Université de Nancy I

Centre de Recherche en
Informatique de Nancy

Prototypage, programmation objet : application à l'édition structurelle

THÈSE

présentée et soutenue publiquement le **3 Février 1989**

pour l'obtention du **doctorat de l'Université de Nancy I**
(Mention Informatique)

par

Dominique COLNET

Composition du jury :

Président :	Jean-Pierre FINANCE
Rapporteurs :	Jean BEZIVIN
	Claude KIRCHNER
Examinateurs :	Philippe JAMART
	Jean-Claude DERNIAME

Université de Nancy I

Centre de Recherche en
Informatique de Nancy

Prototypage, programmation objet : application à l'édition structurelle

THÈSE

présentée et soutenue publiquement le **3 Février 1989**

pour l'obtention du **doctorat de l'Université de Nancy I**
(Mention Informatique)

par

Dominique COLNET

Composition du jury :

Président :	Jean-Pierre FINANCE
Rapporteurs :	Jean BEZIVIN Claude KIRCHNER
Examinateurs :	Philippe JAMART Jean-Claude DERNIAME

Prototypage, programmation objet : application à l'édition structurelle

Dominique COLNET

Vendredi 3 février 1989

Avant-propos

Cette thèse rend compte de quatre années d'activité au Centre de Recherche en Informatique de Nancy, au sein de l'équipe Génie Logiciel dirigée par Monsieur le professeur Jean Claude DERNIAME.

Dès le début de ces quatre années, mon travail de thèse a été orienté vers la conception d'environnements de programmation et plus particulièrement celle d'éditeurs syntaxiques paramétrés avec la description d'un langage. Au fur et à mesure de l'avancement du travail, la complexité du problème, tant au niveau de la définition des fonctionnalités que de l'implantation, nous a poussé à explorer les techniques modernes de conception de logiciels.

A peine terminé, le premier prototype du générateur d'éditeur syntaxique, portant le nom insolite d'*arthur*, fut remis en cause : ce n'était qu'un éditeur syntaxique ordinaire, en un mot, trop dirigiste. De prototypes en prototypes, les fonctionnalités ainsi que le langage de description de langage ont évolués progressivement. La version *marcel* dont je tiens à graver ici le nom fut celle qui précéda la version actuelle présentée dans cette thèse, *gépi*.

La démarche adoptée, qui consiste à construire un logiciel pour l'évaluer et le critiquer, peut paraître désastreuse, voire masochiste. En effet, de nombreuses lignes de code furent écrites inutilement, nos yeux s'épuisant sur le *vert alphanumérique* d'antiques consoles. Nous ne faisions qu'appliquer à la lettre le conseil *"n'hésitez pas à tout recommencer"*, inscrit sur la dernière page de l'excellent livre de Jacques Arsac [Arsac 77]. Ce conseil, loin d'être périmé, nous a naturellement orienté vers les techniques de programmation objet qui, du fait de leur souplesse, permettent justement de tout recommencer sans tout jeter.

C'est à la suite d'informelles réunions de travail au sujet de CEYX [Hullot 83b], que notre intérêt s'est développé pour les langages à objets. Non moins informellement, le groupe langages à objets du CRIN venait de naître. Les travaux menés dans le cadre de ce groupe m'ont temporairement écarté de mon objectif initial, l'édition syntaxique. En retour, ils m'ont permis d'accroître mes connaissances dans le domaine de la programmation objet, de confronter des idées, de rencontrer

des personnalités, et surtout, d'apprendre à travailler.

L'objectif de cette thèse consiste naturellement à présenter le résultat d'un travail, en l'occurrence le manipulateur de documents du projet GÉPI. En exhibant son implantation, nous espérons, si ce n'est déjà fait, faire découvrir aux lecteurs les joies de la programmation objet.

Remerciements

Je remercie Jean-Pierre FINANCE, professeur à l'université de Nancy I, directeur du Centre de Recherche en Informatique de Nancy, président du jury de cette thèse.

Je remercie Jean BEZIVIN, professeur à l'université de Brest, rapporteur de cette thèse, pour l'attention qu'il a portée à mon travail, pour ses remarques pertinentes, et pour s'être déplacé jusqu'à Nancy.

Je remercie Claude KIRCHNER, directeur de recherches à l'INRIA, rapporteur de cette thèse, pour l'intérêt qu'il a porté à mon travail.

Je remercie Philippe JAMART, professeur à l'université catholique de Louvain de s'être déplacé de Belgique pour participer à ce jury.

Je remercie bien sûr Jean-Claude DERNIAME, professeur à l'université de Nancy I, pour son accueil dans l'équipe Génie Logiciel, pour ses encouragements, pour le suivi de mon travail, et pour l'excellente ambiance qu'il sait entretenir au sein de son équipe.

Je remercie vivement les autres membres du projet GÉPI : Gérôme CANALS, actuellement coopérant à Yaoundé, pour les heures passées ensemble à réfléchir et à déverminer les prototypes et pour avoir supporté sans broncher les normes et le style de programmation qui me plaisent. Pour les mêmes raisons et pour la couleur mexicaine qu'il a donnée au bureau que nous occupons ensemble depuis quatre ans, Samuel CRUZLARA Da SILVA.
En espérant que le projet continuera.

Je remercie tout particulièrement les membres du groupe langages à objets du

CRIN : Daniel LEONARD, Gérald MASINI, Amédéo NAPOLI. et Karl TOMBRE, pour les conversations aussi animées furent-elles, pour la formation qu'ils m'ont apportée, pour la rédaction du chapitre 4 à laquelle ils ont tous plus ou moins participé et qui fera partie de l'ouvrage que nous allons produire ensemble [Masini 89]. Je n'oublie pas Yvan NOIRET, qui fait partie des auteurs de l'ouvrage initial [Colnet 86a] et qui est à l'origine de mon intérêt pour la programmation objet.

Je remercie vivement les lecteurs laborieux et courageux qui ont bien voulu relire et commenter soigneusement plusieurs chapitres ou même l'intégralité de cette thèse : Anne BOYER, François CHAROY, Christine FAY-VARNIER, Isabelle GNAEDIG, Bruno LANDI, Azim ROUSSANALY, Monique de SILVESTRI.

Mes parents, pour m'avoir mis au monde, élevé, nourri et, vingt-huit ans plus tard, pour avoir bien voulu corriger les fautes d'orthographe de leur deuxième rejeton : Pierre COLNET et Suzanne COLNET née DROUVOT.

Mon professeur d'Aïkido, Yvan LOSTETTE, pour ses cours inoubliables et pour m'avoir incité à quitter le tapis plus que de raison afin de mettre un terme à cette thèse.

Danielle MARCHAND, secrétaire efficace et dévouée à l'équipe Génie Logiciel.

Enfin, mes amis, camarades, copains, copines, rencontres qui, même s'ils pensent qu'ils n'ont rien à voir avec la rédaction de cette thèse, ont peut être, soit souffert soit profité de mon absence temporaire. Dans l'ordre aléatoire : paul, marguerite, simone, bertrand, dominique, doudou, brigitte, jeannot, niquette, annick, véronique, zouzou, denis, kiki, marie-annick, bruno, gina, cindy, nicolas, tintin, gratounette, léon, daniel, robert, frennsouah, pierre, aligator, ali, pappy, josianne, christine, bernard, jeanine, bertrichou, fabien, targette, jean-luc, coincoin, myriam, françoise, dédé, agnès, gérard, cathy, legrand, michel, pierrot, hélène, fabrice, christiane, françois, foufoune, thierry, michèle, pascal, laurent, chantal, marie, odile, marie-odile, doms, salva, didier, karl, véro, chico, françois, yvan, aïki, christian, tissier, marlène, karol, doris, patricia, luc, titi, jean, charles, azoum, jules, clods, claire, christophe, sylvain, christine, mazou, gérald, fabienne, tgv, éric, bill, sophie, sonia, mob, jean-claude, toinet, choupette, éric, monique, marcel, claudine, vincent, gino, nicole, sam, vévète, samuel, yvette, hassan, gérôme, paulette, jean-charles, amédéo, napps, claude, isabelle, maillardos, jérémy, cabichou, béchir, jeff, noëlle boby,

danièle, jean-pierre, khalid, kim, cendra, richard, martine, martoche, anne, yifan, gosub, gilles, joachim, dalila, joëlle, jean-marie, valérie, abderrafia, jean-charles, danielle, tahar, roger, nadine, michaël, anas, norbert, yves, jean-jacques, marc, maryse, francis, philipe, nacer, alain, henri, stephan, GÉPI, Macintosh, LaTeX et café.

Que tous en soient remerciés.
Nancy, le 3 février 1989
Dominique COLNET
loulou

Résumé

Cette thèse s'inscrit dans le cadre du projet GÉPI : un Générateur d'Environnements de Programmation Intégrés. Elle couvre deux aspects du projet : d'une part la définition du langage de description de langages et d'autre part la description du manipulateur de documents. En outre, elle contient la définition du langage à objets leloup, défini pour le projet. Ce langage s'apparente à Smalltalk, il est utilisé pour décrire l'implantation du manipulateur de documents.

Cette thèse peut être lue sous deux angles différents selon que l'on s'intéresse à l'édition syntaxique d'une part ou à la programmation objet d'autre part.

Première partie

La première partie de cette thèse commence par un survol des différentes caractéristiques de GÉPI. L'interface, un des points forts de GÉPI, est présentée en détails (cf. chap. 1). Ce chapitre contient également les principales références bibliographiques sur les systèmes existants et permet de situer GÉPI par rapport à des systèmes comme MENTOR, CPS et Cépage. Pour terminer cette première partie, le langage de description de langages est présenté (cf. chap. 2).

Deuxième partie

La deuxième partie est essentiellement consacrée à la programmation objets et commence par une présentation des concepts fondamentaux de ce style de programmation (cf. chap. 3). Dans cette partie, Smalltalk est utilisé comme langage de référence. Ce langage est ensuite comparé à un langage plus classique, Ada (cf. chap. 4). Ce chapitre nous donne aussi l'occasion de présenter le modèle MVC, un outil puissant du langage Smalltalk pour le prototypage d'applications interactives. La deuxième partie se termine par la présentation de leloup (cf. chap. 5), le langage qui a servi à l'implantation de GÉPI.

Troisième partie

Cette partie est consacrée à l'implantation du manipulateur de documents de GÉPI. L'implantation est décrite en leloup et peut être vue comme un exercice *grandeur nature* d'utilisation d'un langage à objets. Elle commence par la description de l'implantation du couplage entre le texte d'un document et son arbre représentatif (cf. chap. 6). L'exercice continue avec la description du décompilateur et de l'analyseur syntaxique (cf. chap. 7).

Bilan

La dernière partie (cf. chap. 7) fait le bilan du projet dans son ensemble, aussi bien en ce qui concerne le générateur lui même qu'en ce qui concerne la qualité de son langage d'implantation, leloup, très proche de Smalltalk.

Le glossaire (page 213) et l'index (page 234) permettent un accès direct aux principaux concepts et termes utilisés dans cette thèse.

Table des matières

1
Le projet GÉPI

" By the term "software engineering" we mean the body of theory and practical techniques that can be brought to bear on the process of developing software. "

Doug Bell, Ian Morrey, John Pugh [Bell 87]

GÉPI est un Générateur d'Environnements de Programmation Intégrés qui prend en paramètre la description d'un langage et qui produit un environnement de développement comprenant un éditeur syntaxique, un éditeur d'arbres, un paragrapheur, un analyseur syntaxique ainsi que des outils de vérifications sémantiques adaptés à ce langage.

Chaque environnement est pourvu d'un manipulateur de documents spécialisé dans le traitement du langage associé à l'environnement. Un accent tout particulier a été mis sur la réalisation du manipulateur de documents. La définition de ses fonctionnalités d'une part, et la méthode utilisée pour sa réalisation d'autre part, constituent les deux axes importants de cette thèse.

Plan

Après avoir passé en revue les motivations qui sont à l'origine du projet GÉPI (§1.1), la pièce essentielle du générateur, le manipulateur de documents, est présentée en détails (§1.2). La partie suivante décrit le procédé de définition des outils de vérification sémantique (§1.3). Ce procédé ayant fait l'objet d'une autre thèse [Silva 88], seules les grandes lignes en sont présentées afin de donner une vue d'ensemble du projet. Avant d'effectuer un premier bilan (§1.5), la méthode qui a guidé la définition de l'interface du manipulateur de document est présentée (§1.4).

1.1 Motivations

La définition d'environnements de programmation représente une part importante dans l'éventail d'applications que l'on peut ranger sous l'étiquette *génie logiciel*. Le cadre associé au projet *gépi* est limité à la génération d'environnements ne prenant pas en compte une méthode de développement particulière. Par opposition avec les environnements *orientés méthodes* comme Maiday [Jacquot 84] [Guyard 84] ou ASSPEGIQUE [Bidoit 84] par exemple, GÉPI est dit *orienté langage*.

Un environnement de programmation intégré est constitué d'un ensemble d'outils travaillant simultanément sur un document, tout au long de sa mise en forme. Les outils collaborent et coopèrent en établissant des relations entre eux [Silva 88]. Ils sont spécifiques au langage traité et peuvent donc tenir compte de caractéristiques propres à ce langage. La description du langage à manipuler constitue le paramètre essentiel de GÉPI [Canals 88].

1.1.1 Paramétrer avec le langage

Paramétrer le générateur d'environnement avec la description du langage présente des avantages, mais aussi des inconvénients.

Des avantages.

Les avantages du paramétrage sont évidents. En premier lieu, le fait de disposer d'un générateur évite de recommencer plusieurs fois le même travail. Cette technique, dans le domaine de l'analyse syntaxique en particulier, a donné naissance à de nombreux générateurs d'analyseurs syntaxiques dont YACC est l'exemple le plus célèbre [Aho 74] [Berry 84].

Le deuxième avantage est lié au fait que le développement de logiciel est une activité qui va de pair avec l'utilisation de plusieurs langages : langage de spécification, langage de programmation, langage de commande...
L'utilisation d'un simple formateur de texte comme `nroff` [Ossanna 86] ou comme LaTeX[Lamport 86] pour la rédaction du cahier des charges et de la documentation associée au logiciel impose l'apprentissage d'un langage particulier. De même, sur certains systèmes d'exploitation, l'utilisation d'un éditeur de liens passe par l'apprentissage d'un véritable petit langage. Le plus fameux des systèmes actuels, UNIX [Bourne 85] pour ne pas le citer, recèle, sans compter les langages de programmation, une quantité de langages couramment utilisés : plusieurs interprètes de commandes sont disponibles[1] ainsi que différents filtres pour flux de caractères[2]...
En outre, la syntaxe de ces langages est souvent sensible[3], loin d'être orthogonale [Bell 87, page 88].

De fait, une part importante du temps de développement d'un logiciel est consacrée à la consultation des notices des différents langages utilisés et, souvent, le temps se perd en correction d'erreurs de syntaxe. Dans ce contexte, inutile de préciser qu'un éditeur syntaxique adaptable à toutes ces catégories de langages est un outil précieux.

Des inconvénients

L'approche consistant à réaliser un générateur d'environnements paramétré présente plusieurs inconvénients. Paradoxalement, le premier inconvénient est de ne pas pouvoir paramétrer suffisamment le générateur, en conséquence de quoi, la qualité des environnements engendrés, comparée à celle des environnements spécifiques à un langage, n'est que médiocre. Pour éviter cela, le langage de description de langage, paramètre essentiel du générateur, doit être soigneusement étudié. Il doit prendre en compte toutes les caractéristiques du langage à décrire tout en restant simple.

Les caractéristiques à décrire sont nombreuses, citons par exemple la syntaxe, la lexicographie, la façon d'écrire les commentaires ou les éventuelles options de compilation. Il faut aussi être capable d'exprimer simplement le paragraphage ou le fait que les mots clés du langage sont réservés par exemple. Dans le même ordre d'idées, il faut pouvoir distinguer le fait qu'un terminal du langage est ou non un terminal séparateur : la génération de l'analyseur syntaxique en dépend. En outre, la description du langage doit comprendre la description de sa syntaxe abstraite afin d'être en mesure de construire l'arbre représentatif d'un document, indispensable pour l'éditeur syntaxique.

La description du langage, tout en intégrant les différents aspects mis en évidence précédemment doit rester la plus simple possible. En particulier, les redondances d'informations doivent être évitées. Par exemple, dans un système comme MENTOR, la description de la syntaxe abstraite (structure de l'arbre) est séparée de

1. `sh`, `csh`, `tcsh`, `xsh`, `nsh` pour ne citer que quelques interprètes de commandes disponibles sur le Vax du CRIN.
2. Voici quelques noms qui feront plaisir aux *hackers*, futurs nostalgiques du système UNIX : `lex`, `yacc`, `awk`, `sed`, `grep`, `fgrep`, `egrep` `ed` `ex` ...
3. Les *hackers* UNIX savent tous qu'il ne faut pas confondre une tabulation avec plusieurs caractères blancs dans un `Makefile`.

la description de la syntaxe concrète (syntaxe du document). Deux langages différents sont utilisés pour décrire ces deux syntaxes intimement liées : les redondances d'informations sont nombreuses et exprimées dans un formalisme différent.

Le deuxième inconvénient d'un générateur paramétré réside dans sa complexité de réalisation même. Les problèmes à résoudre sont nombreux et concernent simultanément ceux de la compilation, de la description de langages, de la description d'outils, de l'édition, de la gestion d'écran ou de l'ergonomie. Heureusement, le domaine de la génération d'environnements orientés langages a déjà fait l'objet de nombreux travaux : MENTOR [DonzeauGouge 80], CPS [Reps 83a], CONCERTO [Andre 86], GIPE [Clement 86]. Néanmoins, tous les problèmes ne sont pas résolus, comme par exemple celui de l'édition syntaxique.

1.1.2 Edition syntaxique

Les premières références bibliographiques concernant l'édition syntaxique ont maintenant plus de quinze ans [DonzeauGouge 75]. Des éditeurs syntaxiques paramétrables avec la description d'un langage sont dès à présent disponibles sur le marché. Voici quelques références parmi les systèmes actuels les plus connus. Ces éditeurs servent dans la suite de base de comparaison :

- la dernière version du système MENTOR [DonzeauGouge 84] et son successeur CENTAUR [Clement 86] développé dans le cadre du projet Esprit GIPE, *Generation of Interactive Programming Environments*,
- le célèbre CPS, *Cornell Program Synthesizer* [Reps 81] [Reps 83a] [Reps 83b] [Reps 84] [Reps 85],
- ALOE, *A Langage Oriented Editor*, [MedinaMora 81],
- et enfin Cépage [Meyer 86a] [Meyer 87c] [Meyer 87a], l'éditeur structurel livré avec le langage Eiffel [Meyer 86c].

Depuis les premières versions de MENTOR [DonzeauGouge 75], de nombreux progrès ergonomiques ont été effectués [DonzeauGouge 80]. Grâce à la croissance des puissances de calcul, de mémorisation, des facultés d'affichages ainsi qu'avec l'apparition de dispositifs de pointage comme la souris, les éditeurs syntaxiques actuels sont d'une utilisation plus agréable.
Pourtant, mis à part certains éditeurs syntaxiques spécialisés dans le traitement d'un langage particulier [Greenblatt 84], les éditeurs syntaxiques sont, à notre avis, trop rarement utilisés. Les éditeurs pleine page, non syntaxiques, comme Emacs [Stallman 86] ou vi [Kernighan 81] par exemple, restent les plus utilisés.

Edition structurelle – Edition textuelle

La vocation principale des éditeurs syntaxiques cités précédemment est d'assurer en permanence la correction d'un document vis-à-vis d'une syntaxe définie au préalable. Le programmeur peut ainsi se débarrasser des problèmes de détail pour se concentrer sur les tâches plus importantes de son activité.

Ces outils construisent un document par raffinements successifs. A partir de l'axiome, on étend petit à petit le texte en dérivant les règles de la grammaire qui décrit le langage. Le programmeur se contente de choisir parmi les alternatives celles qui lui conviennent et le système se charge de construire le texte correspondant. Cette idée très séduisante se révèle particulièrement intéressante pour un programmeur débutant dans l'utilisation d'un langage ou pour la mise en place de structures syntaxiques complexes comme les structures de contrôle par exemple. Par contre, dès qu'il s'agit de programmeurs expérimentés, ou d'instructions plus simples comme les expressions ou les identificateurs, ce mode de construction se révèle extrêmement rigide et fastidieux.

La tentation et l'habitude de frapper du texte *au kilomètre* sont trop fortes pour faire accepter ce type d'outil. Pour remédier à ce problème, un premier pas consiste à adjoindre à l'éditeur structurel un analyseur syntaxique, ce qui permet de taper du texte et de l'analyser ensuite [Ebel 83]. Cette solution est adoptée par la plupart des éditeurs syntaxiques actuels comme CPS ou Cépage par exemple. Cependant, il semble que leur utilisation ne fasse pas l'unanimité ou, tout du moins ne soit pas encore passée dans les mœurs. La raison essentielle de cet état de fait s'explique à notre avis par leur manque de souplesse.

L'idée à la base des éditeurs syntaxiques classiques est d'assurer en permanence la correction syntaxique du document. Si on examine les méthodes de travail d'un programmeur expérimenté, on s'aperçoit qu'il peut par moment tirer profit d'un tel outil, pour vérifier un texte ou corriger des erreurs par exemple, mais aussi qu'il lui arrive très souvent de manipuler des structures syntaxiques incomplètes ou incorrectes : fragments de programmes, morceaux d'instructions ou d'identificateurs. Plus simplement, s'il lui arrive parfois de raisonner en termes de structures syntaxiques, il raisonne également en termes textuels : construire un programme ne consiste pas seulement à assembler des structures, il s'agit avant tout de construire un texte.

Il nous semble donc primordial que, pour être utilisé avec souplesse et facilité, un éditeur de documents puisse à la fois tirer parti de la description syntaxique du langage qu'il manipule, sans pour autant abandonner les commandes de type textuel, c'est-à-dire qui agissent sur le texte du document et non pas uniquement

sur sa structure.

Intérêts de l'édition syntaxique

Les avantages de l'édition syntaxique sont nombreux. En premier lieu, l'utilisateur n'a pas besoin de connaître la syntaxe du langage qu'il utilise. Il peut directement écrire un texte dans ce langage à condition bien sûr d'en connaître les concepts. Par exemple, le concept de boucle est similaire dans la plupart des langages algorithmiques. Un programmeur Pascal [Wirth 71b] peut *a priori* écrire directement un programme en langage C [Kernighan 78], l'éditeur servant de guide pour la mise en place des structures syntaxiques. Bien sûr, il n'en est pas de même lorsque l'on ne connaît pas les concepts du nouveau langage. Le passage de Pascal à Prolog [Clocksin 87] pose des problèmes plus complexes que celui de la syntaxe.

Dans le cas où l'on connaît globalement les concepts du langage que l'on utilise, l'éditeur syntaxique peut être un outil d'apprentissage rapide. Les différentes constructions sont proposées interactivement et peuvent être accompagnées d'une documentation spécifique. Par l'intermédiaire de la syntaxe et à l'aide de la documentation spécifique, l'utilisateur découvre les particularités du langage qu'il utilise. Lors de chaque raffinement, toutes les possibilités sont proposées à l'utilisateur. La lecture complète du manuel du langage peut ainsi être évitée.

Un éditeur syntaxique paramétrable avec la description du langage peut facilement permettre la mise en place de normes de programmation. Par exemple, il est possible d'éviter l'utilisation d'une structure de contrôle jugée dangereuse en la suprimant de la description du langage. Dans le même ordre d'idées, des commentaires au format du projet ou de l'équipe qui utilise l'éditeur peuvent être insérés systématiquement en entête des programmes.

1.1.3 Objectifs

Le désintérêt des utilisateurs pour l'édition syntaxique est, à notre avis, uniquement lié à leur manque de convivialité : la manipulation du document est la plupart du temps trop dirigée et les opérations habituellement simples deviennent compliquées voire impossibles. A titre d'exemple, dans un document manipulé avec Cépage, il n'est possible de mettre des commentaires qu'à certains endroits bien précis du document. Comme CPS ou MENTOR, Cépage impose continuellement la correction syntaxique. Chaque phase d'édition est accompagnée d'une vérification. Une partie de document modifiée est vérifiée sur le champ et, tant que cette partie n'est pas correcte syntaxiquement, il n'est pas possible de travailler sur une autre

partie du document.

Notre premier objectif consiste à réaliser un éditeur structurel capable d'intégrer les possibilités de manipulation textuelle des meilleurs éditeurs pleine page en conservant tous les avantages de l'édition syntaxique.

Notre deuxième objectif concerne la description du langage. Pour prendre en compte les spécificités d'un langage comme Ada, C ou LaTeX, l'écriture d'une simple grammaire à contexte libre est insuffisante. Des concepts moins formels comme l'écriture des commentaires, la notion de séparateur ou la mise en page du document doivent être décrits. En outre, la description d'une syntaxe abstraite décrivant la structure de l'arbre que l'on associe à un document est nécessaire dans un cadre d'édition syntaxique. L'objectif du langage de description de langage que nous proposons (cf. chap. 2) est d'intégrer dans un seul document tous les aspects d'une description de langage : syntaxe concrète, syntaxe abstraite, lexicographie, paragraphage...
Ce langage se veut suffisamment simple afin d'être manipulable par l'utilisateur qui pourra ainsi compléter lui même sa bibliothèque de descriptions de langages.

1.2 Le manipulateur de documents

Les stations de travail modernes offrent maintenant la possibilité de soigner particulièrement la convivialité et la souplesse de l'interface externe d'un environnement de programmation. Celle de GÉPI est principalement constituée du manipulateur de documents : c'est lui qui interagit avec l'utilisateur et qui permet la construction et la modification des documents.

Le manipulateur de documents allie la puissance d'un éditeur syntaxique pour la vérification et la correction d'erreurs, et la souplesse d'utilisation d'un éditeur de texte classique comme Emacs par exemple. Il s'agit en fait d'un éditeur structurel basé sur la description syntaxique du langage utilisé mais permettant cependant la manipulation de documents syntaxiquement incorrects.

1.2.1 La représentation arborescente du document

Le principe de base d'un éditeur syntaxique consiste à représenter un document de façon arborescente. Le manipulateur de document de GÉPI n'échappe pas à cette règle : un *arbre abstrait* [4] est construit parallèlement à la construction du texte du document. Cet arbre permet de représenter la structure syntaxique du document.

4. Nous préférons utiliser un arbre abstrait plutôt qu'un arbre de dérivation, plus encombrant et moins intuitif. Les termes arbre abstrait et arbre de dérivation sont définis dans le glossaire.

Comme pour MENTOR, Cépage ou CPS, c'est grâce à cet arbre que les commandes syntaxiques peuvent être réalisées. Cependant, afin de permettre une manipulation plus souple du document, l'arbre représentatif du manipulateur de documents de GÉPI n'est pas contraint à respecter strictement les règles de syntaxe et certaines feuilles de l'arbre peuvent contenir du texte non encore syntaxiquement validé. Par exemple, lorsque l'utilisateur entre du texte *au kilomètre* pour construire le corps d'une procédure, la représentation arborescente de ce corps de procédure est une feuille contenant le texte entré manuellement.

La présence de telles feuilles n'empêche pas l'utilisateur de manipuler les autres parties de l'arbre et, pourquoi pas, de provoquer l'apparition de nouvelles feuilles *texte*. A l'extrême, l'arbre représentatif d'un document entier peut se limiter à une unique feuille contenant tout le texte du programme. Dans ce cas, toutes les manipulations effectuables par l'utilisateur sont d'ordre textuel. A la demande de l'utilisateur, il est possible de déclencher une analyse syntaxique dont le but est de reconstruire soit partiellement soit complètement l'arbre représentatif.

1.2.2 L'interface du manipulateur de documents

L'interface proposée par l'éditeur de GÉPI est une interface hautement interactive, utilisant le multifenêtrage et un dispositif de pointage. La communication se fait au travers d'une souris, par menus ou au clavier. Sur l'écran, un manipulateur de document se présente sous la forme d'une fenêtre principale composée de sous-fenêtres et de boutons.

