).

For example, the following script:

```
if {name:"John", age:23, job:dev} contains {job:dev, name:"John"}
then
   commands
```

```
end if
```

will returns true.

Considering and ignoring

It is possible, to override some aspect of how a comparison is conducted. Consider the following script:

```
if "Caliban" = "Caliban" then  commands
```

It will returns true as, by default, CALIBAN doesn't consider case. But you may want to consider case, and the script to returns false. It is possible with:

```
considering case
  if "Caliban" = "caliban" then  commands
end considering
```

This script will return false.

With considering it is possible to take into account some particularities that are applicable to comparisons. It is also possible to ignore some features that influence comparisons. This can be done with "ignoring":

```
ignoring white space
  if "C a l i b a n" = "Caliban" then commands
end ignoring
```

This script will return true.

The following table describes some of the terms applicable to considering and ignoring and their effect.

Term	Example	CALIBAN's default
case	"Caliban" vs "caliban"	ignored
white space	"Cali ban" vs "Caliban"	Considered
diacritical	"resume" vs "résumé"	Considered
hyphens	"half-hour" vs "halfhour"	Considered
punctuation	"Hi" vs "Hi!"	Considered

In the advanced features some particular terms to "ignoring" and "considering" are presented.

Repeat

Loops perform commands repeatedly. The basic syntax of the "repeat" loop is derived into several different kinds of loops.

Basic repeat loop

This loop simply goes on forever, until the script is interrupted. This simpliest loop form will be normally be used only during development sessions.

```
repeat
  …
end repeat
```

For example:

```
repeat
  copy 1 to counter
end repeat
```

Conditional loops

You may want to perform an action and repeat it if a certain condition is met. There are two forms of conditional loops with *until* and with *while*.

The until conditional loop repeats the action until the condition is met. For example:

```
repeat until myNumber ≥ 10
  …
  copy myMumber + 1 to myNumber
end repeat
```

The while conditional loop repeats the action while the condition is met. For example:

```
repeat while myNumber < 10
  …
  copy myMumber + 1 to myNumber
end repeat
```

Counting repeat loop

You may want to repeat some actions a certain number of times and then exit. It is possible to do that with conditional loop but it is much easier with counting loops. Counting can be with or without counter. For example:

```
repeat 10 times
  …
end repeat
```

In counting loops, you may want to use a counter. The increment is one.

```
repeat with i from 1 to 10
  …
end repeat
```

You can change the value of the increment. For example:

```
copy {} to x
repeat with i from 1 to 10 by 2
  copy x & i to x
end repeat
```

The value of the list x is {1, 3, 5, 7, 9} after the execution of this script.

Traversing a list

The last kind of loop allows you to perform an action for each element of a list (or non-exclusive enumeration, or a non-exclusive aggregate). For example:

```
copy "" to s
repeat with i in {"One","Two", "Three"}
  copy s && i to s
```

```
end repeat
```

The result of this script is the string "One Two Three".

Advanced scripting

This section presents some in-depth aspects of CALIBAN, which could prove very useful but can be ignored at first.

Subroutines

It is possible to factorize some code in subroutines.

```
on mySubroutine()
   commands
end mySubroutine
```

Just call subroutine inside scripts. Subroutines can have arguments.

```
on mySubroutine(x, y)
   commands
end mySubroutine
```

The value returned by a subroutine is either the last value contained by the special variable "it" in the subroutine (the last value computed in "commands" in the above example) or the expression following the operator "return".

```
on mySubroutine(x, y)
   commands
   return expression
   other commands
end mySubroutine
```

will return the evaluation of "expression" as value. The following script uses a subroutine:

```
copy {alpha, beta, gamma, espilon} to alphabet
inverseList(alphabet)
on inverseList(theList)
copy {} to x
   repeat with item in theList
     copy item & x into x
   end repeat
end inverseList
```

Scripts

Scripts are the basic objects of CALIBAN. They have several features:

- a name
- a caller wich contains a reference to objects which called the script.
- an owner containing the script. Generally the owner and the caller will be the same object. It is possible to call another script form a script.
- a content which is the text of the script. User can dynamically construct their scripts.