L'image d'écran de la figure 1.1 montre un manipulateur de documents utilisant la description du langage miniPascal (cf. § 2.4). La fenêtre de gauche est un éditeur pleine page disposant de fonctionnalités comparables à celle d'Emacs. Dans cette fenêtre, la représentation textuelle du document peut être manipulée à l'aide du clavier ou de la souris. Les commandes habituelles d'édition de textes sont disponibles : déplacements, frappe de caractères au kilomètre, substitutions, effacements, recherches de sous-chaînes, désignation d'une portion de texte...

La fenêtre de droite, au dessus de celle des boutons, est un éditeur d'arbre qui permet de manipuler la représentation arborescente du document à l'aide de la souris. Pour correspondre à l'ordre de lecture du texte, l'arbre est représenté couché, la racine est à gauche et les fils sont affichés dans l'ordre, de haut en bas.

A chaque nœud de l'arbre représentatif du document correspond une zone d'influence dans le texte, la zone d'influence d'un nœud englobant toujours celles de ses fils. La zone de texte associée à la racine de l'arbre, `<bloc.>`, correspond au texte tout entier. La zone de texte correspondant à la partie déclaration des

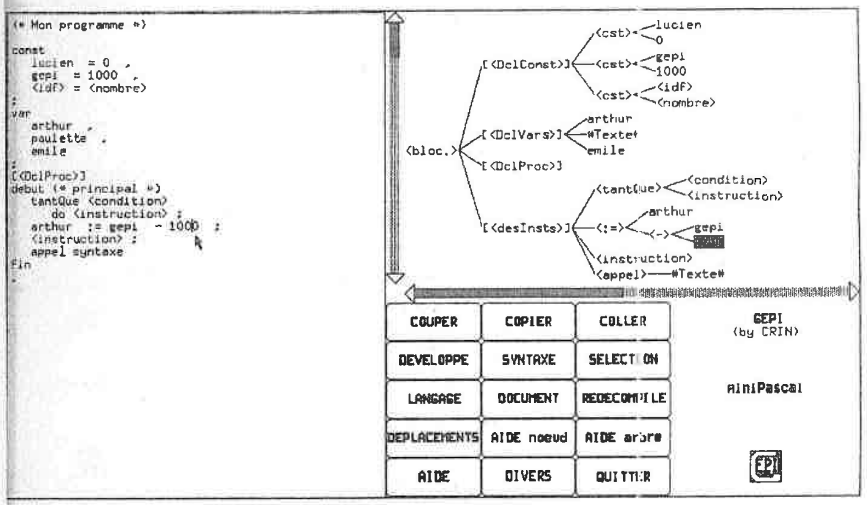


Figure 1.1. Un manipulateur de documents pour le langage miniPascal. La fenêtre de gauche est un éditeur pleine page qui permet de manipuler la représentation textuelle du document. Celle de droite est un éditeur d'arbres qui permet de manipuler la représentation arborescente du document.

constantes, portée par le nœud `[<DclConst>]`, commence à partir du mot clé `const` et se termine après le caractère point-virgule qui indique la fin de cette partie. Ainsi de suite, le texte est découpé en zones de plus en plus petites au fur et à mesure que l'on descend vers les feuilles de l'arbre représentatif. A la feuille `lucien` correspond le texte de l'identificateur de même nom dans le texte du document.

Comme c'est le cas pour l'ensemble des éditeurs syntaxiques, l'arbre peut être incomplètement développé, certaines parties n'étant pas complètement décrites ou *raffinées*. Dans l'exemple, la partie `[<DclProc>]` est réservée aux déclarations de procédures. Pour l'instant aucune procédure n'est déclarée, le nœud correspondant ne porte pas de fils et est représenté tel quel dans le texte du document. Il en est de même pour les nœuds `<instruction>`, `<condition>` ou `<idf>` par exemple.

Les deux représentations du document, le texte et l'arbre, sont couplées en permanence. Une modification du texte entraine une modification automatique de l'arbre. Inversement, la modification de l'arbre entraine celle de la représentation textuelle du document. Le couplage est complet, la désignation avec la souris d'une position dans le texte du document entraine la sélection immédiate du

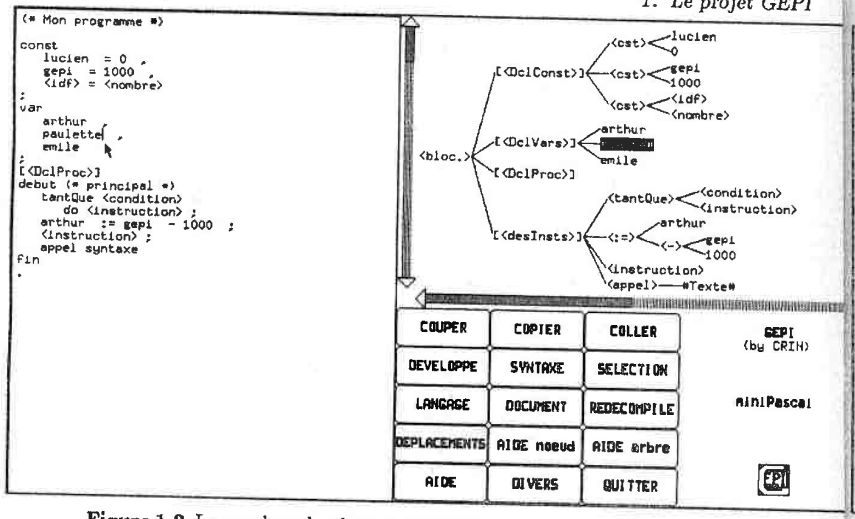


Figure 1.2. Le couplage des deux représentations est complet. Une modification de l'arbre est aussitôt répercutée dans le texte et inversement. La position du curseur dans le texte de l'éditeur pleine page correspond toujours au nœud courant de l'éditeur d'arbre, en inverse vidéo.

noeud correspondant dans l'arbre de la fenêtre droite (cf. Fig. 1.2). Inversement, la sélection d'un nœud à l'aide de la souris dans la fenêtre de l'éditeur d'arbre entraine le déplacement de la position du curseur dans le texte du document.

Les feuilles non encore syntaxiquement validées portent la mention `#texte#` (cf. Fig. 1.2) qui indique visuellement à l'utilisateur que la zone de texte correspondante n'est pas validée syntaxiquement[5]. L'arbre peut comporter plusieurs nœuds *texte*. La présence de tels nœuds n'empêche pas de travailler sur les autres parties du document.

1.2.3 Créer des programmes avec GÉPI

Deux modes d'édition de documents cohabitent en permanence : l'édition textuelle et l'édition structurelle, principalement sous la forme d'expansions. L'expansion structurelle consiste à construire le texte par affinages successifs en développant le programme à partir de l'axiome de la grammaire. A chaque développement, le système propose les différentes constructions syntaxiques valides. L'utilisateur choisit alors celle qui lui convient et le système l'insère dans le texte. La construction d'un document se fait donc sans frappe de texte au clavier, simplement en choisissant les structures syntaxiques qui conviennent.

5. Dans les versions à venir du manipulateur de documents, il est prévu de visualiser l'incorrection syntaxique dans la fenêtre textuelle en mettant la zone correspondante en caractères italiques par exemple.

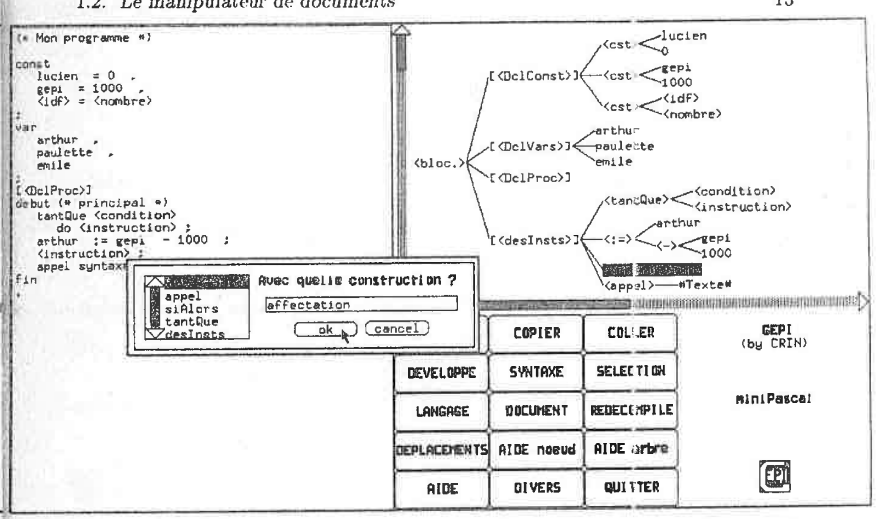


Figure 1.3. L'étape d'expansion structurelle. La commande de développement appliquée sur un nœud `<instruction>` provoque l'apparition d'un menu comportant les différentes possibilités d'expansion. Une fois le choix effectué, la structure correspondante est insérée dans le document, l'arbre et le texte sont mis à jour en conséquence.

Les figures 1.3 et 1.4 montrent une étape d'expansion structurelle. La commande de développement appliquée sur un nœud `<instruction>` provoque un dialogue entre l'utilisateur et l'éditeur. Les différentes possibilités d'expansion de ce nœud sont présentées sur un menu (cf. Fig. 1.3). Le choix effectué, la structure syntaxique correspondante remplace le nœud sur lequel la commande de développement a été déclenchée. La figure (cf. Fig. 1.4) montre le document obtenu après la sélection de la construction `affectation`. Le noeud `<instruction>` est ainsi remplacé par le sous-arbre qui représente une affectation, `<:=>`. Le texte du document est lui aussi mis à jour en replacant le texte "`<instruction>`" par le texte "`<idf> := <expression>`".

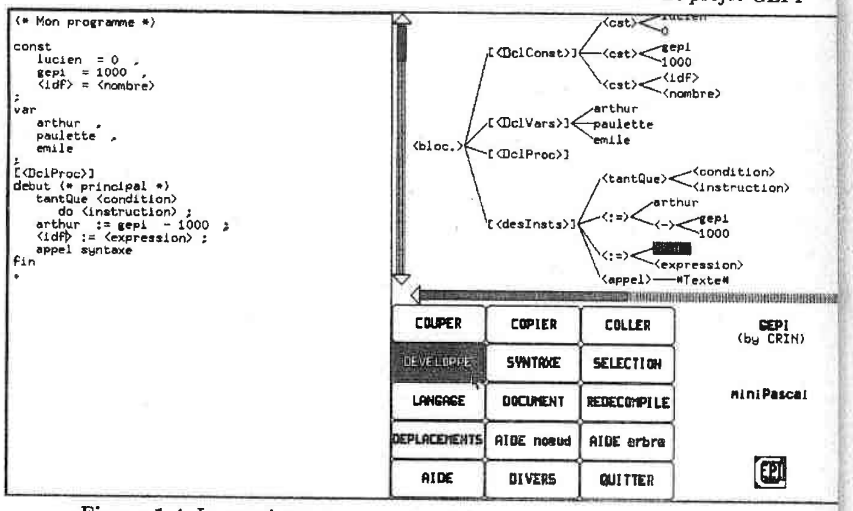


Figure 1.4. Le nœud `<instruction>` a été remplacé par un nœud représentant une affectation, `<:=>`.

1.2.4 Modifier des programmes avec GÉPI

Il est bien entendu possible de taper directement son texte au clavier comme on le fait sur un éditeur ordinaire. Dans ce cas, l'éditeur de GÉPI ne contraint pas l'utilisateur en le forçant à respecter la syntaxe du langage utilisé. La partie du document qui a été modifiée à la main est considérée syntaxiquement incorrecte mais n'est pas rejetée pour autant. Le sous-arbre affecté par les modifications est automatiquement remplacé par une feuille `#texte#`. Le programmeur a la possibilité de demander l'analyse syntaxique du fragment de texte non validé. Cette analyse a deux objectifs : d'une part détecter d'éventuelles erreurs, et d'autre part construire la partie d'arbre correspondant à ce texte. Lorsque la zone correspondante est validée syntaxiquement, l'arbre est aussitôt modifié en conséquence. Pour un identificateur, cela correspond à remplacer le nœud `#texte#` par un nœud portant le nom de l'identificateur lui même.

L'arbre représentatif du document peut être réduit à un unique nœud `#texte#`. Ce cas se produit lorsque du texte est entré manuellement dans la zone correspondant à la racine seule, c'est-à-dire lorsque l'endroit d'insertion d'un caractère ne correspond pas à une des sous-zones de la racine.

D'autres commandes sont également disponibles : insertion dans des listes, commandes de promenade structurelle correspondant à un déplacement dans l'arbre ou de promenade textuelle correspondant à un déplacement dans le texte ... Au niveau de la création de document, l'éditeur GÉPI se comporte donc comme la plupart des éditeurs syntaxiques connus. C'est surtout au niveau de la manipulation et modification de documents qu'il se révèle particulièrement intéressant.

Les éditeurs syntaxiques classiques considèrent un programme comme étant avant tout un arbre et le manipule comme tel. Il n'est donc possible de manipuler un document qu'en termes de structures syntaxiques car les commandes de modifications sont des opérations qui agissent sur l'arbre. Par exemple, une commande de destruction est en fait une commande d'élagage de l'arbre et il n'est donc possible de détruire qu'une partie de texte correspondant à un sous-arbre **complet**. Nous pensons que ceci est beaucoup trop rigide et qu'il est nécessaire de manipuler et de modifier un programme de manière textuelle. Par exemple, il doit être possible de supprimer ou déplacer un fragment de texte ne correspondant pas à une construction syntaxique complète.

1.2.5 Couper, Copier, Coller

Afin d'illustrer de manière plus concrète cette idée, prenons l'exemple d'une commande de modification bien connue des utilisateurs de Macintosh, le *couper, copier, coller* [Lu 84]. Cette commande de modification est constituée de trois opérations élémentaires. L'opération *couper* permet de supprimer une zone préalablement sélectionnée pour la mémoriser dans une mémoire tampon appelée presse-papier. L'opération *copier*, quant à elle, ne fait qu'une copie de la zone sélectionnée pour la mémoriser dans le presse-papier. Elle ne modifie pas le document. La dernière zone coupée ou copiée peut être insérée à un autre endroit du document à l'aide de l'opération *coller*.

Sur un éditeur syntaxique classique, l'équivalent de l'opération *couper* consiste à réaliser un élagage de l'arbre en recopiant le sous-arbre coupé dans un *buffer* structurel. Il n'est possible de couper que des constructions syntaxiques complètes. Ensuite, *coller* réalise une greffe du buffer structurel sur l'arbre représentant le programme. Bien entendu, cette greffe n'est possible que sous certaines conditions d'unification entre le noeud où l'on colle et le noeud racine du buffer. Il apparaît donc qu'il n'est pas possible de déplacer n'importe quel fragment de texte, ce qui impose donc des contraintes assez fortes.

L'éditeur de GÉPI par contre ne se comporte pas de la même manière. Il utilise pour ces opérations deux tampons mémoire : un buffer structurel et un buffer tex-

tuel. L'opération *couper* fonctionne de la façon suivante : le texte à couper est systématiquement recopié dans le buffer textuel et s'il correspond à une construction complète, alors l'arbre est élagué et recopié dans le buffer structurel. Par contre, s'il ne s'agit pas d'une construction complète, l'arbre est également élagué mais la construction syntaxique englobante est alors considérée comme invalide, mise en relief pour l'utilisateur, et nécessitera une analyse syntaxique pour reconstruire correctement l'arbre. Le buffer structurel quant à lui est marqué comme étant incorrect, donc inutilisable. L'opération *coller* fonctionne de manière similaire. Le contenu du buffer textuel est systématiquement collé à l'endroit prévu. Lorsque le buffer structurel est correct (i.e. il contient un arbre) et que les conditions d'unification entre le noeud où l'on colle et la racine du sous-arbre sont réunies, la greffe est réalisée. Si ce n'est pas le cas (buffer invalidé ou greffe impossible), on ne fait pas la greffe mais la zone où le texte a été collé est considérée comme incorrecte et mise en relief pour l'utilisateur. Cette zone pourra, sur décision de l'utilisateur, faire l'objet d'une analyse syntaxique afin de reconstruire la partie d'arbre associée.

Dans tous les cas, afin d'avoir une souplesse d'utilisation maximale, la priorité est donnée aux manipulations textuelles, les zones affectées de l'arbre étant automatiquement remplacées par des feuilles contenant du texte.

1.2.6 Changement de langage

Plusieurs manipulateurs de documents peuvent fonctionner simultanément, permettant la manipulation de documents issus de formalismes différents, C, Ada [Ada 83] ou LaTeX... Les opérations *couper*, *copier*, *coller* peuvent s'appliquer entre ces différents manipulateurs, le presse-papier, comme sur le Macintosh d'Apple étant une donnée globale. Comme d'habitude, lorsque qu'il n'y a pas de correspondance syntaxique entre la partie de texte collée et la zone réceptrice, le collage est purement textuel, et passe par la création de nœud `#texte#`.

Afin de pouvoir mettre au point plus aisément les descriptions de langages, le manipulateur de documents autorise la modification dynamique du langage qui lui est associé. Lorsque l'on manipule un document et que l'on *recharge* la description de langage, seul le texte du document est conservé, l'arbre étant alors remplacé par une unique feuille `#texte#`. Le texte peut bien entendu être aussitôt analysé avec la nouvelle description de langage. Cette technique permet par exemple de vérifier simplement que la mise en place du paragraphage correspond à ce que l'on attendait.

On peut mettre au point une description de langage de façon progressive. En outre, la description du langage de description de langage lui-même est disponible (cf. annexe 10), ce qui permet d'utiliser le manipulateur de documents pour éditer les descriptions de langages.

1.2.7 Analyse syntaxique

L'analyseur syntaxique a une grande importance dans le fonctionnement de l'éditeur car il est chargé de l'analyse des nœuds contenant du texte entré librement. Ses caractéristiques sont adaptées au manipulateur de documents de GÉPI (cf. § 7.4)

La première caractéristique de cet analyseur est d'être capable de démarrer l'analyse à partir de n'importe quel non-terminal de la grammaire du langage, ceci afin de ne pas avoir à faire l'analyse sur l'ensemble du document, mais seulement sur les zones où cela est nécessaire.

Une deuxième caractéristique est sa capacité à prendre en compte des textes contenant des non-terminaux de la grammaire. Il peut en effet arriver qu'une zone destructurée contienne des morceaux de textes incomplètement affinés. L'analyseur doit donc pouvoir les prendre en compte s'il veut reconstruire l'arbre correspondant.

Une troisième caractéristique est qu'il peut travailler sur une classe de grammaires assez large afin d'éviter des contraintes inutiles lors de la paramétrisation du générateur. Cet analyseur syntaxique a été réalisé sur la base de l'algorithme d'Earley[Earley 70] et modifié pour répondre aux besoins spécifiques de GÉPI.

1.3 Les autres outils de l'environnement

Le manipulateur de documents prend en charge les problèmes d'édition, de mise en page et de vérification syntaxique. Comme dans les systèmes CENTAUR [Clement 86] ou CPS [Reps 84], il est possible de décrire des outils permettant de dépasser le cadre purement syntaxique. Ces outils sont décrits séparément de la description du langage et servent à vérifier le respect de la sémantique statique du langage, à écrire des outils de documentation des programmes ou encore à interpréter le document. Les outils sont intégrés à l'interface du manipulateur de documents et sont déclenchés à la demande de l'utilisateur. On peut par exemple ajouter un outil vérifiant le respect de contraintes de type. Cet outil peut être déclenché par l'intermédiaire d'un bouton ou d'un menu du manipulateur de document.

Cette partie du projet GÉPI ayant fait l'objet d'une autre thèse [Silva 88], nous n'en ferons qu'un survol dans cette section.

1.3.1 Grammaires attribuées

Le principe utilisé par GÉPI est fortement inspiré de celui utilisé par CPS. Le formalisme qui sert à décrire les outils est basé sur l'utilisation de *grammaires attribuées* [Knuth 68].

Une grammaire attribuée est composée d'une grammaire à contexte libre et d'un ensemble d'équations appelées *équations sémantiques*. Les équations sémantiques sont attachées aux différentes productions de la grammaire et manipulent des *attributs*. Pour un document particulier, l'évaluation de ces équations en regard de la syntaxe du document permet de valuer un ensemble d'attributs et ainsi de vérifier certaines propriétés du document.

Le principe consiste dans un premier temps à construire *l'arbre de dérivation* du document. Un arbre de dérivation est construit en associant un nœud à chaque production de la grammaire utilisée pour reconnaître le document. Sur chacun de ces nœuds, les attributs de la production correspondante prennent des valeurs particulières. Les attributs permettent ainsi de représenter l'information associée aux différents nœuds d'un arbre syntaxique. Le deuxième temps consiste à parcourir l'arbre attribué en évaluant les équations correspondantes afin de fixer la valeur des attributs. Cette deuxième étape est appelée évaluation d'attributs. La valeur obtenue pour les différents attributs constitue le résultat de l'évaluation.

1.3.2 Description d'un outil

Le principe décrit précédemment est adapté aux spécificités de GÉPI pour construire les outils dépassant le cadre syntaxique [Silva 87]. Ces adaptations sont nécessaires car la structure de l'arbre représentatif d'un document diffère de celle d'un arbre de dérivation. En particulier, l'arbre manipulé par GÉPI peut être incomplètement développé ou encore comporter des nœuds syntaxiquement incorrects.

La définition d'un outil revient cependant à écrire un jeu d'équations s'appuyant sur la description syntaxique du langage. Afin de ne pas dupliquer la description syntaxique du langage, celle utilisée par le manipulateur de documents est utilisée par les différents jeux d'équations.

Par exemple, un outil vérifiant le respect de contraintes de type est écrit à l'aide d'un jeu d'équations particulier. Le déclenchement de cet outil provoque l'évaluation des attributs associés à ce jeu d'équations. La valeur résultante des

attributs portés par l'arbre permet de connaître le résultat de la vérification de contraintes de type.

Bien entendu, lorsque l'arbre est incomplet ou comporte des parties syntaxiquement incorrectes, le résultat de l'évaluation peut être d'intérêt limité, les informations collectables étant peu nombreuses[6]. Si la partie déclaration des variables est syntaxiquement incorrecte, la vérification des contraintes de type dans le reste du document est impossible.

Dans un contexte d'édition syntaxique, l'arbre représentatif du document est continuellement modifié. Afin de ne pas recommencer complètement le processus d'évaluation sur l'ensemble de l'arbre, l'évaluateur travaille de façon incrémentale. D'une évaluation à l'autre, les attributs non concernés par les modifications conservent leur ancienne valeur.

1.3.3 Modularité

Plusieurs outils peuvent être décrits séparément à l'aide de jeux d'équations différents. Les différents outils peuvent communiquer entre eux en partageant des attributs communs. Ainsi, un outil peut être décomposé en plusieurs sous-outils comme un module peut être décomposé en sous-modules. Bien que définis séparément, les outils sont rassemblés au sein d'un environnement complet et sont utilisables simultanément. Un outil défini par morceaux est alors utilisable dans son ensemble. En outre, l'utilisateur peut reconfigurer dynamiquement l'environnement sur lequel il travaille en ajoutant ou retirant certains jeux d'équations.

La modularité est une caractéristique importante de la construction d'outils dans GÉPI. Elle permet de définir séparément chaque composant de l'environnement et éventuellement de découper un outil complexe en plusieurs modules plus simples. Par exemple, un outil pour la vérification de la sémantique statique d'un programme peut se décomposer en plusieurs modules : un module de gestion de la table des symboles, un module chargé de vérifier la déclaration des variables, un module réalisant le contrôle de types, un module de références croisées ... Ainsi de suite, chaque module peut à son tour être décomposé en modules plus simples.

Les différents modules peuvent éventuellement partager des attributs, c'est-à-dire que les attributs ayant été définis pour un outil peuvent aussi être utilisés, sans avoir à les redéclarer, par d'autres outils. Les attributs communs à plusieurs jeux d'équations introduisent des relations de dépendance entre modules.

6. Dans les première versions du projet, l'éditeur syntaxique était lui-même écrit comme un outil particulier, c'est à dire à l'aide d'équations sémantiques [Canals 87]. Cette solution a été abandonnée dans la suite car l'évaluation d'équations sur un arbre incomplet est problématique.

La possibilité de découpage d'outils se révèle particulièrement intéressante dans le cas d'outils complexes, et ce non seulement au moment de la conception mais aussi de l'utilisation de l'environnement. Prenons l'exemple d'un environnement Ada. Un outil réalisant la vérification de la sémantique statique de manière complète se révèlerait très coûteux et nuirait beaucoup à l'interactivité de l'environnement. Il est donc intéressant de pouvoir le décomposer afin de ne l'utiliser que par morceaux.

Ainsi, pour disposer d'un outil complet de vérification de la sémantique statique des programmes, il faut faire évoluer simultanément et coopérer les modules de gestion de la table des symboles, de vérification des déclarations, de contrôle de types, de références croisées...

Dans le système GÉPI, l'intégration est réalisée en faisant évoluer simultanément les évaluateurs d'attributs des différents modules. La coopération entre outils est alors obtenue grâce à l'utilisation par un outil d'attributs déclarés pour un autre.

L'intégration permet de réunir les différents outils au sein d'un même environnement et donc de construire l'environnement lui-même. C'est-à-dire que pour disposer d'un véritable environnement intégré, les outils qui ont été conçus séparément doivent être en mesure de travailler simultanément[7], de coopérer et collaborer tout au long de la construction d'un programme.

La reconfiguration dynamique permet à l'utilisateur de supprimer ou d'ajouter un ou plusieurs outils de l'environnement original. Bien entendu, dans certains cas la suppression d'un outil sera interdite par le système parce que l'outil en question est utilisé par d'autres.

Nous avons donc montré les caractéristiques principales de la construction des différents outils dans GÉPI. La modularité nous semble être la plus intéressante puisqu'elle intervient au moment de la conception et au moment de l'utilisation des environnements. Elle facilite la tâche du concepteur car elle lui offre la possibilité de définir séparément les différents outils ou de découper des outils trop complexes, contrairement à ce qui est fait dans des systèmes similaires au nôtre comme CPS où l'ensemble des outils est défini d'un seul bloc. Au moment de l'utilisation, elle permet la reconfiguration dynamique, ce qui fournit des environnements moins rigides, plus souples et plus agréables à utiliser [Canals 87].

7. Le mot simultanément n'implique pas forcément la notion de parallèlisme.

1.4 Conception de GÉPI

Concevoir un environnement comme celui offert par GÉPI est un exercice difficile. De nombreux problèmes se posent simultanément et, la résolution d'un problème particulier peut nuire à la bonne résolution d'un autre problème. En particulier, la définition du langage de description de langages est certainement la partie de GÉPI qui doit être étudiée avec le plus grand soin car la plupart des composants du manipulateur de documents utilisent cette description :

- le décompilateur utilise la description du langage pour effectuer l'indentation,
- l'analyseur syntaxique utilise aussi cette description qui doit donc contenir toutes les informations permettant d'effectuer l'analyse syntaxique, même les détails de lexicographie,
- le manipulateur de documents doit être capable de construire l'arbre représentatif d'un document, sa structure étant aussi définie par la description du langage.

Ne connaissant pas *a priori* quels seront exactement les besoins de chaque composant du manipulateur de document et chacun de ces derniers devant prendre place dans une même interface, il est difficile de suivre à la lettre les traditionnelles étapes du cycle de vie d'un logiciel.

1.4.1 Prototypage

A l'inverse des bons principes du génie logiciel [Boehm 82], la conception du manipulateur de documents n'a pas commencé par la rédaction d'un cahier des charges précis. La technique adoptée est expérimentale et repose sur l'écriture d'une succession de prototypes.

La technique du prototypage consiste à réaliser un produit le plus rapidement possible pour valider une idée [Choppy 86]. Le prototype permet d'obtenir rapidement des réactions sur la conception d'un système. Après sa production, il est aussitôt soumis à l'essai des utilisateurs et sert de base de discussion pour revoir ou confirmer les fonctionnalités du produit final.

C'est cette technique qui a guidé le développement de GÉPI. Sans même partir d'un cahier des charges détaillé et précis du générateur, nous avons développé GÉPI de prototypes en prototypes. C'est le plus naturellement du monde, c'est à dire à l'aide d'essais successifs, que nous avons pu fixer les fonctionnalités indispensables du générateur. Dans le domaine de l'interfaçage cette approche est

particulièrement intéressante, la qualité d'une interface étant principalement affaire de goûts, en un mot, subjective.

En parallèle avec la définition des fonctionnalités externes, le langage de description de langages a lui aussi évolué pour atteindre sa forme actuelle (cf. chap. 2). En outre, l'écriture des prototypes a permis de tester l'adéquation des structures de données manipulées ainsi que l'efficacité des algorithmes développés.

L'inconvénient majeur de la technique utilisée réside dans la remise en cause continuelle des programmes déjà écrits. Le prototype, à peine terminé, est remis en cause : telle commande n'est pas pratique, cet algorithme n'est pas très efficace ou ne marche pas dans tel cas de figure, le langage de description de langages ne convient pas.

Il va sans dire, qu'en utilisant cette technique, nous avons subi de nombreux échecs, particulièrement lors de la réalisation des premiers prototypes écrits à l'aide de langages classiques, Pascal et C en l'occurence. Il serait cependant trop injuste de n'accuser que les langages utilisés pour deux raisons. En premier lieu, quelque soit la méthode utilisée, la réussite d'un projet est souvent liée à la connaissance du domaine traité. L'échec cuisant des premiers prototypes était directement lié à notre méconnaissance du domaine traité. En deuxième lieu, les langages que nous avions choisis sont de toute évidence inadaptés au prototypage [Pro 86].

Incapables de fixer *a priori* un cahier des charges précis et détaillé de GÉPI, il nous fallait continuer dans notre voie première, construire un prototype et valider petit à petit nos idées. Cette voie, même si elle avait commencé par des échecs, avait déjà porté ses premiers fruits, les premiers échecs étant pour nous l'occasion de repenser le problème, de nous apercevoir des spécificités du domaine, en un mot, de prendre du recul.

Programmation en LISP

Continuant sur notre approche qui consiste à construire un prototype, nous nous sommes naturellement orientés vers des langages plus adaptés à cette méthode. Le passage à LISP [Chailloux 84] a bouleversé nos habitudes. L'homogénéité, la souplesse et la convivialité de LISP en font un outil de développement remarquable [Sandewall 84] [Tichy 87]. Il laisse au programmeur une totale liberté d'expression, en lui donnant le choix entre plusieurs styles de programmation [SaintJames 87]. De par sa nature même, LISP encourage l'emploi de la récursivité, facilitant la programmation des algorithmes complexes. Par sa nature dynamique et récursive, la liste simplifie la conception de structures de données complexes [Winston 84a].

En outre, LISP est avant tout un langage interprété. En phase de développement, l'absence d'étape de compilation raccourcit les délais de mise au point. Avec un langage compilé, il est nécessaire d'apprendre un, voire plusieurs langages de commande pour composer, compiler et mettre au point les programmes. Au contraire, LISP est son propre langage de commande : l'utilisateur a toutes les facilités pour enrichir ou modifier l'environnement de programmation selon ses goûts et ses besoins.

La correction des erreurs est complètement interactive et s'effectue dans le contexte d'exécution, favorisant un développement incrémental. Le programmeur est libéré du souci d'écrire des programmes spéciaux pour tester séparément les nouvelles fonctions.

Le *lecteur*[8] LISP est un analyseur intégré qui peut être directement utilisé pour saisir les données d'une application, sans recourir à des outils conçus spécialement par ailleurs, dont il faudrait en outre apprendre à se servir. En particulier, pour la saisie des descriptions de langage, l'utilisation du lecteur LISP évite l'écriture d'un analyseur syntaxique particulier. La modification du langage de description de langage peut ainsi être effectuée plus librement.

Grâce à l'utilisation de LISP, l'écriture des nouvelles versions du prototype devenait plus simple et plus rapide. La possibilité d'utiliser des structures de données hétérogènes et souples étant un aspect propice au prototypage [Abrial 85].

Programmation objet

De la programmation en LISP à la programmation objet, il n'y a qu'un pas [Cointe 82] [Cointe 85]. C'est d'abord par le biais de la couche objet Ceyx [Hullot 83b] que nous avons découvert cet aspect de la programmation [Colnet 86a].

Comme LISP, un langage à objets permet la construction de structures de données hétérogènes mais, en plus de cette particularité le mécanisme de *liaison dynamique* assure l'adéquation des opérations utilisées sur les éléments de cette structure (cf. § 3.2.4). L'arbre représentatif d'un document est une structure de données essentielle dans la conception d'un éditeur syntaxique. Cette structure de données est hétérogène dans le sens où chaque nœud de l'arbre correspond à une construction syntaxique particulière : un nœud représentant une affectation est différent d'un nœud qui représente une liste d'instructions. Bien entendu, les opérations effectuées sur un nœud comme la décompilation par exemple dépendent de la nature du nœud traité : on ne décompile pas de la même manière un nœud correspondant à une affectation et un nœud correspondant à une liste d'instructions.

8. Partie de l'interprète qui traduit la représentation externe des objets LISP en leur représentation interne.

Le mécanisme de liaison dynamique évite d'écrire le test qui permet de distinguer les différents cas. Ce point particulier est très intéressant en ce qui concerne la taille du code à écrire pour implanter le manipulateur de documents, l'opération de décompilation n'étant pas la seule à dépendre de la catégorie d'appartenance du nœud traité.

D'abord par curiosité, puis dans le cadre d'une étude sur les langages à objets [Colnet 86b], nous avons eu l'occasion d'essayer d'autres langages de cette famille, Smalltalk [Ingalls 81], Flavors [Moon 80], yafool [Ducournau 87b] et Eiffel [Meyer 86c] pour ne citer que les principaux. Très naturellement, certaines caractéristiques trouvées parmi ces langages ont été rajoutées progressivement sur la base de la couche objet de l'interprète LISP que nous utilisions [Chailloux 86]. Rajouté dans un premier temps comme de simples primitives supplémentaires, l'ensemble de ces caractéristiques a pris par la suite le nom d'un langage, *leloup* (cf. chap. 5).

Même s'il existe des divergences sur certains points particuliers, les caractéristiques de leloup restent essentiellement inspirées de celles de Smalltalk, et l'implantation de GÉPI décrite en leloup (cf. chap. 6 et chap. 7) peut très simplement être transcrite en Smalltalk. Dans le choix du système utilisé pour le projet GÉPI, la possibilité d'utiliser le lecteur de l'interprète LISP pour la saisie des descriptions de langages est un facteur important. En outre, l'implantation de leloup nous a permis de mieux comprendre les mécanismes objets en les mettant en œuvre nous-même.

La syntaxe de leloup étant extrêmement simple et assimilable, l'implantation de GÉPI constitue selon nous un exercice grandeur nature d'utilisation des mécanismes objets en général, les techniques utilisées étant applicables à tous les langages de la famille Smalltalk.

Mis à part peut-être, les langages à objets compilés comme Eiffel ou Objective-C qui ne disposent pas d'environnement de programmation hautement interactif, les langages à objets sont d'excellents outils de prototypage [Bezivin 86a], Smalltalk étant particulièrement bien adapté aux applications interactives et graphiques [Bezivin 86b]. Les raisons principales qui en font de bons outils de prototypage peuvent être résumées par les aspects suivants :

1. la plupart du temps, l'environnement de développement est complètement interactif et comporte de nombreux outils comme celui de Smalltalk [Byt 81] [Goldberg 84a] ou celui de la Machine Lisp [Weinreb 81] [Greenblatt 84],
2. la bibliothèque d'utilitaires est riche et souvent, les outils ayant servi à l'implantation du système lui-même sont disponibles pour l'implantation des autres applications (c'est le cas de Smalltalk et de la Machine Lisp),

3. le style de programmation induit par les langages à objets permet d'écrire des programmes hautement modulaires et réutilisables [Johnson 88] [Meyer 87d] [Nierstrasz 88],
4. les langages à objets permettent l'utilisation de structures de données hétérogènes et souples.

Quelques remarques s'imposent au sujet des différents points énumérés. Le point (1), disposer d'un environnement interactif est évident en matière de prototypage [Choppy 86]. Les points (2) et (3) sont fortement liés : une bibliothèque de programmes, même riche, est difficilement utilisable si le langage ne fournit pas de bons mécanismes de réutilisabilité. L'héritage des langages à objets fournit un élément de réponse à ce problème [Meyer 87b] [Snyder 86]. Le point (4), même s'il n'est pas mis en avant dans la littérature consacrée aux langages à objets, nous semble fondamental dans le domaine du prototypage et de la programmation en général (cf. § 3.3.2 et § 4.1.3).

1.4.2 Implantation

Dans les éditeurs syntaxiques classiques, un document est représenté de manière unique par un arbre abstrait et le système manipule uniquement cette structure de données. Le texte, quant à lui, est considéré comme une simple vue sur cet arbre, obtenue par *décompilation*[9]. La vue décompilée de l'arbre est destinée à faciliter le travail de l'utilisateur en lui permettant de manipuler cet arbre au travers de sa représentation textuelle. En fait, sur un tel éditeur, les commandes, même si elles s'effectuent sur un texte, sont des commandes de manipulation d'arbres : construction par greffe de sous-arbres, déplacement de sous-arbres, promenade dans l'arbre...

L'arbre et le texte du document

La solution adoptée dans GÉPI est différente de celle adoptée par Cépage ou MENTOR par exemple. Elle consiste à ne plus avoir une structure de données unique représentant un programme mais à maintenir en permanence deux représentations d'un document : l'une contenant sa forme textuelle et l'autre sa forme arborescente.

L'inconvénient majeur de cette solution réside dans le fait qu'il faut en permanence assurer un couplage entre ces deux représentations. Comme nous l'avons

9. Dans le domaine de l'édition syntaxique, la décompilation est l'opération qui consite à *calculer* le texte d'un document à partir de sa représentation arborescente.

montré précédemment (cf. § 1.2), ce travail est pris en charge par le manipulateur de documents de façon transparente. L'utilisateur peut indifféremment modifier l'arbre ou le texte, ses modifications sont automatiquement reportées dans l'arbre ou dans le texte. L'avantage de disposer en permanence de ces deux représentations est d'obtenir à la fois de la souplesse de l'édition textuelle et la puissance de l'édition syntaxique. En effet, il est possible d'utiliser indifféremment des commandes agissant sur le texte et des commandes agissant sur l'arbre.

Les deux représentations ne sont pas indépendantes mais complémentaires : les informations contenues dans chacune sont nécessaires à l'exécution de toutes les opérations d'édition. Les opérations textuelles n'agissent pas uniquement sur la représentation textuelle. Il en est de même pour les opérations structurelles.

Examinons plus en détails l'utilisation de ces deux représentations. Lorsqu'une opération de nature textuelle est exécutée, elle se répercute bien entendu sur la représentation textuelle du programme. Cependant, il peut arriver qu'elle modifie localement le texte du document, créant ainsi une incohérence entre les deux représentations. Il faut donc répercuter cette information sur l'arbre. Une première solution consiste à détruire le sous-arbre incorrect et à le recalculer systématiquement par analyse syntaxique du texte modifié. Cependant, cette solution semble un peu lourde car les modifications risquent souvent d'être purement temporaires et donc de contenir de nombreuses erreurs, l'utilisateur prévoyant des modifications ultérieures. Il est donc préférable de retarder la phase d'analyse et de ne la faire que lorsque l'utilisateur le demande, c'est à dire lorsqu'il pense que son texte est correct. Pour retarder cette analyse, GÉPI marque le sous-arbre concerné en le remplaçant par une feuille contenant du texte non encore analysé. Les informations permettant de retrouver cette zone de texte dans la représentation textuelle du programme sont stockées dans l'arbre. Ainsi, le système est capable de repérer à tout moment quelles sont les parties de l'arbre qui ne sont pas valides et où se trouvent les zones de texte nécessaires au recalcul de cette partie d'arbre.

Le cas de l'exécution d'opérations structurelles est plus simple car elles ne peuvent agir que sur des morceaux de programme où les deux représentations sont cohérentes, c'est à dire où l'arbre est valide et représente fidèlement la structure du texte. Ces opérations se regroupent dans les catégories suivantes :

- Promenade ou recherche structurelle : ces opérations ne posent pas de problèmes car elles ne modifient pas le document.
- Opérations de type *couper/coller* : la destruction ou l'insertion d'un fragment de programme se fait en même temps sur les deux représentations. Il suffit de mettre à jour les informations permettant de faire le lien entre ces deux structures de données.
- Expansion : l'opération d'expansion se fait en greffant un sous-arbre obtenu par dérivation des règles de la grammaire. La mise à jour du texte se fait alors par décompilation locale du sous-arbre greffé et insertion du résultat dans le texte.

Les remarques précédentes sur l'utilisation des deux représentations mettent en valeur la forme de l'arbre que nous utilisons et qui est un peu particulière. De manière générale, il s'agit d'un arbre abstrait qui représente la structure du document en cours de construction. Cependant, comme nous l'avons vu ci-dessus, il peut arriver qu'il existe des incohérences entre le texte du document et sa représentation structurée. Des feuilles de l'arbre correspondent à des zones de texte entrées librement par l'utilisateur. Nous parlons dans ce cas d'arbre destructuré car les informations qu'il contient et qui sont utilisées par le système sont de nature textuelle et non pas structurelle. La forme générale de la représentation arborescente d'un programme est donc une combinaison de zones structurées et de zones destructurées. Il est bien entendu que ces zones destructurées n'ont qu'une existence temporaire puisqu'elles correspondent à des lieux d'incohérence entre le texte et l'arbre. Ces incohérences seront supprimées lorsque l'utilisateur fera appel à l'analyseur syntaxique qui reconstruira de manière correcte l'arbre représentant le fragment de texte.

L'utilisation de deux représentations présente l'avantage de résoudre de façon simple le problème des commentaires qui peuvent apparaître à n'importe quel endroit du document. En effet, un commentaire ne fait pas partie de la description syntaxique d'un langage, le mot syntaxique étant pris ici dans son sens le plus rigoureux : les commentaires n'apparaissent pas dans la grammaire d'un langage. Aussi, il est difficile et peu cohérent de donner une place aux commentaires dans l'arbre syntaxique du document. Ce faisant, dans notre solution, les commentaires ne sont représentés que dans le texte du document et sont absents de la représentation arborescente.

1.5 Premier bilan

Un des premiers objectifs à l'origine du projet GÉPI est de concevoir un éditeur syntaxique réellement utilisable, même par des programmeurs expérimentés. A notre avis, les éditeurs syntaxiques actuels, Cépage, MENTOR ou CPS, ne font pas partie de cette catégorie car ils sont trop dirigistes, la syntaxe du document devant

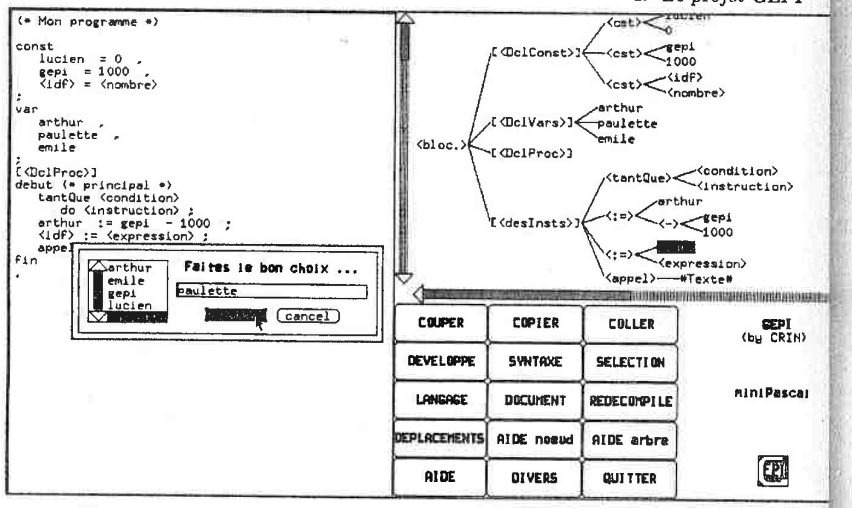


Figure 1.5. Développement d'une construction terminale sur le mode du développement des constructions non-terminales. Les différents identificateurs existants dans le reste du document sont présentés sous la forme d'un menu.

être respectée en permanence. Nous pensons avoir atteint ce premier objectif grâce au manipulateur de documents de GÉPI.

Nous terminerons ce premier chapitre en insistant sur l'aspect créatif et ludique du développement d'une application à l'aide d'une succession de prototypes. Le coté créatif est directement lié à la possibilité d'essayer immédiatement toute nouvelle fonction, la phase d'essai d'une fonction étant souvent l'occasion d'en imaginer une autre. Par exemple, dans le cas des commandes du manipulateur de documents, c'est en mettant au point la commande de développement structurel sur les constructions non-terminales que l'idée du développement des constructions terminales est apparue. La commande de développement, déclenchée sur une feuille destinée à porter un identificateur, provoque l'affichage d'un menu contenant tous les identificateurs déjà utilisés dans le reste du document (cf. Fig. 1.5).

De nombreuses autres commandes, comme "*couper*, *copier*, *coller*" ont vu le jour de cette manière, c'est-a-dire en essayant les commandes existantes. Bien entendu, ces commandes auraient pu être spécifiées dès le départ, pendant l'élaboration du cahier des charges. Cependant, lorsqu'un produit à construire sort un tant soit peu des produits d'utilisation courante, il est bien difficile de fixer *a*

priori ses fonctionnalités.

L'aspect ludique du prototypage avec un langage à objets est plus subjectif et les considérations qui suivent n'engagent que nous. Le fait de développer un logiciel avec un langage à objets suggère la conception d'entités autonomes regroupées au sein de classes : chaque objet possède un comportement qui lui est propre. Bien entendu, cet aspect de la programmation objet favorise l'encapsulation de données et augmente le niveau d'abstraction. Une conséquence indirecte de ce mode de développement réside dans le fait qu'en écrivant les spécificités d'un objet, le programmeur a tendance à s'identifier à l'objet qu'il décrit. Cet aspect est particulièrement évident lorsque l'on regarde les commentaires accompagnant les descriptions de classes du système Smalltalk. Les commentaires sont écrits à la première personne : l'objet parle.

2
Le langage de description de langages

Un chien vint dans l'office
Et prit une andouillette
Alors à coups de louche
Le chef le mit en miettes
Les autres chiens ce voyant
Vite vite l'ensevelirent [...]
Au pied d'une croix en bois blanc
Où le passant pouvait lire :
Un chien vint dans l'office
Et prit une andouillette
Alors à coups de louche
...

Samuel Beckett
En Attendant Godot, acte 2

Le langage de description de langages permet de décrire de façon concise, complète et simple la représentation concrète et abstraite d'un langage. La représentation abstraite permet de construire les arbres représentatifs des documents. La représentation concrète est utilisée pour la décompilation et l'analyse syntaxique.

Grâce à la notion de *construction*, un seul formalisme est utilisé pour décrire simultanément la grammaire abstraite et la grammaire concrète du langage. L'écriture des différentes constructions d'un langage s'apparente à l'écriture d'une grammaire à contexte libre [Aho 73].

Le langage de description de langages de GÉPI s'inspire fortement de celui de Cépage [Meyer 86a] en reprenant l'ensemble de ses *construct paragraphs* [Meyer 87c]. Deux nouvelles catégories de constructions ont été ajoutées pour prendre en compte

plus facilement le sucre syntaxique et pour traiter le cas des répétions contenants au moins un élément. Certaines redondances ont été évitées. En outre, la description du paragraphage, de la lexicographie et des commentaires est prise en compte de façon plus systématique.

Plan

Ce chapitre commence par une vue d'ensemble sur les différentes rubriques constituant la description du langage (§ 2.1). Après la partie consacrée à la lexicographie et aux expressions régulières (§ 2.2), la partie la plus importante du chapitre décrit les cinq catégories de constructions (§ 2.3). Avant de clore le chapitre par la table des actions de paragraphage (§ 2.5), la description complète d'un langage est donnée (§ 2.4). Ce langage, est la version française de PL0, un micro Pascal tirée de [Wirth 76]. Il sert tout au long du chapitre à illustrer nos propos.

2.1 La description d'un langage

L'objectif qui a guidé la définition du langage de description de langages est de pouvoir intégrer au sein d'un unique document plusieurs aspects de la description d'un langage, de façon aussi concise que possible, en évitant les duplications. Les aspects pris en compte par le langage de description de langages sont les suivants :

- description de la syntaxe concrète et de la lexicographie,
- description de la syntaxe abstraite,
- description du paragraphage, la mise en page d'un document,
- description des bruits associées au langage, comme les commentaires par exemple.

Sur d'autres systèmes, comme Mentor par exemple, la description de la syntaxe concrète est séparée de la description de la syntaxe abstraite. En réunissant dans un même document la description de ces deux syntaxes, on facilite la mise à jour en évitant les duplications inutiles. En outre, la description du langage peut être mise au point progressivement à l'aide du générateur qui accepte des descriptions incomplètes. La description du langage de description est disponible (cf. annexe 10) et permet d'utiliser GÉPI lui-même pour manipuler les nouvelles descriptions.

Pour décrire simultanément la syntaxe concrète et la syntaxe abstraite, l'écriture de règles de grammaire est remplacée par l'écriture de constructions. Différentes catégories existent, chacune étant spécialisée dans la description d'une partie ou d'un aspect du langage : structure de contrôle, lexicographie d'un identificateur ou d'un commentaire, répétition, alternative ...

La description d'un langage est principalement contituée par l'énumération des constructions qui le compose. Différentes rubriques, concernant la lexicographie et le paragraphage, doivent être remplies avant de passer à cette énumération (cf. Fig. 2.1). Chaque rubrique est précédée d'un mot clé, en italiques dans la figure. Le nom du langage décrit est indiqué derrière le mot clé *langage*. La liste des constructions du langage, derrière le mot clés *constructions*, constitue la partie principale d'une description de langage. Avant d'en arriver là, les parties qui précèdent fixent en détails la lexicographie ainsi que les valeurs par défaut pour le paragraphage. Pour une première lecture, il n'est pas nécessaire de lire en détail la partie lexicographie (cf. § 2.2). En particulier, la description des expressions régulières peut même être omise pour passer directement à la présentation des cinq catégories de constructions (cf. § 2.3).

2.2 Lexicographie

2.2.1 Les terminaux

La description lexicographique d'un langage commence par la déclaration de ses différents terminaux. Classiquement, on distingue deux catégories de terminaux, les terminaux ordinaires qui ne peuvent pas apparaître dans le texte source collés les uns aux autres, et les terminaux séparateurs qui servent à séparer les différentes entités lexicographiques.

Le mot clé *terminaux* précède la liste des terminaux ordinaires et le mot clé *terminauxSeparateurs*, celle des terminaux séparateurs. Un terminal s'écrit comme une chaîne de caractère LISP et est délimité par le caractère " (double quote). Ainsi, la liste des terminaux de notre mini langage s'écrit :

```
( terminaux
    "programme" "debut" "fin" "tantQue" "faire" "si" "alors" "sinon")
```

Les terminaux séparateurs possèdent la particularité de pouvoir séparer les entités lexicographiques les unes des autres sans qu'on ait besoin de marquer explicitement la séparation à l'aide d'un ou plusieurs caractères séparateurs. Ainsi, le signe d'affectation, le terminal ":=", est un séparateur en Pascal. La partie droite comme la partie gauche de l'affectation peuvent être collées contre ce terminal. Comme les terminaux ordinaires, il s'écrivent à l'aide de chaînes de caractères LISP :

```
( langage  <nomDuLangage>
     ( terminaux
          <t1> <t2> <t3> ....)
     ( terminauxSeparateurs
          <ts1> <ts2> <ts3> ....)
     ( abreviations
          <a1> <a2> <a3> ....)
     ( bruits
          <corpsDeConstruction>)
     ( paragraphage
          ( actionParDefaut
               <action de paragraphage>)
          ( indentationParDefaut
               <entier>))
     ( constructions
          <construction1>
          <construction2>
          <construction3>
          .....
          .....
          ..... ))
```

Figure 2.1. Structure globale de la forme externe de la description d'un langage.

```
( terminauxSeparateurs "=" ":=" "(" ")" "+" "-" "*")
```

Lorsqu'un terminal doit contenir des caractères particuliers, caractères de contrôle, doubles quotes ..., on peut utiliser la même notation qu'en Le-Lisp. Par exemple, pour mettre une double quote dans une chaîne de caractères, il suffit de la doubler. De même, toutes les notations du lecteur de l'interprète Le-Lisp sont utilisables, y compris les macros caractères qui déclenchent des fonctions LISP pouvant modifier le flux d'entrée au moment même de leur lecture (cf. §6-15 du manuel de référence Le-Lisp [Chailloux 86]). Mise à part l'utilisation des caractères de contrôle indispensables en lexicographie (cf. Fig. 2.2), on peut lire ce chapitre sans être un expert du lecteur Le-Lisp.

2.2.2 Les expressions régulières

La description des autres entités lexicographiques, comme les identificateurs ou les commentaires, nécessite l'utilisation d'expressions régulières. Une telle expression permet de décrire un langage régulier [Aho 77, page 85]. Intuitivement, une expression régulière représente un langage que l'on peut obtenir par combi-


```
#\sp     →   Le caractère espace.
#\tab    →   Le caractère tabulation.
#\cr     →   Le caractère fin de ligne.
#\esc    →   Le caractère escape.
```

Figure 2.2. La notation Le-Lisp de quelques caractères de contrôle.

```
( tranche <caractèreDeDébut> <caractèreDeFin>)
```

Figure 2.3. Les intervalles de caractère, l'opérateur *tranche*.

naison de langages finis en se limitant à l'utilisation des opérations d'union, de concaténation et de répétition.

Dans le langage de description de langages, les expressions régulières prennent la forme de listes LISP. Avant de voir les endroits de la description du langage où il faut placer les expressions régulières, étudions ces dernières plus en détail.

Les éléments de base.

L'entité élémentaire pour un analyseur lexicographique est le caractère. Ainsi, un caractère, constitue à lui seul une expression régulière à part entière. Par exemple, l'expression régulière constituée uniquement du caractère `"c"` reconnaît le caractère `"c"` lui-même. Si un autre caractère que `"c"` est le caractère courant du flux d'entrée de l'analyseur lexicographique, l'expression `"c"` n'est pas reconnue.

De même, les chaînes de caractères sont elles aussi des expressions régulières à part entière : la chaîne `"xyz"` reconnaît le caractère `"x"`, immédiatement suivi du caractère `"y"`, lui-même immédiatement suivi du caractère `"z"`. Tout autre caractère apparaissant à ce moment là dans le flux d'entrée fait que l'expression `"xyz"` n'est pas reconnue.

Les caractères isolés et les chaînes de caractères sont les éléments de base de notre langage de description lexicographique. Avant de passer à la présentation des opérateurs de plus haut niveau, voici le plus simple, celui qui permet de représenter des intervalles de caractères, l'opérateur *tranche*.