We have then a new special variables (me) and part slots of script (caller, owner and content).

```
"me" refers to  the script. For example:
```

```
copy 1 + x to x
if x < 10  then
  excute me
end if
```

If the variable x has a previous value, this script will execute itself until x ≥ 10. "caller" refers to the caller of the script. "owner" refers to the owner of the script. Depending of the caller and owner object, information can be asked. "contents" refer to text of the script. A new script can be dynamically created.

Environments

Each declared variable is placed in a script environment (by default it is an environment named world, world is a special variable). Other variables (declared on the fly) are not placed in the environnement but considered as local variables of the script. If the script uses a local variable which has the same name as a declared variable (the two symbols are equal) then the script will only refer to the local variable.

- world

world is the default environment. When a variable or a type is declared whitout specifying its environment, it is placed in world. Expressions involving variables (or types) whitout specifying their environment refer to variables (or types) of world. People who use only one environment can ignore completely this notion.

When you declare a decoration or a type you can specify to which environment it belongs:

```
decor myDecor of environment myEnv is a number
type myType of environment myEnv is any of {a, b, c}
```

When you write a command you should specify to which environment the variable belongs:

```
copy (myNum of env1) + (myNum of env2) to (myNum of env1)
```

You can use the *tell* command to factorize the writing:

```
tell environment myEnv
  decor myDecor is a number
  type myType is any of {a, b, c}
end tell
```

This second form is equivalent to the first.

As a matter of fact, environments behave exactly as aggregate decorations. Let us suppose we have two environments, and the user wants a variable of the first environment to give its value to the variable of the same name in the second environment. We could write:

```
copy env1 & env2 to env2
```

Here, the variables which are in *env1* but not in *env2* are added to *env2*. If we just want to update the variables which are common to *env1* and *env2*:

```
copy env2 & (env1 inter env2) to env2
```

Résumé étendu

Quelles techniques de Génie Logiciel doit-on et peut-on mettre en œuvre pour spécifier et implémenter des systèmes de Traitement Automatique des Langues Naturelles (TALN) en général, et de Traduction Automatique (TA) en particulier ?

Le passé déjà long du TALN et de la TA permet de prendre du recul sur les Langages Spécialisés pour la Programmation Linguistique (LSPL). Il ne s'agit pas ici de savoir si leur usage se justifie, ni d'en inventer des types radicalement nouveaux, mais plutôt de déterminer les approches logicielles les plus efficaces pour les développer et les faire évoluer. Partant des divers types d'objets manipulés par les LSPL existants, nous avons défini LEAF, un modèle à objets qui permet de rendre compte de l'architecture de la plupart des systèmes modernes de TALN. LEAF répartit les composants qui interviennent dans un système de TALN en trois grandes classes : les treillis servent de représentation à des informations géométriques, les décorations sont liées aux informations algébriques, et les moteurs modifient les treillis et les décorations.

En liaison avec des linguistes, nous avons fait plusieurs expériences avec des LSPL existants en les reprenant pour les améliorer, la plupart du temps en CLOS (Common Lisp Object System). Ces expériences montrent clairement que l'intégrabilité, la généricité et l'extensibilité sont trois propriétés essentielles, tant au niveau de la définition des LSPL que de leur implémentation.

I. L'intégrabilité est la possibilité d'ajouter facilement de nouveaux composants dans un système. ODILE a été notre première expérience de "carrossage générique". Il s'agissait d'intégrer dans un même logiciel deux composants existants — un lemmatiseur et un outil dictionnaire — qui n'avaient pas été développés dans cette perspective, et devaient être facilement remplaçables.