Intervalle de caractères

Pour indiquer à l'analyseur lexicographique que le prochain caractère peut être compris dans un intervalle des caractères du code ascii, il suffit d'utiliser l'opérateur

```
( ou <expression1> <expression2> <expression3> ... )
```

Figure 2.4. Alternative d'expressions, l'opérateur *ou*.

tranche (cf. Fig. 2.3) avec deux arguments, le caractère de début de l'intervalle et le caractère de fin. Ces deux caractères font eux-mêmes partie de l'intervalle qu'ils désignent. Par exemple, l'expression suivante indique que le prochain caractère doit être une des lettres majuscules de l'alphabet :

```
( tranche "A" "Z")
```

Alternatives d'expressions

L'opérateur *ou* permet d'exprimer un choix entre plusieurs expressions (cf. Fig. 2.4) mises en compétition. Par exemple, pour indiquer que le prochain caractère peut être soit le caractère "+", soit le caractère "-", ou enfin le caractère "*", on utilise l'opérateur *ou* avec ces trois arguments :

```
( ou  "+"  "-"  "*")
```

Bien entendu, les arguments de l'opérateur *ou* peuvent être des expressions quelconques. Par exemple, pour exprimer que le prochain caractère doit être une lettre minuscule ou une lettre majuscule :

```
( ou
    ( tranche "A" "Z")
    ( tranche "a" "z"))
```

Pour exprimer que le prochain mot attendu est soit "si", soit "tantQue", soit "debut" ou enfin, une lettre minuscule :

```
( ou
   "si"
   "tantQue"
   "debut"
   ( tranche "a" "z"))
```

Parmi les alternatives d'un opérateur *ou*, lorsque plusieurs expressions ont le même début, c'est toujours la plus longue qui est prise en compte. Dans l'exemple précédent, la chaîne "si" sera reconnue plutôt que le caractère "s" de l'intervalle (*tranche* "a" "z").

Séquences d'expressions


```
(<expression1> <expression2> <expression3> ... )
```

Figure 2.5. Séquence d'expressions.

```
(0-n <expression1> <expression2> <expression3> ... )

(1-n <expression1> <expression2> <expression3> ... )
```

Figure 2.6. Séquences répétées, les opérateurs *0-n* et *1-n*.

Le simple fait de mettre à l'intérieur d'une liste des expressions les unes à la suite des autres exprime une séquence de reconnaissance (cf. Fig. 2.5). Une séquence est reconnue si toutes les expressions qu'elle contient sont elles-mêmes reconnues dans l'ordre de la séquence. Ainsi, pour reconnaître une lettre minuscule suivie d'un caractère numérique, il faut écrire :

```
( ( tranche "A" "Z") ( tranche "0" "9") )
```

Répétition d'expressions

Les opérateurs *0-n* et *1-n* permettent d'exprimer la répétition d'une séquence d'expressions (cf. Fig. 2.6). Si la séquence à répéter doit l'être au moins une fois, il faut utiliser l'opérateur *1-n*. Si elle peut ne pas être répétée du tout, on utilise l'opérateur *0-n*. En particulier, l'opérateur *0-n* reconnaît toujours la chaîne vide. L'expression suivante avance dans le texte jusqu'au prochain caractère différent du caractère espace :

```
( 0-n " ")
```

Celle-ci, avance jusqu'au prochain caractère différent du caractère espace, #\sp, du caractère tabulation, #\tab, et du caractère de fin de ligne, #\cr :

```
( 0-n
     ( ou
         #\sp
         #\tab
         #\cr))
```

Si on veut imposer la présence d'un de ces trois caractères habituellement séparateurs, il faut utiliser une répétition *1-n* :

```
(0-1 <expression1> <expression2> <expression3> ... )
```

Figure 2.7. Une séquence optionnelle, l'opérateur *0-1*.

```
(sauf <expression1> <expression2> <expression3> ... )
```

Figure 2.8. Complémentaire d'une séquence, l'opérateur *sauf*.

```
( 1-n
    ( ou
        #\sp
        #\tab
        #\cr))
```

Séquence optionnelle

La séquence optionnelle, l'opérateur *0-1*, permet d'indiquer qu'une séquence d'expressions peut apparaître une fois ou non dans le texte source. Pour indiquer la présence optionnelle du caractère "+", devant un nombre par exemple :

```
( 0-1 "+")
```

Tout comme l'opérateur *0-n*, l'opérateur *0-1* reconnaît la chaîne vide.

Complémentaire d'une séquence

L'opérateur *sauf* permet de reconnaître n'importe quelle séquence à l'exception de celle qu'il contient. On avance dans le texte jusqu'au début de la séquence contenue par l'opérateur *sauf*. Par exemple, pour se positionner au début du prochain commentaire Pascal, c'est à dire juste avant le marqueur de début de commentaires, "(*" :

```
( sauf "(*")
```

Pour reconnaître un commentaire Pascal, il faut reconnaître le marqueur de début, puis avancer jusqu'au marqueur de fin, "*)", et enfin consommer le marqueur de fin lui-même :

```
("(*" ( sauf "*)") "*)")
```

De même, pour un commentaire Ada, qui commence par deux caractères moins, "--", et qui se termine à la fin de la ligne :


```
( abreviations
      (<a1> <expression1>)
      (<a2> <expression2>)
      (<a3> <expression3>)
      .....
      .....
      ..... )
```

Figure 2.9. Définition d'abréviations, la rubrique *abreviations*.

```
("--" ( sauf #\cr) #\cr)
```

Ou encore pour un identificateur Pascal, qui commence obligatoirement par une lettre, suivie éventuellement de caractères alpha-numériques :

```
(( ou
      ( tranche "a" "z")
      ( tranche "A" "Z"))
 ( 0-n
      ( ou
          ( tranche "a" "z")
          ( tranche "A" "Z")
          ( tranche "0" "9"))))
```

2.2.3 Définitions d'abréviations

Afin d'augmenter la lisibilité des expressions régulières, on peut définir des abréviations dans la partie *abreviations* de la description du langage (cf. Fig. 2.1). La liste apparaissant derrière le mot clé *abreviations* est constituée d'une suite de couples nom d'abréviation, expression correspondante (cf. Fig. 2.9). Les abréviations définies dans cette rubrique peuvent être utilisées dans les expressions régulières, dans la suite de la description du langage.

Ainsi, les abréviations `lettre`, `chiffre`, `lettreOuChiffre`, `commentaire` et enfin `caractereSpecial`, la liste des caractères spéciaux séparateurs, sont définies de cette façon :

```
( abreviations
    (lettre
        ( ou
            ( tranche "a" "z")
            ( tranche "A" "Z")))
    (chiffre
        ( tranche "0" "9"))
    (lettreOuChiffre
        ( ou lettre chiffre))
    (commentaire
        ( "(*" ( sauf "*)") "*)"))
    (caractereSpecial
        ( ou #\cr #\lf #\sp #\tab)))
```

L'expression qui décrit les identificateurs Pascal peut maintenant s'écrire :

```
(lettre ( 0-n lettreOuChiffre))
```

2.2.4 Les bruits : commentaires et séparateurs

Nous appelons *bruits* d'un langage, les suites de caractères qui peuvent apparaître entre n'importe quelle unité lexicographique sans pour autant être mentionnées dans l'arbre représentatif du document. Par exemple, les commentaires apparaissant à n'importe quel endroit du programme ne gênent pas l'analyse syntaxique et ne sont pas mis dans l'arbre représentatif.

Généralement, une expression régulière suffit pour représenter les bruits d'un langage. Dans ce cas, on utilise la notation des constructions atomiques qui est décrite dans la suite (cf. § 2.3.4) pour remplir la rubrique *bruits* de la description du langage (cf. Fig. 2.1). Par exemple, en Pascal, le bruit est constitué des caractères espace, tabulation, fin de ligne, ainsi que des commentaires :

```
( bruits
    ( atomique
        ( 0-n ( ou caractereSpecial commentaire))))
```

Si une expression régulière ne suffit pas pour décrire le bruit d'un langage, on peut utiliser n'importe quel autre type de construction. On peut ainsi prendre en compte les bruits plus complexes comme ceux du langage C par exemple. Dans ce dernier cas, les options de compilation constituent à elles seules un micro-langage, possédant en particulier des expressions conditionnelles et des affectations.

2.2.5 Paragraphage par défaut

La rubrique *paragraphage* (cf. Fig. 2.1) contient l'action de paragraphage par défaut à exécuter entre les unités lexicales du langage, ainsi que la valeur par défaut pour l'indentation, un nombre entier. La description des actions de paragraphage est indissociable de la description des différentes constructions du langage. On

peut cependant préciser qu'elles se présentent sous la forme d'appels de fonctions LISP, avec ou sans arguments. Par exemple, `(b 1)` est une action de paragraphage consistant à ajouter un caractère espace, `(cr)` commence une nouvelle ligne, `(a 6)` ajoute suffisamment de caractères espace pour se positionner en colonne 6 ...

D'autres actions de paragraphage sont plus évoluées et en particulier, ne font pas que du positionnement absolu dans le texte, par exemple, l'action `(i+ 3)` commence une nouvelle ligne en décalant la marge gauche de trois caractères espace vers la droite. Lorsque, l'action `(i+)` est appelée sans argument, elle indente le texte du nombre d'espace indiqué derrière le mot clé *indentationParDefaut*. L'action par défaut, indiquée derrière le mot clé *actionParDefaut*, est exécutée entre chaque unité lexicale en cas d'absence d'une action plus spécifique dans la construction correspondante ...
Nous reviendrons sur ce point, le moment venu, en parallèle avec la description des différentes constructions du langage.

En attendant, supposons que la valeur d'indentation par défaut est 3 et que l'action à exécuter entre chaque unité lexicale consiste à ajouter un caractère espace, `(b 1)` :

```
( paragraphage
    ( actionParDefaut
        (b 1))
    ( indentationParDefaut
        3))
```

Une liste complète des différentes actions de paragraphage est donnée au §2.5.

2.3 Les constructions

Cinq catégories de constructions existent pour décrire les différentes parties d'un langage. Chaque catégorie permet d'exprimer une caractéristique particulière, répétition, choix, regroupement, sucre syntaxique ou construction atomique.

C'est à l'aide des constructions que GÉPI va choisir la structure ainsi que les différents types de nœuds qui permettent de construire l'arbre représentatif du document à manipuler. En outre, les mêmes constructions sont utilisées pour l'analyse syntaxique du document. Le choix des différentes constructions, s'il doit être réfléchi, n'est pas très complexe. L'écriture d'une description de langage s'apparente à l'écriture d'un programme ordinaire dans le sens où elle n'est jamais au point du premier coup. Pour faciliter la mise au point d'une description, il est utile de l'essayer progressivement avec GÉPI lui-même, en particulier pour observer l'effet des actions de paragraphage.

2.3.1 Construction *groupe*

Les constructions *groupe* permettent de décrire la majeure partie des éléments d'un langage, en particulier toutes les structures de contrôle y compris les affec-

```
(<nomDuGroupe>
    [( extern  <string>)]
     ( groupe  <element1> <element2> <element3> ... ))
```

Figure 2.10. La construction *groupe*.

tations et les expressions. Ces constructions (cf. Fig. 2.10) comportent une clause *groupe* pour la syntaxe abstraite et la syntaxe concrète et, optionnellement, une clause *extern* pour préciser un affichage plus convivial de la construction elle même lorsqu'elle n'est pas développée ou lorsqu'elle apparaît dans un menu de l'interface.

Une construction groupe est l'équivalent d'une règle de grammaire possédant une seule alternative. Par exemple, la règle de grammaire décrivant une affectation :

```
affectation → identificateur ":=" expression
```

est l'équivalent de la construction *groupe* suivante :

```
(affectation
    ( groupe identificateur ":=" expression))
```

La liste qui suit le mot clé *groupe* correspond à la partie droite de la règle de grammaire. Seuls les terminaux du langage sont entre doubles quotes, ce qui permet de les distinguer des noms de constructions, sans double quote.

La syntaxe abstraite est calculée à partir du contenu de la clause *groupe* en ne conservant que les noms de constructions. Ainsi, la syntaxe abstraite d'une `affectation` est un nœud à deux fils, le fils gauche, une construction `identificateur`, le fils droit, une construction `expression`. Ces deux constructions doivent être définies par ailleurs et peuvent être indifféremment des constructions *groupe* ou des constructions d'un autre type.

Paragraphage

La clause *groupe* peut aussi comporter des indications de paragraphage sous la forme d'appels de fonction LISP. Par exemple, pour la construction `boucleTantQue`, on décide d'indenter, la partie contenant le corps de la boucle en ajoutant l'action `(i+)` avant l'élément qui marque le début du corps de boucle, le terminal `"faire"` :

```
(boucleTantQue
    ( extern "tantQue...")
    ( groupe "tantQue"  condition (i+) "faire" instruction))
```

Lorsqu'aucune action de formatage n'est mentionnée entre deux éléments d'une clause groupe, l'action de formatage par défaut est appliquée. Ainsi, lors de la


```
(<nomDuChoix>
     ( choix  <cstruct1> <cstruct2> <cstruct3>... ))
```

Figure 2.11. Exprimer une sélection, la construction *choix*.

décompilation d'une boucle `"tantQue"`, l'action par défaut, `(b 1)`, est déclenchée avant la `condition` et avant l'`instruction` en plus de l'indentation demandée devant le terminal `"faire"` :

```
... tantQue condition
          faire instruction ...
```

Les actions de paragraphage prédéfinies sont présentées au § 2.5. Une d'entre elles, `(b0)`, n'a aucun effet sur le paragraphage et peut être utilisée pour éviter le déclenchement de l'action par défaut. Par exemple, pour coller le signe d'affectation à sa partie gauche :

```
(affectation
    ( extern "..:=.." )
    ( groupe identificateur (b0) ":=" expression))
```

L'utilisation de la clause *extern*, permet dans ce cas d'afficher la chaîne `"..:=.."` plutôt que le nom de la construction elle même dans les menus ou dans un texte non complètement développé. Cette clause est également utilisable dans les autres catégories de constructions.

Parties optionnelles

Une clause *groupe* peut comporter des parties optionnelles. Comme dans le cas des expressions régulières, l'opérateur *0-1* désigne une telle partie. Par exemple, définissons une construction `siAlorsSinon` dont la "partie sinon" est optionnelle :

```
(siAlorsSinon
    ( extern "si...alors...[sinon]")
    ( groupe
            "si" condition
            (i+) "alors"  instruction
            ( 0-1 (i+) partieSinon)
            "finSi"))
```

2.3.2 Construction *choix*

Une construction *choix* (cf. Fig. 2.11) exprime une possibilité de sélection parmi un ensemble de constructions. On retrouve ainsi l'équivalent des règles de grammaire avec plusieurs alternatives. Par exemple, pour exprimer le fait qu'une instruction peut être soit une affectation, soit un appel à un sous programme, soit une boucle while, on écrit :

```
(instruction
    ( choix
            affectation
            appelSousProgramme
            boucleTantQue
            boucleRepeter
            siAlorsSinon
            groupeDinstructions))
```

En cours d'édition, lorsqu'une construction choix est développée, ses différents items sont présentés à l'utilisateur sous la forme d'un menu. Chaque item est une construction définie par ailleurs avec ou sans clause *extern*. Selon le cas, le nom de la construction ou la chaîne de caractères de la clause *extern* est utilisée pour la composition du menu.

Chaque item d'une construction choix peut être de n'importe quelle catégorie, *groupe* bien sûr, mais aussi *choix* lui-même. Dans ce dernier cas, le menu possède plusieurs niveaux et ainsi de suite en cascade.

Les constructions choix n'apparaissent dans l'arbre qu'en tant que feuilles, celles-ci, en cas de développement, sont remplacées par le nœud correspondant à la construction sélectionnée.

2.3.3 Construction *liste*

Les constructions *liste* permettent de représenter la répétition de l'utilisation d'une autre construction (cf. Fig. 2.12). Les quatre premières rubriques d'un construction liste sont optionnelles. La première, précédée du mot clé *extern*, contient, comme d'habitude, la chaîne d'affichage de la construction elle même. Les trois suivantes contiennent respectivement, le marqueur indiquant le début de la liste, derrière le mot clé *tete*, l'unité lexicale servant à séparer les items de la répétition, *separateur*, et enfin le marqueur de fin de liste derrière *queue*. La seule clause obligatoire d'une construction liste, la cinquième, est celle qui permet d'indiquer la construction à répéter. Le mot clé est soit *liste1-n* si la construction doit apparaître au moins une fois, soit *liste0-n* si elle peut ne pas apparaître du tout.

Par exemple, la liste des paramètres effectifs pour une procédure Pascal commence par une parenthèse ouvrante, doit comporter au moins un argument et se termine par une parenthèse fermante. Lorsqu'il y a plusieurs arguments, il faut les séparer par une virgule :

```
(parametreEffectif
    ( extern "(...)")
    ( tete "(")
    ( separateur ",")
    ( queue ")")
    ( liste1-n expression))
```

La dernière clause d'une construction *liste* est facultative. Elle permet de chan-


```
(<nomDeLaListe>
    [( extern <string>)]
    [( tete <string>)]
    [( queue <string>)]
    [( separateur <string>)]
     ( <typeDeliste> <construction>)
    [( indent <a1> <a2> <a3> <a4> <a5> <a6> <a7> <a8>)])
```

Figure 2.12. Exprimer une répétition, la construction *liste*.

ger l'action de paragraphage par défaut, `(b 1)`, qui consiste à ajouter un caractère blanc, entre les différentes unités lexicales de la liste. Cette clause, si l'on décide de l'incorporer dans la description de la liste, doit comporter 8 actions de paragraphage qui seront exécutées respectivement : avant et après la *tete*, avant et après l'item à répéter, avant et après le *separateur* et enfin, avant et après la *queue*.

Par exemple, voici la description d'un bloc d'instructions dans lequel, les instructions sont séparées par un point virgule et chaque instruction commence exactement en dessous du marqueur de début de liste :

```
(blocDinstructions
    ( extern "debut ... fin")
    ( tete "debut")
    ( separateur ";")
    ( queue "fin")
    ( liste0-n instruction)
    ( indent () (i0)   () ()   () (i0)  (i0)()))
```

Ou encore, une liste de procédures commençant toutes en début de ligne, sans *queue*, ni *tete*, ni *separateur* :

```
(listeDeProcedures
    ( liste0-n declarationDeProcedure)
    ( indent () ()   (cr) (cr)   () ()  ()()))
```

La construction à répéter peut être de n'importe quelle catégorie. La représentation arborescente d'une construction liste est un nœud à arité variable selon le nombre de constructions portées.

Avec une grammaire ordinaire, dans le meilleur des cas, la notion de répétition peut être exprimée par deux productions dont une est récursive et une autre dérive sur le symbole vide ($\wedge$) :

```
listeDeProcedures → procedure listeDeProcedures
listeDeProcedures → ∧
```

Pour exprimer la même structure syntaxique, la récursivité peut être *gauche* comme dans l'exemple précédant, ou *droite* :

```
(nomAtomique
    [( extern <string>)]
    ( atomique <expression régulière>))
```

Figure 2.13. Les terminaux génériques, la construction *atomique*.

```
listeDeProcedures → listeDeProcedures procedure
listeDeProcedures → ∧
```

Lorsqu'il y a un marqueur de tête, de queue et un séparateur, le nombre de productions augmente :

```
blocDinstructions → "debut" desInstructions "fin"

desInstructions → ∧
desInstructions → instruction
desInstructions → instruction ";" desInstructions
```

La construction liste permet de systématiser la représentation des répétitions à l'aide d'une seule construction qui remplace plusieurs productions d'une grammaire ordinaire. La description est plus concise et, surtout, il n'y a pas lieu de choisir entre une récursivité droite ou gauche.

L'analyseur syntaxique de GÉPI interprète directement les constructions du langage. Un analyseur syntaxique classique ne travaille que sur de simples productions et par conséquent, l'analyseur ne sait pas si la production en cours d'analyse fait partie de la description d'une répétition ou non. En cas d'erreurs de syntaxe, il lui est difficile de faire des suppositions quant à la nature de l'erreur : s'agit-il de l'oubli d'un séparateur? de l'oubli du marqueur de fin de liste? d'une erreur dans un item? ...
L'analyseur syntaxique de GÉPI interprétant la construction liste elle-même peut facilement détecter de telles erreurs (cf. chap. 7.4). Une correction peut être effectuée automatiquement par l'analyseur, ou, plus simplement, un message d'erreur plus clair peut être envoyé à l'utilisateur.

2.3.4 Construction *atomique*

La construction *atomique* (cf. Fig. 2.13) permet de représenter les entités terminales du langage, que l'on peut décrire avec une expression régulière (cf. § 2.2.2). Les nombres et les identificateurs sont des constructions atomiques. Pour les décrire, il suffit d'indiquer l'expression régulière correspondante derrière le mot clé *atomique* :


```
(nomDeSucre
    [( extern <string>)]
    ( sucre   <element1> <element2> <element3> ... ))
```

Figure 2.14. Représentation d'un artifice syntaxique, la construction *sucre*.

```
(nombre
    ( atomique (chiffre ( 0-n chiffre))))

(identificateur
    ( extern idf)
    ( atomique (lettre ( 0-n ( ou lettre chiffre)))))
```

2.3.5 Construction *sucre*

Une construction *sucre* (cf. Fig. 2.14) permet de décrire un artifice syntaxique pour lequel on ne désire pas créer un nœud supplémentaire dans la représentation arborescente correspondante.

Par exemple, supposons que le programme principal de notre langage ait la même structure qu'un corps de procédure ordinaire, mais soit suivi d'un caractère point. La construction *groupe*, `unProgramme`, permet de décrire ce fait :

```
(unProgramme
    ( groupe bloc "."))
```

Du point de vue de la syntaxe concrète, tout se passe bien. Par contre, l'arbre représentatif d'une construction `unProgramme` est constitué d'un nœud, `unProgramme`, à un seul fils, le nœud `bloc`. Cette ramification n'apporte rien d'un point de vue visuel et peut être supprimée en utilisant une construction *sucre* :

```
(unProgramme
    ( sucre bloc "."))
```

Dans ce cas, la représentation arborescente est constituée d'un seul nœud, `unProgramme`, portant directement les fils de la construction `bloc`. L'ancienne ramification qui liait `unProgramme` au nœud `bloc` à disparu.

Comme dans le cas d'une construction *groupe*, il est possible d'indiquer des actions de paragraphage :

```
(unProgramme
    ( extern "joli programme principal"
    ( sucre bloc (cr) "."))
```

2.4 Un exemple : Mini Pascal

Cette partie récapitule, dans un exemple complet, *miniPascal*, l'emploi des différentes rubriques de description d'un langage. Si *miniPascal* n'est qu'un sous ensemble du véritable langage Pascal [Wirth 76], l'exemple présente l'intérêt de montrer la concision des descriptions. En outre, en tant que fervents défenseurs du typage et de la liaison dynamique (cf. chap. 4), c'est avec un plaisir certain que nous décrivons un langage où les indications de type sont volontairement escamotées ... Toutes nos excuses à Niklaus Wirth, le père du véritable langage Pascal [Wirth 71b].

Grâce aux multiples aspects qu'il est possible de donner aux noms de constructions, l'utilisation de la clause *extern* est souvent superflue. Si le caractère parenthèse ouvrante est interdit dans l'écriture d'un symbole, tous les autres signes, notamment ceux de l'arithmétique, peuvent être utilisés comme nom de construction. Ainsi, la description de miniPascal contient la définition des constructions *, +, -, >= ,<= ... En outre, la présence simultanée d'un terminal et d'une construction de même nom n'est pas génante puisqu'on distingue le terminal du fait qu'il est une chaîne de caractère, et le nom de construction par le fait qu'il est un symbole Le-Lisp, c'est à dire sans double quote. Par exemple, la construction `tantQue` et le terminal `"tantQue"` ont le même nom, de même, + et "+" etc. De plus, avantage inattendu, cette technique s'avère plus conviviale lors de l'utilisation de l'éditeur d'arbres généré. Les nœuds d'un arbre expression ont le même nom que les constructions qu'elles portent.

Comme dans un programme LISP, les commentaires[1], à l'intérieur d'une description, commencent par le caractère point virgule et se terminent en fin de ligne.

```
( langage miniPascal
  ( terminaux "si" "alors" "appel" "debut" "fin" "tantQue" "do"
    "odd" "const" "procedure" "var")
  ( terminauxSeparateurs "." ";" "," "+" "-" "*" ":=" "(" ")" ">"
    "<" "<>" "=" "<=" ">=" "/")
  ( abreviations
    (lettre ( ou ( tranche "a" "z") ( tranche "A" "Z")))
    (chiffre ( tranche "0" "9"))
    (commentaire ("(*" ( sauf "*)") "*)"))
    (caractereSpecial ( ou #\cr #\lf " " #\tab)))
  ( bruits
    ( atomique ( 0-n ( ou caractereSpecial commentaire))))
  ( paragraphage
     ( actionParDefaut (b1))
     ( indentationParDefaut 3))
  ( constructions ;; ----------------------------
    (unProgramme
      ( sucre bloc (a 0) "."))
    (bloc
      ( groupe ( 0-1 DclConst) (i0)
               ( 0-1 DclVars) (i0)
               ( 0-1 DclProc) (i0)
               ( 0-1 instruction) (i0)))
    (DclConst
      ( liste1-n cst)
      ( tete "const")
      ( separateur ",")
      ( queue ";")
      ( indent () (i+) () (b 1) () (i+) (i0) ()))
    (cst
      ( groupe idf "=" nombre))
    (desInsts
      ( liste1-n instruction)
      ( tete "debut")
      ( separateur ";")
      ( queue "fin")
      ( indent () (i+) () (b1) () (i+) (i0) ()))
    (DclVars
      ( liste1-n idf)
      ( tete "var")
      ( separateur ",")
      ( queue ";")
      ( indent () (i+) () (b1) () (i+) (i0) ()))
    (DclProc
      ( liste1-n procedure)
      ( indent () () (i0) (i0) () () () ()))
    (procedure
      ( groupe "procedure" idf ";" (i+) bloc))
    (instruction
      ( choix affectation appel siAlors tantQue desInsts))
    (affectation
      ( extern ":=")
      ( groupe idf ":=" expression))
    (appel
      ( groupe "appel" idf))
    (siAlors
      ( groupe "si" condition (i+) "alors" instruction))
    (tantQue
      ( groupe "tantQue" condition (i+) "do" instruction))
    (condition
      ( choix odd = <> > < >= <=))
    (odd
      ( groupe "odd" expression))
    (=
      ( groupe expression (b0) "=" (b0) expression))
    (<>
      ( groupe expression "<>" expression))
```

1. [Cointe 85, page 157] *"Nous considérons que les commentaires sont malvenus dans le texte d'un programme LISP dans lequel ils produisent un effet de bruit parasitant la lecture."*

```
(<
  ( groupe expression "<" expression))
(>
  ( groupe expression ">" expression))
(<=
  ( groupe expression "<=" expression))
(>=
  ( groupe expression ">=" expression))
(expression
  ( choix moinsUnaire + - plusUnaire terme))
(moinsUnaire
  ( extern "-")
  ( groupe "-" expression))
(+
  ( groupe terme "+" expression))
(-
  ( groupe terme "-" expression))
(plusUnaire
  ( extern "+...")
  ( sucre "+" expression))
(terme
  ( choix * / facteur))
(*
  ( groupe facteur "*" terme))
(/
  ( groupe facteur "/" terme))
(facteur
  ( choix idf nombre parenthesage))
(parenthesage
  ( extern "(expr.)" )
  ( sucre "(" expression ")" ))
(idf
  ( atomique (lettre ( 0-n ( ou lettre chiffre #/_)))))
(nombre
  ( atomique ( 1-n chiffre))))))
;;  Thats all folk...
```

2.5 Table des actions de paragraphage

La table des actions de paragraphage qui suit a évolué au fur et à mesure que nous définissions de nouveaux langages. Elle contient maintenant suffisamment d'actions différentes pour formater correctement un bon nombre de langages comme Pascal, Ada ou Eiffel par exemple.

Certaines actions permettent un positionnement absolu sur une colonne ce qui peut être utile dans certains langages conçus pour les cartes perforées comme FORTRAN ou COBOL.

La plupart des actions de paragraphage agissent relativement à la colonne où a débuté la décompilation de la construction dans laquelle l'action est utilisée. Par exemple, l'action `(i0)` termine la ligne en cours et se positionne sous la colonne de début de décompilation. Le corps d'une procédure peut ainsi être positionné sous le caractère `"p"` du mot clé procédure indépendamment du niveau d'imbrication de la procédure :

```
(procedure
  ( groupe "procedure" idf ";" (i0) bloc))
```

La table des actions de paragraphage peut être étendue sans aucune modification du décompilateur. Chaque action de paragraphage est implantée à l'aide d'une méthode de la classe `collecteur` dont le sélecteur est le nom de l'action elle même. Par exemple, l'action `(b <N>)` est implantée avec la méthode de sélecteur `b` dans la classe `collecteur`. L'ajout d'une nouvelle action est réalisé en définissant dans la classe `collecteur` une méthode ayant comme sélecteur le nom de la nouvelle action. Toutes ces méthodes ont le même nombre d'arguments et sont déclenchées automatiquement lors de la décompilation. Le sélecteur est lui-même calculé dynamiquement par le décompilateur à l'aide de la liste qui représente l'action de paragraphage. Un des arguments de la méthode de paragraphage est valué avec cette même liste. Cela permet, à l'intérieur du corps de la méthode, de récupérer les arguments d'appel de l'action de paragraphage. On peut ainsi définir de nouvelles actions de paragraphage à nombre quelconque d'arguments sans jamais modifier le décompilateur (cf. § 7.3).

ACTION		*DESCRIPTION*
`(a <N>)`	→	Avance, en insérant des caractères espaces jusqu'en colonne `<N>`. Si la colonne `<N>` est déjà dépassée, un saut de ligne est effectué.
`(b <N>)`	→	Insère `<N>` caractères espaces.
`(b0)`	→	Ne fait rien, mais plus rapidement que `(b 0)`.
`(b1)`	→	Plus rapide que `(b 1)`.
`(cr)`	→	Commence une nouvelle ligne indépendamment de la colonne origine. Equivaut à `(1 1)`.
`(c <code>)`	→	Insère le caractère de code Ascii`<code>`. On peut utiliser les macro-caractères Le-Lisp. `(c 32)` équivaut à `(b1)`.
`(i+ <N>)`	→	Commence une nouvelle ligne et se positionne `<N>` caractères après la colonne origine.
`(i+)`	→	Equivalent de `(i+ <N>)` avec `<N>` prenant la valeur indiquée dans la rubrique *indentationParDefaut* (cf. Fig. 2.1).
`(i0)`	→	Commence une nouvelle ligne et se positionne en colonne origine. Plus rapide que `(i 0)`.
`(i- <N>)`	→	Commence une nouvelle ligne et se positionne en retrait de `<N>` caractères par rapport à la colonne origine.
`(l <N>)`	→	Saute `<N>` lignes. Par exemple, `(l 3)` est plus rapide que `(cr)(cr)(cr)`.
`(s <str>)`	→	Insère la chaîne de caractères `<str>`. La notation est celle des chaînes Le-Lisp.

Figure 2.15. Table des actions de paragraphage.

3
Notions de programmation objet

" We called Smalltalk Smalltalk so that nobody would expect anything from it. "

Alan Kay [Winston 84b]

Nous présentons dans ce chapitre les aspects qui, selon nous, caractérisent la programmation objet : définition de classes, instanciation, héritage et surtout, le mécanisme d'envoi de message connu aussi sous le nom de liaison dynamique. Le vocabulaire et les notions présentées sont nécessaires pour une bonne compréhension des chapitres suivants. Le lecteur intéressé pourra se rapporter aux nombreux ouvrages existant sur le sujet [Goldberg 83] [Cox 84] [Cox 86] [Meyer 89] et bien sûr [Masini 89].

Plan

Après un bref historique (§3.1), les principaux aspects de la programmation objet sont présentés (§3.2). Puis, un exemple de programme utilisant la liaison dynamique permet d'exhiber les avantages et les inconvénients de ce mécanisme (§3.3). Nous continuons ensuite par un exercice plus difficile, une classification des langages à objets (§3.4). Enfin, le bilan de ce chapitre récapitule les problèmes et permet de mieux cibler le domaine d'application d'un langage objet (§3.5).

3.1 Bref historique

Depuis le début des années 70, la programmation objet s'est largement répandue dans le monde informatique et touche aujourd'hui le grand public grâce aux ordinateurs individuels. Le temps où les langages à objets n'existaient qu'à l'état

de prototypes cantonnés dans les laboratoires et les centres de recherche est désormais bien révolu. Loin d'être de simples phénomènes de mode, ils proposent une nouvelle approche de la programmation qui doit sa souplesse à un mariage naturel entre programmes et données. Grâce à leur puissance déclarative, à leur évolutivité et à leur grande convivialité, ils se révèlent particulièrement bien adaptés à une large gamme d'applications.

Les principes de la programmation objet sont principalement nés de deux tendances en fait relativement anciennes. D'une part la difficulté d'exploiter des programmes de taille importante a imposé la notion de *programmation structurée* [Wirth 71a] [Dahl 72] [Arsac 77]. L'analyse du problème privilégie les traitements, la tâche à effectuer est découpée en sous-programmes. D'autre part, la manipulation de volumes importants d'informations complexes a mis en évidence la nécessité de structurer et de regrouper en entités les ensembles de données partageant les mêmes caractéristiques. Ce point est particulièrement sensible dans les travaux liés aux sciences cognitives. Il a donné naissance à la programmation *dirigée par les données* [White 78] [Bobrow 83b] [Winston 88]. Cette fois, ce sont les données qui servent de fondement à l'analyse et qui déterminent la structure des programmes.

Destiné à l'origine aux traitements des problèmes de simulation, Simula fut le premier langage à marier ces deux approches. La deuxième version, Simula 67 [Birtwistle 73], reprend et clarifie les concepts de classe et d'objet introduits dans la première version [Dahl 66] qui était une extension d'ALGOL 60 [Naur 63]. Cette démarche remettait en cause l'habituelle séparation établie entre données et programmes.

Smalltalk est l'héritier direct de Simula. Il généralise la notion d'objet qui devient l'entité unique de son univers. Trois versions successives ont vu le jour, Smalltalk72 [Kay 76], Smalltalk76 [Ingalls 78] et enfin Smalltalk80 [Goldberg 83]. Le langage devient alors un véritable environnement de programmation très sophistiqué [Goldberg 84a] dont l'interactivité est essentiellement inspirée de LISP [Cointe 82] [Cointe 83]. Smalltalk a largement contribué au développement d'interfaces utilisateur perfectionnés telles que les gestionnaires d'écran et de fenêtres, l'iconisation etc., et a popularisé l'usage de la souris et des écrans haute résolution du type *bitmap*.

Smalltalk est actuellement le langage de référence lorsque l'on parle de programmation objet et, sa syntaxe étant facilement assimilable, nous retenons ce langage pour présenter les notions essentielles de la programmation objet.

3.2 Notions de programmation objet

Dans ce paragraphe les notions de base et le vocabulaire de la programmation objet sont introduits de façon progressive, en parallèle avec le développement d'un exemple, en l'occurence la gestion d'un stock. L'exemple est volontairement simpliste afin d'en faciliter la compréhension. Il ne nécessite aucune connaissance particulière, si ce n'est d'avoir déjà feuilleté le catalogue d'un magasin de vente


```
Object subClass: #Article
  instanceVariableNames:
    'reference designation prixUnitaire quantite'
    ...
prixTTC
  ↑ prixUnitaire * 1.186

ajouter: combien
      quantite← quantite + combien

retirer: combien
      quantite← quantite - combien
```

Figure 3.1. Définition de la classe `Article`, ses quatre variables, `reference`, `designation`, `prixUnitaire`, `quantite` et ses trois méthodes, `prixTTC`, `retirer:` et `ajouter:` en Smalltalk.

par correspondance.

3.2.1 Les classes

Classiquement, un programme peut être considéré comme un ensemble de données sur lesquelles agissent des procédures [Wirth 76]. Cette conception implique une nette dichotomie entre une partie statique, les données, et une partie dynamique, les procédures.

En programmation objet, un programme est essentiellement un ensemble d'entités appelées objets, auxquelles sont associées des opérations spécifiques. L'objet intègre à la fois les aspects statique et dynamique qui viennent d'être mis en évidence.

Les ensembles d'objets ayant un comportement commun sont regroupés en classes. Une classe sert de modèle pour la création des objets qui la représentent. Ce modèle décrit une structure comprenant des variables, aussi appelées champs, et des procédures, les méthodes. Les variables constituent la composante statique : ce sont les données décrivant les caractéristiques communes des objets de la classe. Les méthodes en revanche constituent la composante dynamique : ce sont les procédures décrivant les comportements communs.

Autrement dit, définir une classe d'objets revient à définir une structure de données avec des procédures pour manipuler ces données. Passons immédiatement à un exemple concret en définissant une première classe d'objets, `Article`, qui décrit un article du stock.

Dans le programme Smalltalk correspondant (Fig. 3.1), la structure de données caractérisant un article du stock est composée de quatre variables comme un enregistrement Pascal (*record*) peut être constitué de quatre champs par exemple. Ces différentes variables sont mentionnées entre apostrophes derrière le mot clé `instanceVariableNames`[1] et ne comportent aucune indication de type.

- `reference`, désigne le numéro référençant un article,
- `designation`, un texte décrivant l'article,
- `prixUnitaire`, correspond au prix unitaire hors taxes de l'article,
- et, `quantite`, donne le nombre d'articles du type décrit disponibles dans le stock.

La définition de la classe est complétée par la liste de ses méthodes, des procédures de manipulation spécifiques à la structure de données que nous venons de définir. Ces méthodes sont au nombre de trois (Fig. 3.1) :

- `prixTTC`, la méthode qui retourne le prix toutes taxes comprises d'un article. Le prix toutes taxes est obtenu en ajoutant le montant de la TVA au prix hors taxes donné par la variable `prixUnitaire`. La flèche montante indique que la valeur de cette expression est retournée par la méthode qui peut donc être comparée à une fonction.
- `retirer:`, une méthode a un argument qui permet de retirer des articles du stock. La valeur désignée par l'argument formel `combien` est soustraite à celle de la variable `quantite`. Le résultat de cette soustraction est aussitôt affecté à la variable `quantite` qui prend ainsi une nouvelle valeur.
- `ajouter:`, une méthode a un argument qui ajoute des articles dans le stock. Cette fois, la variable `quantite` est incrémentée avec la valeur désignée par l'argument `combien`.

On remarquera que les variables d'un objet sont rémanentes : une fois modifiées, elles conservent leur nouvelle valeur. Dans le cas contraire, les opérations de mise à jour du stock, effectuées par les méthodes `ajouter:` et `retirer:`, n'auraient aucun sens.

3.2.2 Instanciation

La classe est l'entité conceptuelle qui décrit l'objet. Sa définition sert en quelque sorte de moule pour construire ses représentants physiques appelés instances.

Une instance est donc un objet particulier qui a été créé en respectant les plans de construction de sa classe. Elle possède les mêmes variables et les même

1. En programmation objet, le terme *variable* est habituellement utilisé comme une abréviation du terme *variable d'instance*, plus complet.


```
pantalons ← Article
                reference: 25
                designation: 'pantalon tergal'
                prixUnitaire: 100.00
                quantite: 1000.
...
congelateurs ← Article
                reference: 450
                designation: 'congelateur de 200 litres'
                prixUnitaire: 3295.95
                quantite: 100.
...
televiseurs ← Article
                reference: 747
                designation: 'televiseur portable'
                prixUnitaire: 2000.00
                quantite: 200.
```

Figure 3.2. Instanciation de la classe `Article` en Smalltalk. Les nouveaux objets possèdent des valeurs particulières pour chacune des variables d'instance.

méthodes que les autres instances de la classe, les variables prenant cependant les valeurs correspondant à la nature particulière de l'entité ainsi représentée. Par exemple, dans la figure 3.2, trois nouvelles instances sont créées. La première instance créée est affectée à la variable `pantalon` qui représente les pantalons de tergal de notre stock. De même, les deux autres instances créées représentent respectivement les congélateurs en stock, `congelateurs`, et les téléviseurs en stock, `televiseurs`. Pour chaque instance, les variables `designation`, `prixUnitaire` et `quantite` prennent des valeurs particulières.

A l'aide des instances nouvellement créées, il est maintenant possible de déclencher les méthodes de la classe `Article` qui sont spécialisées dans la manipulation de telles instances. La notation qui permet de déclencher une méthode en Smalltalk, contrairement par exemple à la notation d'appel des procédures en Pascal, n'est pas une notation parenthésée. Lorsque, comme pour la méthode `prixTTC`, aucun argument mis à part l'instance concernée elle même n'est nécessaire, il suffit de mettre le nom de la méthode derrière l'expression qui référence l'instance. L'exemple suivant montre la notation utilisée pour déclencher la méthode `prixTTC` avec l'instance désignée par la variable `pantalons` :

```
pantalons prixTTC
" La valeur de cette expression est 118.6"
```

Dans le cas d'une méthode avec un argument supplémentaire, comme `retirer:`, l'argument est mis à la suite du nom de la méthode. La notation qui permet de décrémenter le stock de pantalons de 5 éléments est la suivante :

```
pantalons retirer: 5
" Modifie la variable quantite de l'instance référencée par pantalons."
```

Il est important de remarquer dès à présent qu'une méthode est différente d'une procédure dans le sens où il est impossible de déclencher une méthode sans mentionner l'instance concernée par l'opération. En quelque sorte, une méthode est une procédure ou une fonction possédant au minimum un argument, l'objet concerné par le déclenchement de la procédure ou de la fonction.

3.2.3 Héritage

Bien entendu, il est possible de gérer le stock avec la seule classe créée pour l'instant, **Article**, dont chaque catégorie d'article deviendrait une instance particulière. Cette technique est cependant extrêmement lourde car elle ne permet pas de représenter facilement les spécificités de chaque catégorie d'article, que ce soit au niveau des variables ou au niveau des méthodes. Par exemple, une variable `taille` est nécessaire pour enregistrer la taille d'un vêtement mais est superflue pour un congélateur. Dans le même ordre d'idées, la méthode `prixTTC` ne convient pas pour tous les articles, les produits de luxe étant soumis à une TVA plus élevée que les produits de consommation courante.

Le problème est résolu par la définition de nouvelles classes, plus spécifiques, pouvant réutiliser la classe **Article**, existante. Une classe qui réutilise les définitions d'une classe existante est appelée sous-classe. Les sous-classes permettent d'introduire de nouvelles variables et/ou de nouvelles méthodes qui représentent les caractéristiques propres aux nouveaux objets décrits. La classe mère, c'est à dire la classe de rattachement, est habituellement appelée la superclasse.

Dans la plupart des langages à objets, les classes sont organisées hiérachiquement en une arborescence, la racine correspondant à une classe générale dont toutes les autres classes sont issues. Cette organisation rappelle celles qui sont utilisées en taxonomie[2] ou en biologie, par exemple pour la classification du règne animal. Les caractéristiques des classes supérieures sont héritées par les classes inférieures : tout se passe comme si les informations des superclasses étaient recopiées dans les sous-classes qui leur sont associées. Les variables et les méthodes de la superclasse sont adjointes aux variables et aux méthodes de chaque sous-classe. Cette recopie est cependant virtuelle, les duplications inutiles étant évitées.

Revenons à notre exemple. Pour régler le problème de la taille des vêtements, il suffit de définir une classe **Vetement**, sous-classe de **Article**, intégrant une variable `taille` supplémentaire (Fig. 3.3).

Nul besoin de méthode supplémentaire pour cet exemple particulier. Les variables de la superclasse **Article** sont héritées par la classe **Vetement**. Une instance de la classe **Vetement** possède donc cinq variables :

- la variable **taille** définie dans la classe **Vetement**,

2. ou taxinomie : sciences des lois de la classification (Larousse)


```
Article subClass: #Vetement
  instanceVariableNames:
    'taille'
    ...
```

Figure 3.3. Définition de la classe **Vetement**, sous-classe de la classe **Article** en Smalltalk. La classe **Vetement** possède une variable supplémentaire, **taille**.

- les variables **reference**, **designation**, **prixUnitaire** et **quantite**, héritées de la superclasse **Article**.

On peut ainsi instancier la classe **Vetement** en précisant cette fois la taille du stock de vêtements représenté :

```
pantalons ← Vetement
                reference: 25
                designation: 'pantalon tergal'
                prixUnitaire: 100.00
                quantite: 1000
                taille: 44.
```

De la même façon, les méthodes définies pour la classe Article sont partagées par ses sous-classes. C'est ainsi que la méthode `prixTTC` permet de calculer le prix toutes taxes comprises d'une instance de la classe **Vetement** :

```
pantalons prixTTC
" La valeur de cette expression est 118.6"
```

En respectant le même principe, les produits de luxe sont décrits par la sous-classe **ArticleDeLuxe**. Cependant, cette classe redéfinit la méthode `prixTTC` pour prendre en compte le taux de TVA propre à cette catégorie d'articles (Fig. 3.4). La méthode `prixTTC` de la classe ArticleDeLuxe calcule le prix toutes taxes comprises en ajoutant 33% au prix hors taxes de l'article. La méthode de même nom définie dans la superclasse **Article** est masquée par cette nouvelle définition.

Ainsi, le calcul de la TVA peut être modifié pour les instances de la classe **ArticleDeLuxe** :

```
caviar ← ArticleDeLuxe
                reference: 12
                designation: 'œufs d'esturgeon salés'
                prixUnitaire: 100.00
                quantite: 2.
...
caviar prixTTC
" La valeur de cette expression est 133.00"
```

La structure hiérachique des classes constitue ce qu'il est coutume d'appeler *arbre d'héritage*. Dans notre exemple, trois classes ont été définies, **Article**, puis

```
Article subClass: #ArticleDeLuxe
  instanceVariableNames:
    ' '
    ...
prixTTC
  ↑ prixUnitaire * 1.33
```

Figure 3.4. Définition de la classe `ArticleDeLuxe`, sous-classe de la classe `Article` en Smalltalk. La méthode `prixTTC`, spécifique aux articles de luxe, est redéfinie localement dans la classe `ArticleDeLuxe`.

`Vetement` et `ArticleDeLuxe`, les deux dernières étant des sous-classes de la première. La classe `Article` possède, il est temps de l'avouer maintenant, une superclasse, `Object` (Fig. 3.1). Cette classe est prédéfinie en Smalltalk et ne contient que des définitions de méthodes d'intérêt général , héritées par les sous-classes. Ainsi, l'arbre d'héritage de notre application est constitué de quatre classes, la classe `Object` étant la racine de cet arbre (Fig. 3.5).

Certains langages autorisent l'héritage multiple. Une classe peut alors hériter de plusieurs classes sans que ces dernières ne soient liées par des relations hiérarchiques, et l'ensemble des classes n'est plus structuré en arborescence, mais en graphe. Il devient ainsi possible de décrire des objets plus complexes comme étant composés d'objets plus simples. L'héritage multiple pose de nombreux problèmes sur lesquels nous reviendrons ultérieurement (chap. 5) et qui, à notre avis, sont moins importants que le mécanisme dont la description suit.

3.2.4 La liaison dynamique

La *liaison dynamique* est, selon nous, le mécanisme qui caractérise le mieux les langages à objets. Ce mécanisme est également connu sous le nom de *transmission de message*. Cette différence de terminologie n'est justifiée que par le fait que l'on distingue actuellement deux techniques d'implantation du déclenchement des méthodes. Pour notre part, ne prenant en compte que l'effet et non la façon d'y arriver, nous utiliserons indifféremment les deux appellations.

Le mécanisme de liaison dynamique repose sur un principe essentiel des langages à objets : une même variable peut, au cours du temps, référencer des instances issues de classes différentes. Si l'on assimile le type d'une variable au nom de la classe de l'instance qu'elle référence, on peut dire que les variables d'un langage à objets ne sont pas typées statiquement. Le langage Smalltalk applique ce principe de façon générale et, aucune mention de type n'est attachée de façon statique à une variable. Par exemple, dans le programme de la figure 3.6, après

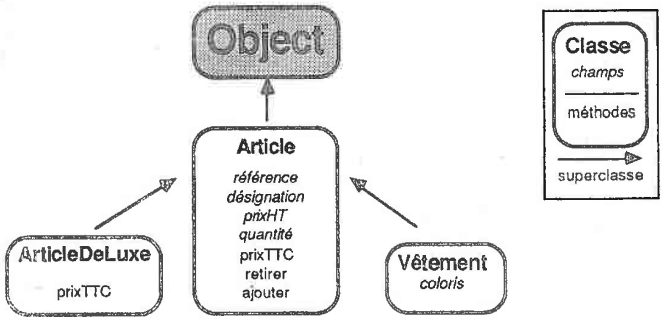


Figure 3.5. Un arbre d'héritage simple, les classes `Object`, `Article`, `ArticleDeLuxe` et `Vetement`.

l'exécution de la conditionnelle, la variable `yZt` référence soit une instance de la classe `Vetement` soit une instance de la classe `ArticleDeLuxe`.

Si les variables de Smalltalk ne sont pas typées statiquement, elles sont en revanche typées dynamiquement, c'est-à-dire à l'exécution. Lorsqu'une méthode est déclenchée, la classe d'appartenance de l'instance référencée détermine dynamiquement le choix de cette méthode. Ainsi, dans l'exemple précédent, le mécanisme de liaison dynamique a pour effet de provoquer le déclenchement de la méthode `prixTTC` définie dans la classe `ArticleDeLuxe` si la variable `yZt` référence une instance de cette classe. Si la variable `yZt` référence une instance de la classe `Vetement`, c'est la méthode `prixTTC` de la classe `Article` qui est déclenchée. Cette dernière méthode étant bien entendu héritée par la classe `Vetement`.

Vocabulaire

Dans la terminologie objet, l'action qui consiste à déclencher une méthode est appelée *envoi de message*. L'instance concernée est appelée objet *receveur* ou, plus simplement, le receveur. Le nom de la méthode est, pour sa part, appelé *sélecteur*. Voici un envoi de message de sélecteur `prixTTC` et où le receveur est désigné par la variable `yZt` :

```
yZt prixTTC
```

3.3 Programmer avec la liaison dynamique

Comme nous l'avons vu précédemment, le mécanisme de liaison dynamique va de pair avec le fait qu'une variable peut référencer des instances de natures

```
...
uneConditionCompliquee
    ifTrue:[yZt ← Vetement
                    reference: 25
                    designation: 'kimono coton'
                    prixUnitaire: 750.00
                    quantite: 1000
                    taille: 190]
    ifFalse:[yZt ← ArticleDeLuxe
                    reference: 27
                    designation: 'diamant "Le Régent"'
                    prixUnitaire: 135.00
                    quantite: 1].
    ↑ yZt prixTTC
    "La valeur de cette expression est ???"
```

Figure 3.6. La liaison dynamique : le principe essentiel des langages à objets. La variable `yZt` peut désigner indifféremment une instance de la classe `Vetement` ou de la classe `Article`. Selon le cas, c'est la méthode `prixTTC` de la classe `Article` ou celle de la classe `Vetement` qui est déclenchée.

différentes : il n'est pas nécessaire de typer les variables que l'on utilise. A priori, pour un programmeur habitué à un langage comme Pascal, le fait de ne pas typer les variables que l'on manipule peut paraître dangereux. Le mécanisme de liaison dynamique, s'il a des inconvénients, possède aussi de nombreux avantages.

3.3.1 Où sont les types ?

Le terme "*langage fortement typé*" ou "*langage à typage fort*" est habituellement employé lorsque l'on parle d'un langage dont le typage est fortement contrôlé de façon *statique*, le mot statique étant sous-entendu. Ada ou Pascal sont deux exemples de langages fortement typés statiquement. Dans ces langages, chaque variable, chaque argument de procédure, chaque expression ou chaque résultat de fonction est soumis statiquement, c'est-à-dire avant l'exécution, à une vérification de cohérence de type [Fairley 85, page 230]. Par exemple, une variable déclarée de type entier ne peut pas être affectée avec le résultat d'une expression de type différent. Ce mécanisme permet ainsi de détecter statiquement certaines erreurs.

En Smalltalk, aucune vérification statique de respect de type n'est effectuée : le langage n'est pas typé statiquement. Une variable peut être affectée avec le résultat de n'importe quelle expression. Statiquement, mis à part les erreurs de syntaxe, très peu d'erreurs sont détectées. Si le langage Smalltalk n'est pas typé

statiquement, il est en revanche fortement typé dynamiquement. A l'exécution, le langage assure qu'une méthode est toujours déclenchée correctement en regard de la classe d'appartenance du receveur.

La suppression du typage statique rend difficile, voire impossible, la vérification des programmes avant l'exécution. En contrepartie, les structures de données que l'on peut définir sont beaucoup plus souples et permettent de modéliser plus simplement les entités du monde réel en permettant la manipulation de structures hétérogènes.

3.3.2 Structures de données hétérogènes

Le typage statique d'un langage comme Ada rend impossible la définition d'une structure de données composites contenant une suite d'éléments de types différents : impossible de définir une liste, une table ou un fichier contenant des objets de types différents (§4.1.3). Ces structures hétérogènes, plus souples, sont cependant pratiques dans de nombreux cas, principalement lorsqu'il s'agit de modéliser le monde réel. Les objets qui nous entourent sont rarement uniformément composés d'entités identiques. Une corbeille à papier peut contenir autre chose que du papier. On range ce que l'on veut dans une armoire. Les voitures rouges ne sont pas les seules à pouvoir se garer dans les parkings...

Même parmi les exemples plus informatiques, la nécessité de structures composites hétérogènes trouve sa place. En compilation, pour représenter un arbre abstrait, il est intéressant de pouvoir manipuler un arbre dont les nœuds sont de natures différentes, chaque nœud représentant un opérateur de la grammaire abstraite. Dans le même ordre d'idées on peut par exemple vouloir disposer d'une pile pour évaluer des expressions, cette pile pouvant contenir les opérandes comme les opérateurs. Sans sortir de notre exemple de gestion du stock, la représentation d'une commande passe naturellement par la création d'une liste d'articles différents, articles ordinaires, articles de luxe, vêtements....

De ce point de vue, le langage LISP, qui permet la manipulation de listes hétérogènes, s'apparente à Smalltalk. L'équivalent d'une liste LISP peut d'ailleurs s'écrire très simplement en Smalltalk (Fig. 3.7). La classe `Cons` est définie avec deux variables, `car` pour désigner le premier élément de la liste et `cdr` le lien de chaînage vers l'élément suivant.

La classe `Commande` peut être définie comme étant composée de quatre variables (Fig. 3.8). La variable `client` référence un objet représentant le client qui est à l'origine de la commande. Comme d'habitude, aucune mention de type n'est précisée, et, cette variable peut désigner une simple chaîne de caractères ou une instance d'une classe définie par ailleurs, permettant de représenter un client. Les variables `dateDeCommande` et `dateDelivraison` désignent respectivement la date de commande et la date de livraison de la commande. Comme pour la variable client, aucun type n'est précisé. La quatrième variable, `listeDarticles` représente la liste des articles commandés. Cette liste peut être implantée à l'aide d'une instance de la classe `Cons`, chacun des éléments de cette liste étant lui-même instance

```
Object subClass: #Cons
  instanceVariableNames:
    'car cdr'
    ...
premier "Retourne le premier élément."
  ↑ car

reste "Retourne la liste amputée du premier élément."
  ↑ cdr
```

Figure 3.7. Définition de l'équivalent d'une liste LISP, en Smalltalk. Une instance de la classe **Cons** représente une cellule cons. Une liste chaînée construite à partir d'instances de cette classe peut contenir des éléments de différentes natures.

d'une classe d'article, **Article**, **ArticleDeLuxe** ou **Vetement**.

La méthode **totalCommandeTTC**, chargée de calculer le prix toutes taxes comprises de l'ensemble de la commande peut également être définie dans la classe commande. L'algorithme mis en œuvre est fort simple puisqu'il consiste à parcourir la liste d'articles en totalisant progressivement les prix toutes taxes des différents articles (Fig. 3.8). Pour réaliser cet algorithme, la méthode **totalCommandeTTC** comporte deux variables locales. La variable **totalTTC** qui sert à cumuler progressivement les différents prix TTC est initialisée avec 0 avant de commencer le parcours de la liste d'articles. La variable **liste** qui permet de progresser dans la liste chaînée d'articles est initialisée avec la valeur de la variable **listeDarticles**. Dans le corps de la boucle *while*, la variable **totalTTC** est incrémentée du prix toutes taxes de l'élément courant. Après avoir parcouru entièrement la liste d'articles, la valeur contenue dans la variable **totalTTC** est retournée par la méthode **totalCommandeTTC**.