Dans un cadre plus général, la difficulté vient de l'importante variété et de la grande taille des composants que l'on peut souhaiter intégrer. Les nombreux

systèmes de TA existants illustrent les approches possibles. Nous avons approfondi le modèle de "tableau blanc" et obtenu ainsi une architecture qui semble bien résoudre les problème d'intégration. Ce modèle permet de définir un mode général de communication entre les composants. Nous raffinons notre tableau blanc en choisissant une structure de treillis LEAF appropriée comme support de l'information.

II. Une double réalisation (à base de classes et à base de prototypes) d'un Langage de représentation linguistique nous a permis d'approfondir l'analyse de la généricité dans les LSPL.

Les langages de représentation constituent une sous-classe des LSPL. Nous avons centré notre étude sur les formalismes de structures à attributs. Le plus souvent, il s'agit de traits booléens ou attributs simples. On trouve aussi des structures de traits complexes, typés ou non. Enfin, certains systèmes dont le cadre dépasse le TALN utilisent des prototypes.

Les frames, que nous considérons comme des sous-classes de prototypes, offrent une généricité bien adaptée à l'implémentation d'un système de décorations linguistiques. Cependant, il semble que les frames ne soient pas assez contraintes pour bien guider les développeurs linguistes. C'est pourquoi nous proposons une version étendue de la notion traditionnelle de "décoration", et l'implémentons dans le langage DÉCOR. En définissant les types de décoration comme des contraintes sur les valeurs des traits, nous retrouvons les avantages des langages de classes tout en préservant l'unicité de la représentation. Notre extension consiste à ajouter aux "valeurs immédiates" des "valeurs par référence" et des "valeurs par formule". DÉCOR offre aussi la dynamicité, c'est-à-dire la possibilité de changer les relations entre prototypes durant l'exécution, ce qui peut être très utile lors de la mise au point de "linguiciels".

Les LSPL doivent être génériques et implémentés à partir de composants, eux-mêmes génériques. Dans notre première implémentation de DÉCOR, nous avons utilisé directement CLOS, qui n'a pas paru très bien adapté. Dans la seconde, plus satisfaisante, nous avons introduit une couche de prototypes, eux-mêmes écrits en CLOS. Nous avons aussi envisagé une réalisation à base de prototypes, mais fondée sur le Metaobject Protocol (MOP) offert par CLOS, et avons évalué son effet sur la généricité.

III. Nous avons analysé la nature et les limites de l'extensibilité quand on veut l'appliquer aux LSPL au travers de la réingénierie de deux LSPL, ATEF et ROBRA. ATEF est un langage qui permet d'écrire des analyseurs morphologiques. Nous avons déterminé quelles fonctionnalités de l'ATEF existant pouvaient être étendues et généralisées avec profit. En découplant les informations linguistiques et le moteur, nous avons alors obtenu un nouvel outil dont les deux composants sont réellement modulaires. Les fonctions de contrôle du non-déterminisme peuvent maintenant être étendues grâce à un protocole extensible de "Piles à Filtres Dynamiques". La possibilité de produire différentes formes de sortie et d'en définir de nouvelles a ainsi été réalisée de manière particulièrement simple, grâce à un autre protocole extensible.

ROBRA est un langage qui permet d'écrire des systèmes transformationnels travaillant sur des arbres décorés. Nous avons réutilisé tout ce qui avait été fait pour ATEF, en profitant de la généricité du gestionnaire d'automates.

En “génie logiciel pour le génie linguiciel”, il faut non seulement offrir des “boîtes à outils”, mais aussi se donner les moyens de construire des “familles d’outils”, c’est-à-dire se doter de protocoles extensibles. Pour rendre des protocoles extensibles tout en continuant à les comprendre, nous avons dû les “stratifier”, c’est-à-dire les définir selon plusieurs niveaux d’utilisation. Cependant, même avec cette approche modulaire, on atteint vite une “barrière de complexité statique”. Les programmes obtenus ne sont pas plus lents, mais deviennent de plus en plus difficiles à comprendre, et donc à étendre dans le futur.