Grâce au mécanisme de liaison dynamique, le prix TTC des différents articles est correctement calculé en regard de la classe d'appartenance des différentes instances de cette liste. Lorsqu'il s'agit d'un article ordinaire ou d'un vêtement, la méthode définie dans la classe **Article** est automatiquement utilisée. Pour les instances de la classe **ArticleDeLuxe**, c'est la méthode spécifique à cette classe qui est déclenchée.

3.3.3 Évolutivité

Le mécanisme de liaison dynamique permet d'implanter un algorithme de façon générale, en limitant au minimum la connaissance de l'algorithme sur la structure de données qu'il manipule. Par exemple, l'algorithme consistant à parcourir un


```
Object subClass: #Commande
  instanceVariableNames:
    'client dateDeCommande dateDeLivraison listeDarticles'
    ...

totalCommandeTTC
  | totalTTC liste |
    totalTTC ← 0.
    liste ← listeDarticles.
    [liste isNil]
      whileFalse: [totalTTC ← totalTTC + liste premier prixTTC.
                   liste ← reste liste].
    ↑ totalTTC
```

Figure 3.8. Définition de la classe **Commande** et de la méthode **totalCommandeTTC** qui totalise les prix toutes taxes comprises des différents articles.

arbre ne dépend que de la structure de l'arbre manipulé. La nature des valeurs portées par l'arbre n'influe que sur l'opération qui est effectuée à chaque point du parcours. De même, sur une pile, l'action d'empilement peut être décrite sans faire de suppositions sur la nature des objets empilés.

L'algorithme de la méthode **totalCommandeTTC** est écrit indépendamment de la définition des différentes classes d'articles, la seule supposition faite étant que chaque élément de la liste d'articles soit capable de répondre au message **prixTTC**. Sans modifier cette méthode, il est possible de faire évoluer notre application de gestion du stock en définissant une nouvelle classe d'articles modélisant par exemple les articles détaxés ou les articles abîmés, le mécanisme de liaison dynamique se chargeant de déclencher la méthode **prixTTC** adaptée à l'instance traitée. Après l'adjonction d'une nouvelle classe d'article, la méthode **totalCommandeTTC** reste valable.

En outre, il est intéressant de remarquer que le mécanisme de liaison dynamique évite souvent l'écriture de tests, la nature de l'objet receveur guidant automatiquement le choix de la méthode. Par exemple, pour l'ensemble de l'application de gestion du stock, il n'est jamais nécessaire de tester la nature d'un article. Le simple fait d'envoyer un message évite d'écrire explicitement un test dont le but est de savoir s'il s'agit d'un article ordinaire, d'un article de luxe ou d'un vêtement...

Dans le cas de programme devant manipuler des objets de natures différentes, la diminution du nombre de tests à effectuer est un avantage certain. Le programme est moins complexe à écrire, chaque classe correspondant au traitement d'un cas bien précis sur lequel on peut se concentrer en faisant abstraction des autres cas.

Par exemple, lorsque l'on écrit la méthode `prixTTC` dans la classe `ArticleDeLuxe`, on peut focaliser son intérêt sur cette seule classe. La diminution du nombre de tests, outre l'avantage de réduire la taille du code, permet de faciliter la phase de tests unitaires d'un programme. Une classe est testée séparément et, la diminution du nombre d'instructions conditionnelles réduit la taille du jeu d'essai qui permet d'essayer toutes les branches du programme.

3.3.4 Les erreurs

La méthode `totalCommandeTTC` suppose que la liste d'articles d'une commande contient effectivement des instances des classes `Article`, `ArticleDeLuxe` et `Vetement`. Si cette liste contient une instance d'une classe ne comportant pas de méthode `prixTTC`, une erreur est déclenchée à l'exécution, pendant le parcours de la liste. En Smalltalk, aucune vérification statique ne peut empêcher ce genre d'erreur.

Dans un langage objet qui, comme Smalltalk, applique systématiquement le principe de liaison dynamique, cette erreur est *a priori* la seule erreur signalable par le système. Elle se produit à chaque fois qu'un objet ne sait pas répondre à un message ou, autrement dit, à chaque fois que la classe d'appartenance de l'objet receveur ne comporte pas de définition correspondant au sélecteur du message.

Dans le cas de la méthode `prixTTC`, une solution simpliste permettant d'éviter l'erreur consiste à définir dans la classe `Object` une méthode `prixTTC` qui retourne invariablement la valeur `0`. De cette manière, toute instance est capable de répondre au message `prixTTC` car toute classe hérite en dernier lieu des méthodes de la classe `Object`. Si cette technique permet d'éviter l'erreur, elle ne fait que contourner la véritable cause du problème lié à la présence dans la liste d'articles d'une instance qui ne sait pas répondre au message `prixTTC`. L'erreur évitée pendant l'exécution de la méthode `totalCommandeTTC` à toutes les chances de réapparaître pendant l'exécution d'une autre méthode. Cette technique est à proscrire.

Le système Smalltalk fournit un mécanisme permettant de prendre en compte ce genre d'erreur. Lorsque la recherche de sélecteur provoquée par un envoi de message échoue, le système envoie un second message à l'objet receveur. Ce message a pour sélecteur **`doesNotUnderstand`** et pour arguments le sélecteur et les arguments du premier message. La recherche du sélecteur **`doesNotUnderstand`** dans le graphe d'héritage suit donc le même chemin que la recherche du premier sélecteur. Le programmeur a toute latitude pour définir dans les classes visitées des méthodes portant ce nom, pour personnaliser ou récupérer les erreurs. S'il ne le fait pas, la méthode standard **`doesNotUnderstand`**, définie dans la classe `Object`, signale de toute façon l'erreur. Si ce mécanisme permet de traiter plus proprement l'échec d'un envoi de message, il s'apparente au mécanisme d'exceptions du langage Ada dans le sens où il faut avoir prévu l'erreur a priori, c'est à dire avant l'exécution... En Smalltalk comme en Ada, une exception non captée se solde par une erreur fatale.

3.4 Une classification des langages à objets

La programmation objet est une technique de programmation fondée sur l'emploi du concept d'objet. Il est possible d'adopter ce style avec pratiquement tous les langages, de FORTRAN à LISP, mais seuls certains d'entre eux intègrent de façon standard les principes permettant de programmer directement avec des objets. Un langage à objets est donc un langage qui fournit toutes les facilités pour programmer dans un style objet.

Un certain nombre de spécialistes [Stroustrup 87b] [Wegner 87a] [Wegner 87b] s'accordent à distinguer :

- les langages offrant uniquement les propriétés d'abstraction de données et d'encapsulation, tels que Ada [Ada 83] ou Modula-2 [Wirth 84] : en Ada par exemple, des données et les procédures les manipulant *peuvent* être regroupées dans un *package* ; ceci n'est toutefois en aucun cas une obligation et le programme peut être structuré autrement qu'en termes de classes ;
- les langages qui regroupent les objets en *classes*, comme CLU [Liskov 77] : à l'inverse de Ada, ce langage *oblige* le programmeur à structurer son programme en classes appelées *clusters* ;
- les "vrais" langages à objets, qui intégrent en plus la notion d'*héritage*.

Si cette définition permet déjà de restreindre l'ensemble des langages que nous qualifierons de langages à objets, cet ensemble reste cependant très disparate, tant du point de vue des concepts mis en œuvre que de leur implantation physique. Trois grandes familles de langages apparaissent naturellement, chacune privilégiant un point de vue sur la notion d'objet [Ferber 88] :

- Point de vue *structurel* : l'objet est un type de données, qui définit un modèle pour la structure de ses représentants physiques et un ensemble d'opérations applicables à cette structure.
- Point de vue *conceptuel* : l'objet est une unité de connaissance, représentant le prototype d'un concept.
- Point de vue *acteur* : l'objet est une entité autonome et active, qui se reproduit par copie.

3.4.1 Les langages de classes

Les langages appartenant à la première catégorie sont appelés *langages de classes*. La classe est une généralisation de la notion de type (au sens Pascal par exemple) et décrit un ensemble d'objets partageant la même structure et le même comportement. Elle sert de moule pour créer ses représentants physiques, ou instances.

Une classe peut avoir certaines caractéristiques en commun avec une ou plusieurs autres classes. Leur ensemble forme ainsi une hiérarchie dans laquelle chaque

sous-classe partage les connaissances des classes dont elle est une spécialisation. On dit qu'elle *hérite* des méthodes et des données de ses *superclasses*.

Simula [Birtwistle 73] et Smalltalk-80 [Goldberg 83] sont les chefs de file historiques de cette famille. Simula est le premier vrai LO mais, pour nous, l'archétype même du LO reste Smalltalk-80, car son univers est complètement uniforme : la seule entité est l'objet et la seule structure de contrôle est l'envoi de message, qui permet de communiquer avec les objets. Plus qu'une simple description, une classe Smalltalk-80 est elle-même un objet, instance d'une autre classe appelée sa *métaclasse*.

3.4.2 Les langages de frames

Les *langages de frames*, qui constituent la deuxième famille, sont issus des travaux sur la représentation des connaissances. Ce sont les héritiers des *réseaux sémantiques* [Quillian 68], formalisme de représentation inspiré des travaux en psychologie cognitive.

Un réseau sémantique est un graphe où les nœuds sont typés et représentent des unités de connaissance élémentaire. Pour élargir et mieux structurer la connaissance liée aux nœuds, Marvin Minsky a introduit la notion de *frame* [Minsky 75], objet prototype[3] décrivant une situation standard. Un frame possède des *attributs* dont les différents aspects sont décrits par des *facettes* déclaratives ou procédurales. Les premières associent des valeurs aux attributs, alors que les secondes décrivent des procédures appelées *réflexes*, activées lors des accès aux valeurs des attributs. Un frame n'a pas de comportement propre décrit par des méthodes : ce sont les attributs qui détiennent les actions. Comme les classes, les frames s'inscrivent dans une hiérarchie d'héritage, sans toujours faire de distinction entre les prototypes et leurs représentants.

L'article de Marvin Minsky a motivé une série de travaux qui ont abouti à des langages se réclamant tous des frames, mais ayant des caractéristiques très diverses. Pour notre part, nous appelons langage de frames tout langage dont l'unité de base est le triplet (frame, attribut, facette). Un tel langage est avant tout un outil pour gérer des bases de connaissances. L'envoi de message ne fait pas partie de ses structures de contrôle, mais des mécanismes de filtrage et de classification lui sont souvent associés.

3.4.3 Les langages d'acteurs

La troisième famille regroupe les *langages d'acteurs* issus des travaux menés au Massachusetts Institute of Technology (MIT) par Carl Hewitt [Hewitt 73]. A l'origine, le modèle acteur a été développé pour représenter des connaissances sous la forme d'une société d'experts coopérant les uns avec les autres.

Un *acteur* est un objet actif qui communique avec ses semblables par envois

3. premier d'une série.

de messages asynchrones. Son comportement définit la manière dont il réagit aux événements provoqués par les autres acteurs. Contrairement aux classes, les acteurs ne sont pas organisés hiérarchiquement, mais en réseaux d'*accointances* : chaque acteur est en relation avec un ensemble d'autres acteurs, qui évolue dynamiquement. Il leur *délègue* les messages qu'il ne peut pas traiter, réalisant ainsi l'équivalent du mécanisme d'héritage [Lieberman 86] [Stein 88].

Un acteur peut créer d'autres acteurs par copie de lui-même. Cette copie est soit rigoureusement identique, soit *différentielle*, c'est-à-dire spécialisée par adjonction de nouvelles caractéristiques. Elle joue le rôle de mécanisme d'instanciation.

Par son côté dynamique, le modèle proposé par les langages d'acteurs se prête particulièrement bien à la programmation *parallèle* et à la mise en œuvre de systèmes répartis sur un réseau de processeurs [Agha 86].

3.4.4 Les langages hybrides

S'il existe des représentants "purs" de chacune des trois familles présentées, de nombreux langages intègrent des caractéristiques empruntées à l'une et à l'autre et sont regroupés dans une quatrième famille, celle des langages ou systèmes *hybrides*.

La plupart du temps, la conception de ces systèmes est motivée par des considérations pratiques, particulièrement sensibles en intelligence artificielle. Il est difficile de représenter dans un formalisme objet des connaissances qui se traduisent naturellement en formules logiques ou encore en règles de production. Plutôt que de rester prisonnier d'un formalisme unique, il est pratique de faire coexister dans un même système plusieurs styles de programmation.

Typiquement, les langages hybrides associent des méthodes à des frames et possèdent souvent une composante de programmation logique. Deux écoles se sont développées autour des deux langages favoris de l'intelligence artificielle, LISP et Prolog, afin d'y intégrer les concepts de la programmation objet. LOOPS [Bobrow 83a] a été un précurseur dans ce domaine et constitue le chef de file de l'école LISP. L'école Prolog est surtout liée au projet japonais de machines de cinquième génération [Albert 85].

3.5 Premier bilan

La classification des langages à objets donnée précédemment (§3.4) ayant peut-être embrouillé les idées d'un novice en programmation objet, il est temps de dresser un premier bilan sur les apports essentiels de ce style de programmation. Notre expérience de programmeurs se limitant principalement à la catégorie des langages de classes (3.4.1), c'est uniquement selon ce point de vue que nous dressons ce bilan.

S'il est clair que les langages de classes offrent toutes les propriétés d'abstraction de données et d'encapsulation, il nous semble important d'insister sur le mécanisme d'héritage et sur le mécanisme de liaison dynamique. La possibilité de réutilisa-

tion offerte par l'héritage, couplée au mécanisme de liaison dynamique permet la manipulation de structures de données hétérogènes. Ces deux aspects de la programmation objet offrent une grande souplesse aussi bien du point de vue de l'écriture des algorithmes qui peuvent être écrits de façon générale que du point de vue des structures de données qui ne sont pas contraintes statiquement.

L'absence de typage statique limite considérablement les possibilités de vérification avant exécution. On ne peut se convaincre de la validité d'un programme qu'en le relisant et surtout en l'essayant. Notons qu'il en est de même pour un programme Ada par exemple. Même s'il est possible d'effectuer certaines vérifications statiques, ces vérifications ne garantissent en aucun cas la correction du programme. En outre, ces vérifications ne garantissent pas non plus que le programme ne provoquera pas d'erreur fatale à l'exécution. Un programme Ada peut en effet s'arrêter brutalement pour de nombreuses raisons, débordement d'une table, non respect du type d'une variable contrainte, exception non captée ou encore accès incohérent dans une structure à discriminant (page 84), etc.

Dans le domaine particulier du prototypage, une relative insécurité des programmes est tolérable surtout lorsque le langage apporte plus de souplesse pour la représentation des données et permet une évolution rapide des programmes. De ce point de vue, Smalltalk est un exellent outil de prototypage [Bezivin 86a].

En ce qui concerne la sécurité des programmes, l'approche de la programmation objet par les langages de l'école scandinave comme Simula [Birtwistle 73], C++ [Stroustrup 86], Eiffel [Meyer 86c] ou Object Pascal [Schmucker 86b] est intéressante. Ces langages font un compromis entre le typage statique fort d'un langage comme Ada et l'absence de typage statique de Smalltalk. Le principe consiste à contraindre statiquement une variable en l'obligeant à ne manipuler que des instances d'une partie du graphe d'héritage. Par exemple, il est possible de limiter statiquement une variable pour qu'elle ne puisse référencer que des instances de la classe `Article` et de ses sous-classes. Pour réaliser cela en Eiffel par exemple, la variable est déclarée de type `Article`. En désignant cette classe, on désigne aussi ses sous-classes. La variable possède ainsi un certain degré de liberté qui permet de tirer partie du mécanisme de liaison dynamique. Le même principe est utilisé pour typer les arguments et l'éventuel résultat des méthodes. Chaque expression du langage possède ainsi un type statique sous la forme d'un nom de classe. La cohérence des affectations et des envois de messages peut ainsi être vérifiée.

Ce chapitre n'est pas une véritable présentation du système Smalltalk. De nombreux aspects de ce système ont été passés sous silence, l'environnement de développement et ses célèbres *browsers* ainsi que les nombreuses classes prédéfinies. Ces dernières font d'ailleurs toute la richesse du système en fournissant à l'utilisateur les classes qui servent à implanter le système lui-même. Le modèle MVC qui permet l'implantation rapide d'applications interactives est présenté au chapitre 4.

Dans un souci de simplification, certains concepts moins fondamentaux de la programmation objet, comme la notion de métaclasses, ont été écartés de cette présentation. L'index (page 234) et le glossaire (annexe 12) permettent cependant

d'avoir la définition des termes les plus courants. Pour sa part, le mécanisme d'héritage est traité plus en détails dans le chapitre consacré au langage leloup (chap. 5), notamment en ce qui concerne les problèmes liés à l'héritage multiple.

Le coté le plus spectaculaire de Smalltalk est de supprimer complètement le typage statique. Le typage "fort" à la Pascal est remis en cause. A ce sujet, Jean-Raymond Abrial, dans l'article *"Spécification ou prototypage ?"* [Abrial 85], définit un ensemble de procédures et de fonctions qui permettent de programmer en Pascal en escamotant complètement le système de types. Sans parler de Smalltalk ni de liaison dynamique, il exhibe en fait les désirs secrets du programmeur Pascal :

> *"Tout le monde connaît bien les difficultés qui rendent la programmation en Pascal particulièrement fastidieuse : elles sont essentiellement dues à ce que le système de types est trop "fort" et au manque d'orthogonalité de certains mécanismes. ... Enfin, qui n'a pas désiré pouvoir facilement modifier la structure des objets rémanents avec lesquels il travaille (par exemple en ajoutant ou en enlevant un ou plusieurs "champs" à certains d'entre eux) ?"*

4
Programmer avec des classes

"FORTRAN "the infantile disorder", by now nearly 20 years old, is hopelessly inadequate for whatever computer application you have in mind today: it is now too clumsy, too risky, and too expensive to use. PL1 "the fatal disease", belongs more to the problem set than to the solution set. It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as a potential programmers they are mentally mutilated beyond hope of regeneration. The use of COBOL cripples the mind: it's teaching should, therefore, be regarded as a criminal offence. APL is a mistake, carried throught perfection. It is the langage of the futur for the programming technics of the past: it creates a new generation of coding bums"

E.W. Dijkstra
How Do We Tell Truths that Might Hurt
Nuenen, 18 Juin 1975

Les langages de classes constituent un sous-ensemble des langages à objets avec lesquels la programmation revient à définir un ensemble de classes (3.4). Le style de programmation, induit par les langages de classes, n'a rien de révolutionnaire. Outre la modularité, qualité espérée de tout *bon* programme, les langages de classes favorisent une structuration de l'univers sous forme de types abstraits de données comme le font les langages Ada et CLU par exemple.

Nous étudions dans ce chapitre les avantages qu'offrent la liaison dynamique et l'héritage, spécifiques aux langages de classes, pour développer, faire évoluer et maintenir des logiciels de taille importante. Les problèmes de coût de développement, d'évolutivité et de maintenance des logiciels sont du reste une des principales raisons de l'émergence de ces langages.

Plan

La première partie de ce chapitre (§ 4.1) met en évidence les principales qualités des langages de classes par rapport à d'autres langages qui, comme Ada ou CLU, ne disposent ni du mécanisme d'héritage ni de la liaison dynamique. A l'aide d'un exemple, nous présentons dans la seconde partie comment concevoir une application avec des classes et l'intérêt d'une telle démarche par rapport à d'autres méthodes de conception (§ 4.2). Nous mettons ensuite l'accent sur les avantages que présentent certains langages de classes comme Smalltalk pour le protoypage rapide (§ 4.3).

4.1 Intérêt des langages de classes

La question n'est pas tant de savoir s'il est possible de programmer telle ou telle application avec des objets, mais plutôt de savoir si la méthode apporte un bénéfice par rapport à une approche plus classique. Il serait en effet illusoire de penser que l'utilisation des langages de classes permet de traiter plus facilement n'importe quel type de problème.

4.1.1 Algorithmique et encapsulation

Si l'approche objet impose une structuration de l'univers en termes d'objets plutôt qu'en termes de traitements, cela ne veut pas dire qu'elle exclut toute composante algorithmique. En effet, les langages de classes permettent la programmation impérative comme en Pascal par exemple. Dans de tels langages, les objets sont des entités qui regroupent des données et des méthodes, ces dernières ressemblant à s'y méprendre aux procédures d'un langage impératif classique. Contrairement à la programmation logique avec Prolog par exemple, la programmation avec des classes ne bouleverse pas les habitudes.

Algorithmique

Les langages de classes possèdent tous les structures de contrôle des langages impératifs. Tous les algorithmes classiques peuvent être réalisés avec un langage de classes.

Considérons par exemple le problème, célèbre entre tous, des tours de Hanoï. Ce problème est traité dans de nombreux ouvrages [Livercy 78] [Pair 88]. Il consiste à déplacer N disques, enfilés par taille décroissante sur un piquet de départ, vers un piquet d'arrivée en utilisant un piquet intermédiaire. De plus, les règles suivantes sont imposées :

- un seul disque est déplacé à la fois,
- tout disque doit se trouver sur un piquet ou être en cours de déplacement,


```
Object subClass: #Piquet
  instanceVariableNames: 'pileDisques'
  ...

depiler
  ↑ pileDisques depiler

empiler: unDisque
  (pileDisques estVide or:[unDisque < pileDisques sommet])
    ifTrue:[pileDisques empiler:unDisque]
    ifFalse:[self error:'empilement incorrect']
```

Figure 4.1. Définition de la classe `Piquet` en Smalltalk. Solution 1, réutilisation de la classe `Pile` par composition : la variable `pileDisque` référence une instance de la classe `Pile` définie par ailleurs.

- un disque ne peut être déplacé que s'il se trouve au sommet d'un piquet,
- enfin, un disque peut être empilé sur un piquet à condition que le disque se trouvant au sommet du piquet soit de diamètre supérieur ou qu'il n'y ait pas de disque sur le piquet.

Pour décrire l'univers du jeu des tours de Hanoï dans un style objet, il vient assez vite à l'esprit de faire de chaque piquet un objet dont l'état est matérialisé par une variable d'instance servant à stocker les disques qui y sont empilés. Cette variable peut par exemple désigner une instance de la classe `Pile`, définie par ailleurs. Deux méthodes, `empiler:` et `depiler`, permettent d'ajouter ou de retirer un disque sur un piquet. La méthode d'empilement vérifie en particulier que le disque à ajouter est plus petit que le disque se trouvant au sommet (cf. Fig. 4.1). Pour implanter la classe `Piquet`, une autre solution est envisageable et consiste à définir cette classe comme une sous-classe de la classe `Pile`. Cette deuxième solution peut être préférée à la première car elle évite en particulier de redéfinir la méthode `depiler`, héritée de la classe `Pile` (cf. Fig. 4.2). Qu'il s'agisse de la première ou de la deuxième solution, la classe `Pile` est réutilisée. La technique consistant à réutiliser une classe par l'intermédiaire d'une variable (réutilisation par composition) plutôt que par héritage permet de ne prendre qu'une partie du comportement d'une classe. En effet, l'héritage de certaines méthodes peut s'avérer gênante et source d'effets de bord non souhaités. Dans l'exemple, on peut supposer que la classe `Pile` ne dispose pas d'une méthode permettant d'extraire un élément en milieu de pile, ce qui irait à l'encontre des règles du jeu. Comme il s'agit ici de ne réutiliser que deux des méthodes de la classe `Pile` il est inutile de prendre le risque d'hériter d'une méthode *génante*. La réutilisation par composition de la

```
Pile subClass: #Piquet
  ...

empiler: unDisque
  (self estVide or:[unDisque < self sommet])
    ifTrue:[super empiler:unDisque]
    ifFalse:[self error:'empilement incorrect']
```

Figure 4.2. Définition de la classe `Piquet` en Smalltalk. Solution 2, réutilisation de la classe `Pile` par héritage : la classe `Piquet` est sous-classe de la classe `Pile`.

```
Object subClass: #ToursDeHanoi
  instanceVariableNames:
    'piquetDepart piquetIntermediaire piquetArrivee'
    ...
```

Figure 4.3. La classe `ToursDeHanoi` en Smalltalk. Les trois variables référencent respectivement les trois piquets du jeu.

classe `Pile` (cf. Fig. 4.1) est donc préférable dans ce cas.

Les disques, quant à eux, peuvent être matérialisés par des entiers ou mieux, par des instances d'une nouvelle classe, `Disque`. Cette classe doit alors disposer d'une méthode de comparaison de deux disques qui sert à vérifier que l'opération d'empilement d'un disque sur un piquet est correcte. Enfin, la classe `ToursDeHanoi`, avec ses trois célèbres piquets représentés par trois variables d'instance, peut à son tour être définie (cf. Fig. 4.3).

Cette technique nous permet de décrire avec précision l'univers du problème. La solution à ce problème reste cependant purement algorithmique. Elle peut être définie par récurrence sur le nombre de disques. Supposons que l'on sache déplacer N-1 disques, rangés par taille décroissante d'un piquet vers un autre en respectant les règles imposées par le jeu. Il devient alors possible de montrer que l'on peut également déplacer N disques. En effet, cette opération consiste à déplacer les N-1 disques du piquet de départ vers le piquet intermédiaire en laissant le plus grand disque à la base du piquet de départ. Ce disque est ensuite déplacé vers le piquet d'arrivée puis les N-1 disques du piquet intermédiaire sont à leur tour déplacés vers le piquet d'arrivée en se servant du piquet de départ comme piquet intermédiaire. A aucun moment ces opérations n'ont violé les règles du jeu à condition que l'hypothèse de récurrence soit vérifiée. Cette dernière peut facilement être validée au rang 1 qui consiste à déplacer seulement un disque.


```
deplacer: n depart: p1 arrivee: p2 pivot: p3
  n > 0
    ifTrue:[
      self deplacer:(n-1) depart:p1 arrivee:p3 pivot:p2.
      self deplacerUnDisqueDe:p1 vers:p2.
      self deplacer:(n-1) depart:p3 arrivee:p2 pivot:p1]
```

Figure 4.4. L'algorithme des tours de Hanoï en Smalltalk. On retrouve naturellement les trois étapes de la démonstration par réccurence.

La méthode `deplacer:depart:arrivee:pivot` de la classe `ToursDeHanoi`, réalise cette solution (cf. Fig. 4.4). Elle est analogue, à la syntaxe près, à une procédure écrite classiquement en Pascal.

Le test du respect des règles d'empilement qui est effectué dans la classe `Piquet` peut paraître inutile. En effet, le déroulement de la méthode `deplacer:depart:arrivee:pivot` ne viole jamais la règle d'empilement des disques. Toutefois, cette technique d'écriture, qui garantit le respect de contraintes d'intégrité sur les objets, présente deux avantages. D'une part, elle s'avère utile pour la mise au point d'un programme ne respectant pas ces contraintes. D'autre part, l'univers du problème est totalement représenté. Ainsi, le jour où l'on veut enrichir le programme des tours de Hanoï pour permettre par exemple un déplacement interactif des disques par l'utilisateur, la validité des déplacements est assurée par la classe `Piquet`.

Un langage de classes n'apporte donc aucun avantage particulier pour implanter un algorithme excepté peut-être pour sa mise au point. En tout état de cause, aucun gain ou aucune perte d'efficacité ne peuvent leur être imputés, les performances d'un programme étant essentiellement liées à la complexité de l'algorithme mis en œuvre.

Encapsulation

Un autre aspect important concernant la qualité d'un langage se rapporte aux mécanismes qu'il offre pour encapsuler les données. De ce point de vue, les langages de classes ne sont pas les seuls à fournir de tels mécanismes. CLU [Liskov 77], Ada [Ada 83] et plus généralement les langages de types abstraits de données permettent une bonne encapsulation et facilitent la production de programmes robustes aux changements de représentation des objets [Derniame 79] [Liskov 75]. Par exemple, Ada permet de représenter un nouveau type de données à l'aide de deux parties distinctes. L'interface du type, ou *package*, décrit les opérations applicables à un représentant du type tandis que le *package body* contient leur *réalisation*, c'est à dire la description de leur implantation physique.

```
with DISQUE; use DISQUE;
package PIQUET is
  type UN_PIQUET is private;
  procedure DEPILER(P: in out UN_PIQUET);
  procedure EMPILER(P: in out UN_PIQUET; D: in UN_DISQUE);
  function SOMMET(P: in UN_PIQUET) return UN_DISQUE;

  private
    type INDEX is range 0 .. 10 ;
    subtype TAILLE is INDEX range 1 .. INDEX'LAST ;
    type TABLE is array(TAILLE) of UN_DISQUE;
    type UN_PIQUET is
      record
        PILE: TABLE;
        SOMMET: INDEX := 0;
      end record;
end PIQUET;
```

Figure 4.5. L'interface du type PIQUET réalisé par un *package* Ada. Seules les déclarations qui précèdent la partie privée (**private**) sont visibles de l'extérieur.

Un type peut utiliser d'autres types à condition de ne se servir que des opérations qui leur sont applicables, c'est à dire celles définies dans leur interface. Ainsi, à l'aide de la clause **with**, le package PIQUET[1] utilise, ou importe, le package DISQUE que nous supposons être défini par ailleurs (cf. Fig. 4.5).

Seules les entêtes des opérations utilisables par l'extérieur sont mentionnées dans le package. Celles-ci précisent le type et le mode d'utilisation de leurs différents arguments, **in** s'il s'agit d'un paramètre d'entrée non modifiable par la procédure, **out** si c'est un paramètre de sortie non consultable par la procédure et enfin, **in out** si le paramètre est à la fois un résultat et une donnée d'entrée. Ainsi, la procédure exportée EMPILER possède deux arguments, P, la pile de disques à modifier et D, le disque à empiler.

L'exportation d'un type tout en cachant la structure de données qui lui est associée nécessite l'utilisation d'une partie privée introduite par le mot clé **private**. UN_PIQUET est la définition de la structure de données qui permet de mémoriser l'état d'un piquet. Cette définition peut être réutilisée dans des packages qui importent le package PIQUET à condition de n'utiliser que les opérations déclarées dans l'interface et de ne pas accéder directement à la structure de données représentant le type[2].

1. En Ada, par convention, les mots clés sont en minuscules et les identificateurs en majuscules.
2. Pour des raisons techniques liées à la compilation séparée des modules, la définition des types dont la réalisation doit être masquée ne se fait pas dans le package body [Booch 87, page 186].


```
with ERROR; use ERROR;
package body PIQUET is
  procedure DEPILER(P: in out UN_PIQUET) is
    begin
      P.SOMMET:= P.SOMMET - 1;
      exception when others
        => RAISE_ERROR(PILE_VIDE);
    end DEPILER;
  procedure EMPILER(P: in out UN_PIQUET; D: in UN_DISQUE) is
    begin
      if SOMMET(P) < D
        then
          P.SOMMET:= P.SOMMET + 1;
          P.PILE(P.SOMMET) := D ;
        else
          RAISE_ERROR(EMPILEMENT_INCORRECT) ;
        end if ;
      exception when others
        => RAISE_ERROR(PILE_PLEINE);
    end EMPILER;
  function SOMMET(P: in UN_PIQUET) return UN_DISQUE is
    begin
      return P.PILE(P.SOMMET);
      exception when others
        => RAISE_ERROR(SOMMET_INDEFINI);
    end SOMMET;
end PIQUET;
```

Figure 4.6. L'implantation du module PIQUET, un *package body* Ada.

Seules les réalisations des opérations définies dans le package body ont ce droit d'accès (cf. Fig. 4.6).

Si besoin, un package body peut importer d'autres packages en plus de ceux mentionnés dans son interface. Par exemple, le package body PIQUET importe le package ERROR qui contient un ensemble de routines de traitement d'erreurs. De même, un package body peut posséder ses propres définitions de structures de données, de variables et d'opérations qui ne sont utilisables que localement à ce package. Grâce à cette technique, le principe d'encapsulation est bien respecté : seules les opérations définies dans l'interface d'un type sont visibles de l'extérieur.

L'encapsulation de données n'est donc pas non plus l'apanage des langages de classes. En s'astreignant à une bonne discipline de programmation, un bon niveau d'encapsulation peut même être atteint avec un langage dont ce n'est pas la qualité

première, comme C par exemple [Dutta 85] [Sobelman 85].

4.1.2 Modularité et partage de code

Nous envisageons dans la suite des applications où les langages de classes se trouvent à leur avantage par rapport à d'autres langages du fait des mécanismes d'héritage et de liaison dynamique. Ada nous sert de point de comparaison car il présente certains aspects objets, en particulier la notion de package vue précédemment qui permet de représenter *l'équivalent* d'une classe (cf. Fig. 4.5), sans toutefois disposer des mécanismes d'héritage et de liaison dynamique.

L'intérêt du mécanisme d'héritage est de permettre le partage et la réutilisation de code entre des représentations de types différents sans avoir à dupliquer ou à modifier ce code.

Afin de mieux percevoir l'apport de ce mécanisme, nous reprenons l'exemple de gestion d'un stock, vu au chapitre 3, pour le traiter en Ada. Pour simplifier, nous ne considérons que trois catégories d'articles, les articles courants, les articles de luxe et enfin les vêtements. De plus, supposons que tout article doit pouvoir être consulté, en particulier pour fournir son prix HT et son prix TTC. Bien entendu, les articles de luxe diffèrent des articles de base sur le mode de calcul du prix TTC. Enfin, les articles de la catégorie vêtement se comportent comme des articles de base mais diffèrent par le fait que l'on peut les interroger pour connaître leur taille.

Nous n'insisterons pas sur la solution obtenue avec un langage de classes qui consiste à définir trois classes modélisant les trois catégories d'articles : une classe `Article` et ses deux sous-classes `ArticleDeLuxe` et `Vêtement` (cf. Fig. 3.5, page 61). Intéressons nous plutôt aux différentes manières de représenter les catégories d'articles lorsque l'on ne dispose pas du mécanisme d'héritage. Dans la suite, nous envisageons trois solutions différentes.

Solution 1 : duplication

Une première solution est la duplication de code qui consiste à définir la classe `Article` à l'aide d'un premier package puis à recopier deux fois ce package pour définir les classes `ArticleDeLuxe` et `Vêtement`. Les copies sont modifiées pour prendre en compte les spécificités des sous-classes `ArticleDeLuxe` et `Vêtement` par rapport à la classe mère, `Article`. En particulier, la méthode `prixTTC` est modifiée dans le package `ArticleDeLuxe`.

Cette technique a comme défaut d'augmenter sensiblement la taille des programmes sources comme celle des programmes exécutables. Un autre problème majeur concerne l'étape de maintenance du logiciel. La détection d'une erreur dans le package `Article` par exemple, impose une correction dans les trois packages. Le mécanisme d'héritage des langages de classes a l'avantage de répercuter automatiquement dans les sous-classes une correction faite au niveau d'une superclasse.


```
with ARTICLE; use ARTICLE;
package VETEMENT is
  type UN_VETEMENT is private;
  function PRIX_HT(V: UN_VETEMENT) return UN_PRIX;
  function PRIX_TTC(V: UN_VETEMENT) return UN_PRIX;
  function TAILLE(V: UN_VETEMENT) return UNE_TAILLE;
  ...
  private
    type UN_VETEMENT is
      record
        ARTICLE : UN_ARTICLE;
        TAILLE : UNE_TAILLE;
      end record;
end VETEMENT;
```

Figure 4.7. Le package VETEMENT en Ada.

Solution 2 : définition d'un package unique

Une autre solution, qui évite la duplication, consiste à définir un seul package pour représenter la hiérarchie des classes. L'ensemble des propriétés des différentes catégories d'objets est défini dans le même package. Dans notre exemple, cette technique conduit à ne réaliser qu'un seul package pour représenter simultanément les trois catégories d'articles. Cette solution présente l'inconvénient de ne plus rendre compte de la réalité et de ne plus être modulaire. En outre, des entités différentes étant définies à l'aide d'un seul package, certaines opérations ne s'appliquent pas uniformément sur l'ensemble de ces entités. Par exemple, l'opération `taille`, qui retourne la taille d'un vêtement, n'a de sens que lorsque l'article est un vêtement. Nous reviendrons sur cette solution dans la suite (cf. § 4.1.3) car elle s'avère être la seule permettant de simuler, en Ada, le mécanisme de liaison dynamique.

Solution 3 : composition de package

La troisième solution consiste à utiliser les packages existants et à construire les nouveaux packages par composition. Chaque classe est représentée par un package séparé dans lequel les liens d'héritage sont définis au niveau de chaque objet par l'intermédiaire d'attributs. Dans l'application de gestion de stock, les classes ARTICLE, ARTICLE_DE_LUXE, VETEMENT sont donc représentées par trois packages distincts. Le package VETEMENT, par exemple, définit l'attribut TAILLE, spécifique aux vêtements. L'attribut ARTICLE permet de réaliser le lien d'héritage (cf. Fig. 4.7).

L'attribut supplémentaire ne résoud pas totalement le problème de l'héritage. Les opérations *héritées* du package ARTICLE doivent être redéfinies dans le package

```
package body VETEMENT is
  function PRIX_TTC(V: UN_VETEMENT) return UN_PRIX is
    begin
      return PRIX_TTC(V.ARTICLE); -- opération "héritée"
    end PRIX_TTC;
  function TAILLE(V: UN_VETEMENT) return UNE_TAILLE is
    begin        -- opération spécifique aux vêtements
      return V.TAILLE;
    end TAILLE;
  ...
end VETEMENT;
```

Figure 4.8. La réalisation du package VETEMENT en Ada.

VETEMENT. Leur réalisation consiste à invoquer les procédures de mêmes noms définies dans le package ARTICLE. Ainsi, l'opération PRIX_TTC du package body VETEMENT est réalisée par un appel à l'opération de même nom du package ARTICLE (cf. Fig. 4.8). L'appel PRIX_TTC(V.ARTICLE) correspond à un appel de la fonction du package ARTICLE car l'argument, V.ARTICLE, est du type UN_ARTICLE.

Du seul point de vue de l'héritage, cette solution est comparable à celle que l'on obtiendrait avec un langage de classes. Le programmeur est néanmoins obligé de gérer lui-même ce que fait automatiquement le mécanisme d'héritage.

4.1.3 Réutilisation, évolution et maintenance

A priori, s'il ne s'agit que de modéliser des liens de similitudes entre différentes catégories d'objets, la solution précédente (Solution 3) peut paraître satisfaisante. Mais, dans la réalité, ces différentes catégories doivent pouvoir permettre de définir d'autres objets, par exemple des collections d'articles tels que les commandes ou l'ensemble des articles du stock. Dans la suite, nous nous intéressons à l'opération qui consiste simplement à calculer le montant toutes taxes comprises d'une commande, constituée d'une liste d'articles.

L'algorithme correspondant, déjà vu au chapitre 3 (page 65), est fort simple. Il effectue à l'aide d'une boucle le cumul des prix TTC pour chacun des différents articles de la commande (cf. Fig. 4.9). La fonction **prixTTC**, invoquée dans l'algorithme, est chargée de retourner le prix TTC de son argument. Bien entendu, la façon de calculer le prix TTC d'un article dépend de la catégorie d'appartenance de l'article.

Outre l'aspect modulaire de cette solution, elle présente l'avantage d'être facilement extensible et modifiable. Le changement du calcul du prix TTC pour une catégorie d'article ou l'ajout d'une nouvelle catégorie d'articles ne nécessitent que


```
fonction totalCommandeTTC (commande)
" Une commande est une liste d'articles "
  totalTTC ← 0 ;
  pour art ∈ commande faire
    début
      totalTTC ← totalTTC + prixTTC(art) ;
    fin
  retourner(totalTTC)
```

Figure 4.9. Calcul du total TTC d'une commande.

la modification de la fonction prixTTC. L'algorithme est indépendant des différentes catégories d'articles appartenant à une commande. Il est général et reste valable quels que soient les articles manipulés. Cette qualité est essentielle pour construire des programmes facilement maintenables et évolutifs.

Avec liaison dynamique

La mise en œuvre de l'algorithme **totalCommandeTTC** avec un langage de classes est immédiate et ne pose pas de difficultés particulières (cf. § 3.3.2). Il suffit de le réaliser par une méthode de la classe **Commande** qui décrit la structure et le comportement des commandes. L'appel à la fonction **prixTTC** à l'intérieur de cet algorithme correspond à un envoi de message à un article. Grâce au mécanisme de liaison dynamique, la méthode invoquée dépend de la catégorie d'appartenance de l'article. Chaque classe correspondant à une catégorie particulière d'articles doit donc disposer d'une méthode **prixTTC** soit spécifique soit héritée d'une classe plus générale. L'ajout d'une nouvelle catégorie d'articles, modélisée par une nouvelle classe, ne remet pas en cause la méthode **totalCommandeTTC** de la classe **Commande** à condition que ses instances soient capables de répondre au message **prixTTC**.

Sans liaison dynamique

Ada, comme C et Pascal d'ailleurs, sont des langages à typage statique. Le type des variables est unique et doit être connu avant l'exécution des programmes. Ainsi, au moment de la traduction de l'algorithme **totalCommandeTTC** (cf. Fig. 4.9), se pose le problème du type de la variable **art**. Celle-ci doit être capable de désigner n'importe quelle catégorie d'articles. Son type doit cependant être unique.

En Ada, pour réaliser l'agorithme **totalCommandeTTC**, il est donc nécessaire de définir un package unique pour représenter toutes les catégories d'articles. Pour traiter les différents modes de calcul de la valeur TTC d'un article, la fonction **prixTTC** de ce package doit disposer d'une information permettant de déterminer

```
with GESTION; use GESTION;
package ARTICLE is
  type CATEGORIE is (ARTICLE, ARTICLE_DE_LUXE, VETEMENT);
  type UN_ARTICLE is private;
  ...
  function REFERENCE(A: UN_ARTICLE) return UNE_REFERENCE;
  function DESIGNATION(A: UN_ARTICLE) return STRING;
  function PRIX_TTC(A: UN_ARTICLE) return UN_PRIX;
  function TAILLE(A: UN_ARTICLE) return UNE_TAILLE;
  private
    type ARTICLE_RECORD (SORTE : CATEGORIE) is
      record
        REFERENCE : UNE_REFERENCE;
        DESIGNATION : STRING(1..80);
        PRIX_HT : REAL;
        case SORTE is
          when VETEMENT => TAILLE : range 36..52;
          when others => null;
        end case;
      end record;
    type UN_ARTICLE is access ARTICLE_RECORD;
end ARTICLE;
```

Figure 4.10. Représentation des trois catégories d'articles par un seul package Ada.

la catégorie d'appartenance de l'article. Par conséquent, la structure de données représentant un article doit posséder un attribut indiquant sa nature pour savoir s'il faut lui appliquer la TVA normale ou celle des articles de luxe. En Ada, il est classique d'utiliser pour cela une *structure à discriminant*. Une telle structure possède un indicateur, le discriminant, qui permet d'interpréter différemment les autres champs de la structure en fonction de sa valeur. Ainsi, le package ARTICLE (cf. Fig. 4.10) inclut la définition d'une structure variable selon la valeur du discriminant SORTE.

Avec cette solution, la simulation de la liaison dynamique est explicitement réalisée par le programmeur qui manipule les objets différemment selon leur catégorie *logique* d'appartenance. Par exemple, la valeur du discriminant de la structure est explicitement testée à l'intérieur de la fonction PRIX_TTC (cf. Fig. 4.11).

Ce test apparaît chaque fois que la définition d'une opération nécessite la connaissance de la catégorie d'appartenance de l'article. Les inconvénients de cette technique sont nombreux. Outre le fait qu'elle n'est pas modulaire, cette pratique ne favorise pas la réutilisabilité et l'évolutivité du logiciel [Meyer 89]. L'ensemble


```
package body ARTICLE is
  function PRIX_TTC(A: UN_ARTICLE) return UN_PRIX is
    begin
      case A.SORTE is
        when ARTICLE_DE_LUXE =>
          return PRIX_TTC_ARTICLE_DE_LUXE(A);
        when others =>
          return PRIX_TTC_ARTICLE(A);
      end case;
    end PRIX_TTC;
  ...
end ARTICLE;
```

Figure 4.11. Simulation de la liaison dynamique en Ada. Le programmeur doit lui-même tester la catégorie d'appartenance de l'article traité

des choix pour le discriminant est figé, alors que la solution avec héritage permet d'ajouter une nouvelle sous-classe à la classe `Article` sans affecter l'existant. En outre, l'ajout d'un nouveau choix nécessite la remise en cause du package et, par la même occasion, celle des unités de programme se servant du package modifié.

Ainsi, les mécanismes d'héritage et de liaison dynamique apparaissent comme deux mécanismes efficaces lorsque l'univers à modéliser est composé d'objets ayant une partie de comportement commun tout en ayant des spécificités propres. De plus, la prise en compte de l'évolution de l'univers, soit par l'ajout d'une nouvelle classe d'objets, soit par la modification du comportement d'objets existants, est facilitée par ces deux mécanismes.

4.2 Développer avec des classes

Cette section n'a pas la prétention de proposer une méthodologie de la programmation objet. C'est encore un domaine peu exploré et il faudrait sans doute une thèse complète pour en venir à bout. Dans un premier temps, nous comparons deux familles de méthodes de conception, puis nous mettons l'accent sur les avantages de l'approche objets pour l'évolution et réutilisation du logiciel.

4.2.1 Analyse du problème

La programmation avec un langage de classes est une technique d'écriture de programme qui n'échappe pas à une phase préalable plus conceptuelle d'analyse.

Le principe essentiel de toute méthode de conception consiste à découper l'application à analyser en parties indépendantes souvent appelées *modules.* L'objectif d'une telle démarche est de diviser la complexité globale de l'application afin que le concepteur puisse concentrer son travail sur des points précis sans avoir à se soucier du reste de l'application [Dijkstra 76].

Deux grandes familles de méthodes de conception obéissant à ces principes se dégagent suivant que l'accent est mis sur les *traitements* que doit effectuer l'application [Wirth 71a] [Wirth 87] [Arsac 77] ou sur les *données* qu'elle manipule [Liskov 74] [Booch 86] [Cox 86].

Les méthodes orientées traitements relèvent souvent d'une approche *descendante* de la phase d'analyse. Elles supposent que le niveau le plus général de l'application traitée peut être décrit par un "programme" principal [Meyer 89, § 4.3]. Celui-ci est décomposé en sous-programmes indépendants, de complexité moindre, sans se soucier des détails de leur réalisation. Dans la mesure du possible, ce processus est répété jusqu'à obtenir des sous-programmes élémentaires, dont la réalisation ne pose plus de problèmes. En revanche, les méthodes orientées *données* consistent à déterminer d'abord quelles sont les entités appartenant à l'univers à modéliser et quelles opérations leur sont applicables, avant de se préoccuper de l'enchaînement de ces opérations. Dans le premier cas, l'unité de décomposition est le traitement, dans le second cas, c'est l'objet *et* les opérations qui lui sont associées.

Le résultat de la phase d'analyse est théoriquement indépendant du langage de programmation utilisé pour écrire l'application. Dans les faits, ce n'est pas le cas. Une méthode de conception favorisant les traitements s'adapte bien à un langage procédural de type Pascal où, chaque traitement est représenté à l'aide d'une procédure. Les méthodes mettant l'accent sur les données s'appliquent généralement à des langages de classes ou de types abstraits de données.

Un exemple : l'impression d'un livre

La conception puis la programmation d'une application telle que l'*impression d'un livre* va nous servir à comparer ces deux familles de méthodes. Décidons que la structure du livre à imprimer est comparable à celle de la présente thèse : l'entête est composé d'un titre et du nom de l'auteur et chaque chapitre est décomposé en une liste de sections, elles-mêmes composées de paragraphes et de figures.

L'analyse orientée traitements d'une telle application conduit, dans un premier temps, à s'interroger sur le traitement le plus général : celui-ci consiste en l'impression de l'entête, suivi de l'impression des chapitres et enfin de l'impression de la bibliographie (cf. Fig. 4.12).

Dans un deuxième temps, les traitements issus de la première étape font à leur tour l'objet d'une décomposition. Par exemple, l'opération `imprimerEntête` est décomposée en un enchaînement en séquence de deux autres opérations qui réalisent respectivement l'impression du titre et l'impression du nom de l'auteur. Le processus de décomposition est appliqué jusqu'à obtenir des traitements élémentaires.


```
procédure imprimerLivre
  imprimerEntête ;
  imprimerListeDesChapitres ;
  imprimerBibliographie ;
```

Figure 4.12. Procédure principale d'impression d'un livre.

Chaque traitement est composé d'un ensemble de sous-traitements. La composition des traitements repose sur trois constructeurs. Le premier permet d'enchaîner séquentiellement des opérations, le second d'itérer l'exécution d'un traitement tant qu'une condition est vérifiée, et, le troisième exprime une alternative déterminée par une condition. Le résultat de l'analyse descendante peut alors être représenté sous la forme d'un schéma tel que celui de la figure 4.13.

Une conception orientée données du même problème produit une organisation différente de la solution. L'univers du problème est décomposé en entités conceptuelles. Le choix de ces entités, qui seront ultérieurement matérialisées par des classes, est la partie la plus délicate de la conception. Il n'y a pas de techniques infaillibles et pour déterminer les *bonnes* catégories d'objets qui interviennent dans la structuration de l'univers d'un problème. Cette étape nécessite une certaine expérience, voire du talent [Meyer 89].

Dans l'exemple, heureusement, ce découpage est relativement simple. Les figures, les paragraphes, les chapitres, les références bibliographiques et les livres apparaissent comme étant autant d'entités ayant un comportement spécifique. Leur représentation, à l'aide de classes, ne posent pas de difficultés particulières. La classe `Livre`, par exemple, peut être définie à l'aide de trois variables d'instances, l'une représentant l'entête du livre, une autre la liste des chapitres et la troisième les références bibliographiques (cf. Fig. 4.14).

Le problème algorithmique lié à l'impression du livre n'est pas pour autant résolu. Comme dans le cas des tours de Hanoï, il reste à définir le comportement du livre. En particulier, la méthode `imprimer` doit être définie dans la classe `Livre` (cf. Fig. 4.14). De même, il est nécessaire de définir de telles méthodes dans les autres classes, `Chapitre`, `Section`, `Paragraphe` et enfin `Figure`.

Réutilisabilité

A ce stade, si le but de l'application ne consiste qu'à imprimer un livre, les deux méthodes de conception, celle orientée traitements et celle orientée données, sont comparables, aussi bien pour la modularité que pour la lisibilité. Toutefois, dans la réalité, une application ne cesse d'évoluer et doit être capable de rendre

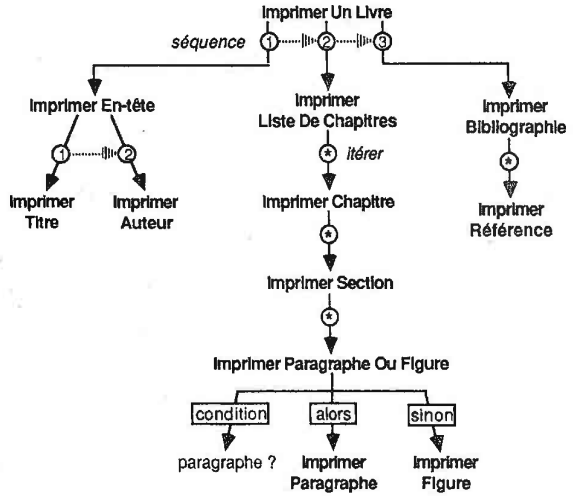


Figure 4.13. Décomposition du traitement de l'impression d'un livre.

de nouveaux services. Les fonctionnalités définies au départ doivent pouvoir facilement être enrichies. Par exemple, certains utilisateurs vont avoir besoin d'imprimer d'autres types de documents, comme des lettres, des thèses, des rapports, des articles etc.
Un article se comporte comme un livre à la différence qu'il possède en plus une partie mots clés et une partie résumé. Se pose alors le problème de la réutilisation de tout ou partie de ce qui a déjà été écrit pour l'impression du livre.

Il est évident qu'il existe une forte analogie entre le problème déjà résolu, celui de l'impression d'un livre, et le nouveau problème, l'impression d'un article. La preuve en est que la nouvelle analyse, qu'elle soit orientée données ou orientée traitements, peut reprendre des parties entières des résultats précédents. Au niveau conceptuel, il est bien difficile de dire laquelle des deux méthodes facilite le plus la réutilisabilité. En revanche, si l'on observe l'organisation au niveau du code, la solution objets possède d'indéniables qualités de réutilisabilité. Des classes telles que **Chapitre**, **Paragraphe** et **Figure** par exemple peuvent être réutilisées directement dans le nouveau problème, sans modification.

Considérons maintenant le programme Pascal issu de l'analyse orientée traitements (cf. Fig.4.13). S'il suit fidèlement l'analyse, les procédures qu'il contient sont *emboîtées* les unes dans les autres comme des poupées russes (cf. Fig. 4.15). Bien entendu, avec une telle organisation, les procédures qui sont locales à d'autres procédures ne peuvent être réutilisées à moins d'être dupliquées. Les problèmes


```
Document subClass: #Livre
  instanceVariableNames:
    'entete listeDeChapitres bibliographie'
    ...

imprimer
  entete imprimer.
  listeDeChapitres imprimer.
  bibliographie imprimer
```

Figure 4.14. La classe `Livre` et sa méthode `imprimer` en Smalltalk.

```
program imprimerLivre ;
  type livre= record
                unEntete : entete;
                desChapitres : listeDeChapitres ;
                laBiblio : listeDeReferences ;
              end ;
       paragraphe= ...
       ...
  var leLivre: livre
      ...
  procedure imprimerEntete (...);
      procedure imprimerTitre (...);
        begin ... end;
      procedure imprimerRemerciements (...);
        begin ... end;
      procedure imprimerAvantPropos (...);
        begin ... end;
    begin
      imprimerTitre (...);
      imprimerRemerciements (...);
      imprimerAvantPropos (...);
    end;
  procedure imprimerListeDeChapitres (...);
    ...
    ...
begin (* programme principal *)
  imprimerEntete (...);
  imprimerListeDeChapitres (...);
  imprimerBibliographie (...);
end.
```

Figure 4.15. Structure du programme `imprimer` en Pascal.

inhérents à la duplication de code apparaissent (cf. § 4.1.2). Heureusement, dans la pratique, les programmes écrits avec un langage procédural respectent rarement tous les niveaux d'emboîtement issus de la phase de conception. Les parties de traitement identiques qui sont détectées à l'intérieur d'un même programme sont la plupart du temps sorties de leur bloc d'origine pour être utilisables plus globalement. Du reste, certains langages comme C, n'autorisent pas la définition de fonctions locales à d'autres fonctions.

Malheureusement, cette technique de mise en facteur de traitements, faisant de la procédure l'unité de partage de code, s'avère très contraignante et insuffisante pour être utilisée à grande échelle. En effet, si le programme qui importe une procédure n'a pas à connaître les détails de réalisation de celle-ci, il en est tout autrement de la structure des données qu'il lui passe en paramètre. A moins de revenir à une technique objet qui consiste à regrouper dans une même unité de compilation à la fois la définition de la structure de l'objet et les opérations qui lui sont associées, faire de la procédure l'unité de partage de code ne respecte pas les principes d'encapsulation et d'abstraction de données.

Intégration

Posons nous maintenant un autre problème qui consiste cette fois à ajouter à l'application précédente un éditeur de livre. Ce problème d'édition est complètement différent du problème précédent et l'on pourrait penser que les approches orientées traitements ou données sont cette fois équivalentes. Il en serait ainsi si le programmeur ne partait de rien pour développer cette nouvelle application. Néanmoins, dans le cas de l'approche objets, puisque le problème de la réutilisation est bien maîtrisé, le programmeur dispose généralement d'une bibliothèque de composants riche et variée : éditeur de textes, éditeur graphique, gestionnaire de menus... Il a donc la possibilité d'*intégrer* un ensemble de composants logiciels pour réaliser un éditeur de livres, comme nous allons le voir dans le paragraphe suivant.