Au total, cette étude, fondée aussi bien sur des considérations théoriques que sur des implémentations complètes, permet de cerner plus précisément les possibilités et les limites de la quête de l’intégrabilité, de la généricité, et de l’extensibilité dans le “génie logiciel pour le génie linguiciel”.

Colophon

Ce document a été rédigé sur Apple Macintosh avec des versions successives du système d'exploitation (System 7, 7.1 et 7.5). Le corps de texte est en Garamond 12. Les exemples de programmes sont en Courier. Les notes de bas de page sont en Times Roman. Les gros titres sont en Helvetica.

Les figures qui présentent des hiérarchies de classes et de manière générale des graphes ont été générée à partir des outils Geta-grapher et Geta-browser créés par l'auteur. Le lecteur attentif s'en sera douté, ces outils sont génériques et extensibles, ce qui a permis de multiples variations sur l'aspect des graphiques.

La plupart des figures ont été imprimées sous forme de fichiers EPS, incluses dans le corps du texte puis réduites.

Résumé

Cette thèse concerne l'étude de différentes techniques modernes de génie logiciel qui peuvent être mises en œuvre pour développer des systèmes de Traitement Automatique des Langues Naturelles de façon générique et extensible.

La première partie fait le point sur l'état de l'art en TALN à propos des Langages Spécialisés pour la Programmation Linguistique et permet d'identifier l'intégrabilité, l'extensibilité et la généricité comme trois qualités qu'il est souhaitable de fournir aux systèmes de TALN.

La définition d'un modèle à objets (LEAF) et l'affinage d'un modèle d'architecture (tableau blanc) constituent deux premiers éléments de réponse au problème de l'intégration. Une première approche de la généricité et de l'extensibilité est également présentée avec une expérience de réingéniérie du langage LT.

La seconde partie approfondit les problèmes de généricité et les illustre avec la définition d'un langage original de représentation linguistique (DECOR). L'introduction de protocoles internes et externes permet de rendre ce langage particulièrement générique et dynamique.

La troisième partie fait état de la réingériérie de deux langages spécialisés (ATEF et ROBRA). Cette expérience permet d'introduire une grande extensibilité dans les moteurs de ces langages par une programmation par objets et protocoles.

Une question transverse à ces trois parties concerne l'identification des limites et des écueils liés à la recherche de l'intégrabilité, de l'extensibilité et de la généricité. Ces difficultés viennent principalement de la complexité croissante des protocoles adéquats, dont la maîtrise pourrait rapidement échapper au développeur si l'on n'y prend garde.

Mots-clés

Génie Logiciel, Génie Linguiciel, Langages Spécialisés pour la Programmation Linguistique, Protocoles, Programmation par objets, Intégrabilité, Généricité, Extensibilité, Lisp, CLOS.

Abstract

The aim of this thesis is the study of various modern engineering techniques which can be used for the development of Natural Language Processing systems.

The first part presents the state of the art in NLP concerning Specialised Languages for Linguistic Programming. We identify integrability, genericity and extensibility as the most crucial properties of NLP systems.

The definition of an object model (LEAF) and the enhancement of an architural model (WhiteBoard) are the first two answers to integrability. A first approach to genericity and extensibility is also provided by a reengineering experiment of the LT language.

The second part gives a thorough account of the problems linked to genericity and exemplifies them with the definition of an original linguistic representation language (DECOR). This language can be made both generic and dynamic by the introduction of internal and external protocols.

The third part concerns the reengineering of two specialized langages (ATEF and ROBRA). This experiment allows us to introduce a good extensibility in the engines of these languages, using object and protocol oriented programming techniques.

A common question to these three parts comes up with the identification of the limits and difficulties concerning the journey to integrability, genericity and extensibility. These obstacles come mainly from of the increasing complexity of the appropriate protocols, which may easily cause the developer to lose control over the implementation.

Keywords

Software Engineering, Lingware Engineering, Specialized Languages for Linguistic Programming, Protocols, Object Oriented Programming, Integrability, Genericity, Extensibility, Lisp, CLOS.