4.3 Prototypage

La conception d'une application interactive est un problème complexe. S'il est déjà difficile d'établir un cahier des charges précis dans d'autres domaines, ce travail est particulièrement délicat lorsque les applications visées doivent posséder un haut degré d'interactivité. La forme des interfaces comme les séquences de communication entre la machine et l'usager doivent pouvoir facilement être remises en cause afin de satisfaire l'utisateur. Aussi, dans ce cas particulier, il est bénéfique de pouvoir disposer d'un prototype, même incomplet, pour affiner l'élaboration du cahier des charges.

Les langages de classes favorisent la réutilisation et l'évolution du logiciel et sont donc adaptés à la construction de prototypes. Smalltalk et les langages de la famille LISP sont plus aptes à remplir cette fonction de prototypage du fait

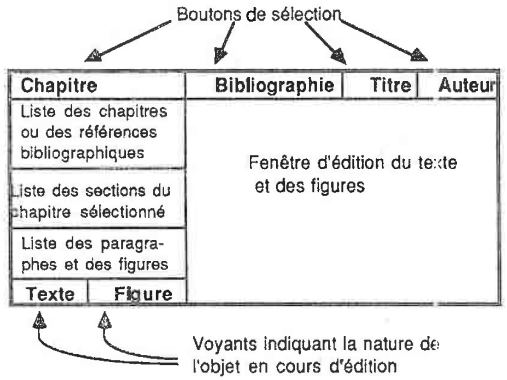


Figure 4.16. Le projet d'interface de l'éditeur de livres.

de leur caractère interprété [Bezivin 86a]. Cette spécificité a permis le développement d'environnements de programmation puissants autour d'eux, incluant différents outils (débogueur, pisteur pas à pas, browser...). Elle facilite également la programmation incrémentale (cf. chap. 1.4.1).

4.3.1 L'interface de l'éditeur

Dans la suite, le système Smalltalk nous sert de support pour développer le prototype d'éditeur de livres. L'aspect visuel de l'interface que nous proposons pour cet éditeur (cf. Fig. 4.16) est fortement influencé par l'utilisation des browsers du système Smalltalk. D'une manière générale, il est important de respecter le style des interfaces du système hôte pour que l'application développée ait le plus de chances de ne pas perturber les habitudes de l'utilisateur. Ainsi, un utilisateur Macintosh s'adapte rapidement à une nouvelle application Macintosh tout comme un utilisateur habitué à l'environnement Smalltalk s'adapte à un nouveau browser.

L'interface de l'éditeur que nous allons définir se présente comme une fenêtre découpée en plusieurs zones ou *vues*. La plus grande des vues est destinée à l'édition d'un texte ou d'une figure. Les deux petits voyants, dans le coin inférieur gauche de l'éditeur, indiquent la nature de l'objet manipulé. Ces deux zones ne sont pas actives et ne réagissent pas aux actions de la souris, leur rôle n'est qu'informatif.

Pour l'utilisateur, les quatre zones en haut sont comme les boutons d'un poste de radio. Un seul peut être enfoncé à la fois, permettant ainsi la sélection d'un des quatres modes de fonctionnement : édition d'un chapitre, de la bibliographie, du titre ou de la liste des auteurs du document.

Le contenu et le rôle des autres zones dépend du mode sélectionné. Les trois zones restantes, situées sur la partie gauche de l'interface, sont liées entre elles en mode **chapitre**. Elles permettent la sélection d'un paragraphe ou d'une figure à éditer. La plus haute des trois contient la liste des chapitres du livre. Si un chapitre est sélectionné dans cette vue, la vue située en dessous contient la liste des sections qui constituent ce chapitre. Sur le même principe, la vue située sous celle des sections permet de choisir le paragraphe ou la figure à éditer et provoque l'allumage du voyant correspondant. En mode **bibliographie**, seule la plus haute des trois zones précédentes est utilisée pour le choix de la référence bibliographique à éditer. Dans les autres modes, **titre** et **auteur**, ces vues ne sont pas utilisées et ne contiennent rien.

Problèmes

La réalisation de l'interface de l'éditeur est complexe si l'on ne dispose pas d'outils adaptés. Il faut construire tous les éléments de l'interface : les boutons, les fenêtres contenant des listes d'éléments sélectionnables, les voyants, les fenêtres d'édition de textes et de figures. Les problèmes à résoudre sont nombreux. Outre les problèmes liés à l'affichage de fenêtres, il faut fournir à l'utilisateur les moyens de sélectionner un élément dans une liste à l'aide de la souris, de construire des figures interactivement, de saisir du texte...

Encore une fois, si le programmeur n'est pas en mesure de réutiliser différents composants prédéfinis, il lui faudra gérer les entrées/sorties clavier et souris afin de mettre à jour les zones de l'éditeur. La tâche s'annonce ardue. De plus, il faudra être en mesure d'*intégrer* l'ensemble des éléments de l'interface aux programmes déjà écrits pour résoudre le problème de l'impression du livre.

Classiquement, un système informatique offre au programmeur un ensemble d'outils qui lui permettent d'éditer des programmes, de les mettre au point, puis de les exploiter [Krakowiak 85]. Dans la plupart des cas, les différents logiciels assurant ces fonctions sont indépendants et le passage de l'un à l'autre nécessite l'emploi d'un langage de commande externe. Ayant été conçues séparément, les interfaces entre l'utilisateur et ces outils sont en outre rarement homogènes. Apporter une modification, même minime, à un logiciel de l'environnement s'avère bien souvent difficile, voire impossible, à moins que cela n'ait été soigneusement prévu par le concepteur du système.

4.3.2 Le modèle MVC

L'environnement de programmation Smalltalk est bâti selon des principes totalement différents. L'ensemble des outils, éditeur de textes, éditeur graphique, débogueur, *browsers* et gestionnaire de fenêtres, est *intégré* au système. Les objets composant cet environnement sont créés à partir de classes spécifiquement prévues pour cela et sont bien entendu activés par envoi de message. Comme il a accès à la définition de ces classes, l'utilisateur a tout loisir de créer un environnement de

travail personnalisé ou de développer des applications interactives, en réutilisant et en enrichissant l'environnement prédéfini. Les interfaces de ces applications sont très semblables puisqu'elles sont conçues à partir des mêmes éléments de base. L'utilisateur n'est donc pas désorienté lorsqu'il passe de l'une à l'autre.

Le paradigme *Modèle Vue Contrôleur* du système Smalltalk, MVC [Krasner 88], plus qu'une simple bibliothèque de composants élémentaires comme celle du Macintosh [Mac 86], permet de systématiser l'écriture d'applications interactives. Des outils analogues existent dans quelques systèmes LISP comme par exemple la bibliothèque Aida [Aid 88] écrite avec la couche objet de Le-Lisp. Le schéma de développement d'applications interactives proposé par Smalltalk, MVC, repose sur trois composantes :

- le *modèle* représente les données manipulées dans l'application,
- la *vue* des données : elle est présentée dans un ensemble de fenêtres sur le terminal et constitue l'interface visuelle entre l'utilisateur et les données,
- le *contrôle* des données : il constitue l'interface permettant à l'utilisateur d'agir sur les données, à l'aide du clavier et de la souris, en se référant à la vue.

Si le modèle des données est spécifique d'une application, la manière de les présenter à l'écran ou de les manipuler est en revanche commune à un ensemble d'applications. Editer un texte, sélectionner un item dans une liste ou désigner une icône avec la souris nécessitent la mise en œuvre des mêmes outils, quelle que soit l'application. Le système Smalltalk propose donc toute une gamme prédéfinie de classes permettant de créer des fenêtres et des contrôleurs adaptés à différentes manières de présenter des données.

Les vues

Les vues sont des parties de l'écran dans lesquelles se déroulent des activités spécifiques. Leur structure et leur comportement sont définis dans les classes `View` [3]. Certaines fenêtres sont spécialisées pour l'édition de textes (classe `StringHolderView`), d'autres pour l'édition graphique (classe `FormHolderView`), d'autres encore permettent de visualiser une liste d'items et d'en sélectionner un (classe `SelectionInListView`). Au total, une vingtaine de types différents de fenêtres sont disponibles dans la bibliothèque initiale. Elles représentent autant de manières de visualiser tout un objet ou une de ses parties.

Les contrôleurs

A chaque vue est associé un contrôleur qui est activé lorsque la fenêtre correspondante est pointée par la souris. Une seule vue est donc active à la fois. Le contrôleur répercute les actions de l'utilisateur sur la fenêtre qui en dépend.

3. `Pane` en Smalltalk/V

Il existe un contrôleur particulier pour chaque type prédéfini de fenêtre. Ainsi, une fenêtre principale, généralement instance de la classe `StandardSystemView`, est associée à un contrôleur instance de la classe `StandardSystemController`. Le rôle de ce dernier consiste essentiellement à déterminer la sous-fenêtre qui se trouve désignée par la souris et à activer le contrôleur correspondant. De même, une fenêtre instance de la classe `StringHolderView`, conçue pour l'édition de textes, va de paire avec un contrôleur instance de la classe `StringHolderController`. Celui-ci scrute l'arrivée des caractères en provenance du clavier et gère leur affichage dans la fenêtre.

Les différents éléments de l'interface de l'éditeur de livre, boutons, voyants, zones d'édition de textes, etc., vont pouvoir être réalisés grâce à des composants prédéfinis dans l'environnement Smalltalk. De tels composants sont du reste utilisés par le système lui même et servent à définir les browsers de l'environnement de programmation.

L'affichage à l'écran des vues est réalisé grâce à des fenêtres, instances de différentes classes sous-classes de `View`. Par exemple, l'aspect visuel d'un bouton, s'il est enfoncé ou non, est réalisé avec une instance de la classe `BooleanView`. Le contrôle des événements en provenance du clavier ou de la souris est effectué par un contrôleur associé à chaque fenêtre. Les contrôleurs sont des instances de sous-classes de `Controller`. Ces sous-classes définissent des contrôleurs adaptés aux différentes sortes de fenêtres prédéfinies dans les sous-classes de `View`. Par exemple, le contrôleur par défaut d'une instance de la classe `BooleanView` est une instance de la classe `SwitchController`. Ainsi, lorsque l'utilisateur appuie sur un bouton, cet évènement est capté par le contrôleur associé à ce bouton. Le contrôleur avertit alors la fenêtre qui lui est associée afin qu'elle matérialise cet évènement à l'écran. Dans ce cas précis, cela peut consister à noircir le bouton pour signifier à l'utilisateur que son action a été prise en compte.

En fait, pour l'instant, aucune action n'est attachée au bouton : si celui-ci réagit visuellement, il ne déclenche aucun autre évènement. En particulier, lorsque l'utilisateur appuie sur le bouton `chapitre` de l'éditeur, l'affichage de la liste des chapitres doit apparaître dans la sous-vue correspondante. Il est donc nécessaire de pouvoir associer des actions aux agissements de l'utilisateur sur les vues de l'éditeur et de traiter la communication entre les différentes vues. Cette tâche est dévolue au *modèle* qui joue, en quelque sorte, le rôle de chef d'orchestre.

Le modèle

Dans le paradigme MVC, chaque fenêtre connaît un objet particulier, appelé son *modèle*. Une classe, définissant le comportement de cet objet, est associée à chaque catégorie d'applications interactives. Par exemple, un modèle associé à un browser est instance de la classe `Browser`, sous-classe de `Model`. Ainsi, chaque fenêtre peut interroger son modèle en lui envoyant des messages, en particulier pour connaître l'aspect qu'elle doit prendre. En outre, le modèle peut avertir les fenêtres qui lui sont associées d'éventuels changements les concernant. Par exemple, si l'utilisateur


```
Model subClass: #ModelLivre
  instanceVariableNames:
    'boutonEnfonce chapitreSelectionne sectionSelectionnee
     paragrapheSelectionne voyantAllume biblioSelectionnee
     unLivre'
    ...
```

Figure 4.17. Définition de la classe `ModelLivre` en Smalltalk.

appuie sur un bouton alors qu'un autre bouton est déjà enfoncé, ce dernier doit se relever. Dans ce cas, le modèle avertit le bouton enfoncé qu'il doit se relever.

Le modèle associé à l'éditeur de livre doit donc disposer de suffisamment d'informations pour déterminer le contenu des fenêtres : quel bouton est enfoncé, quel voyant est allumé, quel chapitre, quelle section, quel paragraphe ou, quelle référence bibliographique sont sélectionnés. Enfin, le modèle de l'éditeur de livre doit connaître le livre sur lequel il est ouvert afin de pouvoir renseigner les fenêtres sur leur contenu. C'est à ce niveau que se réalise l'intégration entre, d'une part, un objet instance de la classe `Livre`, et, d'autre part, l'éditeur de livre. Toutes ces informations peuvent être matérialisées à l'aide de variables d'instances (cf. Fig. 4.17) spécifiques à cette nouvelle classe de modèles. La variable d'instance `boutonEnfonce` par exemple contient le nom du bouton qui est enfoncé.

Une vue dialogue avec son modèle

Entre chaque fenêtre et son modèle s'établit une communication particulière qui est fonction de la nature de la fenêtre. Par exemple, le dialogue entre un bouton et son modèle se compose de deux séquences de communication. D'une part le bouton peut questionner son modèle pour connaître son état, enfoncé ou non. D'autre part, il avertit son modèle lorsque l'utilisateur le sélectionne à l'aide de la souris. Ces deux séquences sont réalisées par des envois de message. Les noms de ces messages sont définis à la création du bouton. Instancions par exemple la classe `BooleanView` pour créer le bouton `Chapitres` qui se trouve dans le coin supérieur gauche de l'interface (cf. Fig. 4.16). La méthode d'instanciation comporte cinq arguments (cf. Fig. 4.18) :

- le modèle du bouton introduit par le mot clé `on:`,
- le sélecteur du message, que le bouton envoie à son modèle pour connaître son état, introduit par le mot clé `aspect:`,
- le texte à afficher sur le bouton introduit par le mot clé `label:`,
- le sélecteur du message, que le bouton envoie à son modèle pour lui signaler que l'utilisateur vient de le sélectionner, introduit par le mot clé `change:`,

```
BooleanView
  on:unModelLivre
  aspect:#bouton
  label:'Chapitres'
  change:#boutonSelectionne:
  value:#chapitres
```

Figure 4.18. Création du bouton `Chapitres`, instance de la classe `BooleanView` en Smalltalk. Deux sélecteurs de méthodes sont donnés en arguments de la méthode d'instanciation : `#bouton` et `#boutonSelectionne:`.

```
bouton
  ↑boutonEnfonce
```

Figure 4.19. La méthode `bouton` de la classe `ModelLivre` permettant de répondre à un bouton.

- et enfin, introduite par le mot clé `value:`, la valeur du bouton.

En conjonction à cette instanciation, il faut, pour que le modèle soit en mesure de dialoguer avec le bouton, définir les méthodes `bouton` et `boutonSelectionne:` dans la classe `ModelLivre`.

Lorsque le bouton `Chapitres` veut connaître son état, il interroge son modèle en lui envoyant le message `bouton`. Si la réponse du modèle est égale à la valeur du bouton, celui-ci matérialise sur l'écran le fait qu'il est sélectionné. La variable d'instance du modèle, `boutonEnfonce`, sert à mémoriser lequel des boutons est enfoncé. Cette variable contient le symbole `#chapitres` lorque le bouton `Chapitres` est sélectionné (cf. Fig. 4.19).

L'autre méthode, `boutonSelectionne:` (cf. Fig. 4.20), est automatiquement activée lorsque l'utilisateur appuie sur le bouton. L'argument de la méthode désigne la valeur du bouton sélectionné. La première action de cette méthode est de modifier la variable d'instance du modèle, `boutonEnfonce`, pour lui affecter la valeur du nouveau bouton sélectionné. Les actions qui suivent permettent d'avertir certaines fenêtres de l'interface qu'une modification est intervenue sur le modèle. Celles-ci, une fois averties, interrogeront le modèle pour mettre à jour leur contenu. Dans le cas présent, tous les boutons de l'interface sont avertis de la modification de l'état d'un bouton. Cette opération est effectuée à l'aide de la méthode `changed:` prédéfinie dans la classe `Model`. L'argument d'appel de la méthode `changed:` est un symbole qui permet de désigner sélectivement certaines fenêtres. Seules celles


```
boutonSelectionne:valeurDuBouton
  boutonEnfonce ←valeurDuBouton.
  self changed:#bouton.
  self changed:#listeChapitreBiblio.
```

Figure 4.20. La méthode `boutonSelectionne` de la classe `ModelLivre` qui est automatiquement activée lorsqu'un bouton est sélectionné.

```
BooleanView
  on:unModelLivre
  aspect:#bouton
  label:'Bibliographie'
  change:#boutonSelectionne:
  value:#bibliographie
```

Figure 4.21. Création du bouton `Bibliographie`, instance de la classe `BooleanView`. Le sélecteur concernant l'aspect du bouton, `#bouton`, est le même que pour le bouton chapitre (cf. Fig. 4.18) et la méthode correspondante est partagée (cf. Fig. 4.19).

qui ont été instanciées avec le même symbole derrière le mot clé `aspect:` sont concernées. En particulier, lorsque le modèle invoque la méthode `changed:` (cf. Fig. 4.20) avec comme paramètre `#bouton`, le bouton `Chapitres` (cf. Fig. 4.18) est averti. A son tour, celui-ci interroge automatiquement son modèle, à l'aide du message `bouton`, pour connaître son état.

Tous les autres boutons de l'interface, `Bibliographie`, `Titre` et `Auteur` peuvent être créés sur le principe du bouton `Chapitres`. Ils diffèrent seulement par leur valeur et bien entendu par leur label (cf. Fig. 4.21).

Les deux voyants situés dans le coin inférieur gauche de l'interface peuvent être réalisés de manière analogue. Ils diffèrent cependant par le fait que leur sélection par l'utilisateur n'a aucun effet (cf. Fig. 4.22).

Comme pour les boutons, le modèle doit être en mesure de répondre aux messages qui réalisent le dialogue avec les voyants (cf. Fig. 4.23).

Le dialogue qui s'établit entre une instance de `BooleanView` et son modèle est relativement simple. Celui qui s'établit entre une fenêtre contenant une liste d'éléments sélectionnables, comme la fenêtre contenant la liste des chapitres par exemple, est plus riche. Ces fenêtres, instances de la classe `SelectionInListView`, utilisent cinq messages pour communiquer avec leur modèle. Comme pour les bou-

```
BooleanView
  on:unModelLivre
  aspect:#voyant
  label:'texte'
  change:#neRienFaire:
  value:#texte
```

Figure 4.22. Création du voyant `texte`, instance de la classe `BooleanView`.

```
voyant
  |voyantAllume

neRienFaire:valeurVoyant
  "Aucune action n'est déclenchée lorsqu'un voyant
  est sélectionné par l'utilisateur."
```

Figure 4.23. Les méthodes du modèle de l'éditeur de livre permettant de répondre aux voyants.

tons, certains messages servent à interroger le modèle, pour connaître quel élément est sélectionné par exemple. Les autres servent à avertir le modèle d'une action de l'utilisateur comme la sélection d'un élément à l'aide de la souris.

Evolution de l'interface

Quelle que soit la complexité du dialogue entre une vue et son modèle, le principe reste le même. Les noms des messages que peut envoyer une vue à son modèle, soit pour connaître son état, soit pour l'avertir d'une action de l'utilisateur, sont définis à l'instanciation de la vue. Bien entendu, les méthodes associées à ces messages doivent être définies dans la classe du modèle.

Le fait que les sélecteurs des messages envoyés par les vues à leur modèle soient définis à l'instanciation et donc non prédéfinis offre une très grande souplesse. Il est possible d'ajouter ou de supprimer des fenêtres sans remettre en cause l'interface dans sa globalité. Cette dynamicité ne permet pas d'effectuer les vérifications statiques de cohérence de type comme le font les compilateurs des langages fortement typés. En contrepartie, elle facilite à la fois la réutilisation de composants existants et l'évolution des applications. Ces deux points sont essentiels pour le prototypage rapide.

Problèmes résiduels

Bien entendu, la programmation de l'éditeur de livre n'a pas été présentée complètement. En particulier, les problèmes de positionnement des différentes vues composant l'interface dans la fenêtre principale n'ont pas été abordés. Si la prise en compte des événements est résolue de façon acceptable par les contrôleurs du système MVC, la mise en place géographique des composants d'une vue est réalisée de façon assez archaïque en indiquant des valeurs numériques de positionnement. Une solution interactive de positionnement *à la souris* des composants, comme dans le système "SOS Interface" [Hullot 86] serait souhaitable. En outre, le système MVC ne permet pas de définir une sous-vue de taille fixe, ce qui peut être gênant pour certains composants comme les boutons ou les barres de défilement pour lesquels il est préférable de fixer la taille une bonne fois pour toutes [Epstein 88].

Programmation par contraintes

Une critique importante qui peut être faite au système MVC est que la description d'interfaces reste pour une grande partie procédurale[4]. La tendance actuelle consiste à préférer une description d'interface à l'aide de contraintes. L'origine de ces travaux est le projet ThingLab [Borning 81] , un laboratoire de simulation de contraintes. Le principe est simple et consiste à réaliser le couplage entre deux objets à l'aide d'un ensemble de contraintes dont le respect est assuré automatiquement par le système. On peut par exemple établir un couplage entre une barre de défilement représentant un thermomètre et la valeur de la température rangée dans une variable entière. Dans ce cas, la relation permettant de mettre en place la contrainte est une simple équation :

```
hauteurDuThermomètre = constante × valeurDeLaTemperature
```

Deux points sont particulièrement intéressants dans cette approche. En premier lieu, le respect des contraintes est assuré automatiquement par le système ce qui évite les oublis éventuels de mise à jour d'un des composants de l'interface, comme l'oubli `d'un self changed:` par exemple. En deuxième lieu, cela permet d'établir un couplage bi-directionnel entre les objets contraints [Borning 86]. Dans le cas du thermomètre, le couplage est assuré dans les deux sens. Si l'entier qui contient la température est modifié, l'aspect de la barre de défilement est automatiquement mis à jour. Inversement, si l'utilisateur modifie avec la souris la hauteur de la colonne du thermomètre, la valeur de l'entier correspondant est également mise à jour.

4. "*Our approach was guided by experience with the Smalltalk MVC paradigm. Programming experience has shown that this paradigm is hard to follow.*" [Ege 87]. Pour notre part, c'est aussi en suivant le déroulement des opérations à l'aide du pisteur Smalltalk que nous avons découvert la *cuisine interne* [Krasner 88] du modèle MVC pour lequel la véritable notice d'utilisation n'existe pas.

La mise en œuvre d'un système capable d'assurer le respect automatique des contraintes pose cependant deux problèmes majeurs. D'une part, il faut que l'efficacité du système soit suffisante pour une utilisation interactive. D'autre part, la panoplie des contraintes que l'on peut fixer entre les objets doit être suffisamment riche pour prendre en compte tous les cas de couplage. En outre, il est important de conserver la reversibilité des contraintes. Quoiqu'il en soit, le système MVC de Smalltalk reste à l'heure actuelle un des meilleurs systèmes commercialisé pour la construction rapide d'interface. Avec un peu de pratique et une certaine expérience, le prototype d'éditeur interactif de livre peut être rapidement mis au point et servir de base de discussion avec les futurs utilisateurs en vue de son amélioration.

4.4 Conclusion

La modélisation du monde réel à l'aide d'objets organisés en classes offre des avantages indéniables par rapport à une approche plus classique. A chaque entité du monde réel peut être associé un objet dont le comportement est défini à l'intérieur d'une classe. En ce sens, les langages de classes s'inscrivent dans la lignée des langages de types abstraits de données. Cependant les deux mécanismes qu'ils ajoutent, l'héritage et la liaison dynamique, permettent de supporter une nouvelle forme de polymorphisme. Une variable peut référencer des objets instances de classes différentes.

A première vue, ces mécanismes peuvent paraître dangereux en introduisant un certain indéterminisme sur la procédure qui est réellement activée suite à un envoi de message. En contrepartie, comme nous l'avons présenté dans ce chapitre, ils favorisent aussi bien la réutilisation que l'évolution ou l'intégration de logiciels.

Cet indéterminisme est contrôlé dans certains langages de classes, dits fortement typés, comme Eiffel par exemple (page 70). Ce contrôle augmente le niveau de sécurité des programmes, permettant ainsi l'utilisation de ces langages par des équipes de taille importante. Néanmoins, ces langages restent relativement rigides dans leur utilisation. En particulier, l'implantation du système MVC dans un langage comme Eiffel n'est pas directement envisageable car ce langage ne permet pas la manipulation dynamique de sélecteurs, une des clés de l'implantation actuelle du système MVC.

En Smalltalk et dans les langages objets construits au-dessus de LISP, aucun contrôle n'est effectué sur le type des variables. Cette absence de contrôle a permis le développement des environnements de programmation les plus perfectionnés à ce jour. A notre avis, les erreurs de type, qui sont normalement détectées dans la phase de compilation ne sont en fait que des erreurs *bêtes* de programmation. Celles-ci sont, la plupart du temps, rapidement détectées et corrigées dès les premiers essais grâce aux environnements de programmation qu'offrent ces langages. Il serait dommage de se priver de cette souplesse, *a fortiori* si le langage est utilisé pour le prototypage, le niveau de sécurité des programmes n'étant pas un critère essentiel.

5
Programmer en leloup

" Il n'est que naturel que l'artisan qui polit le sabre doive aussi polir l'esprit de celui qui le manie.
[page 205] "

Eji Yoshikawa [Yoshikawa 83]

Conçu dans le cadre du projet GEPI [Canals 88], leloup[1] est un langage objets qui emprunte des caractéristiques à de plusieurs langages comme Smalltalk [Goldberg 84b], Flavors [Moon 80] et, dans une moindre mesure, des langages à base de frames [Bobrow 76] [Bobrow 77].

Les caractéristiques que nous avons choisie d'intégrer au langage sont relativement nombreuses et diverses, et ont été choisies pour faciliter le développement rapide de prototypes par des équipes de taille réduite. Dans cette optique, l'objectif principal de leloup est de permettre un partage de code maximum entre les différents modules de l'application.

Le principal banc d'essai du langage reste, à l'heure actuelle, le projet GEPI qui a permis, au fur et à mesure de son avancement, d'étendre et de tester la version de leloup que nous présentons dans la suite. En plus de l'influence du projet GEPI, la définition du langage a bénéficié indirectement des travaux menés en parallèle dans le cadre de l'écriture de l'ouvrage sur les langages à objets [Masini 89].

Plan

Le chapitre commence par la présentation des mécanismes de base du langage (§5.1), définition des classes, instanciation, envoi des messages et accès aux variables. La deuxième partie décrit en détail l'héritage multiple de leloup (§5.2), algorithme de linéarisation du graphe d'héritage, règles d'héritage des méthodes et des variables ainsi que le mécanisme de définition d'exceptions qui permet de

1. leloup est une marque déposée par le projet GEPI !

traiter des cas particuliers d'héritage. La partie suivante (§5.3) présente les possibilités de leloup pour vérifier dynamiquement le type et les propriétés des objets. A la manière des langages de frames, il est possible de définir des réflexes et ainsi de contraindre les variables associées aux objets. Avant de conclure, la quatrième partie (§5.4) regroupe différents outils qui complètent la panoplie d'outils déjà offerte par Le-Lisp. Un compilateur de programme leloup ainsi qu'un utilitaire qui permet de réutiliser les classes définies avec les primitives objets de Le-Lisp ...

5.1 Les classes leloup

A la manière de Ceyx [Hullot 83a] ou Alcyone [Hullot 85], leloup est construit comme une couche objets au dessus de Le-Lisp [Chailloux 84]. La puissance de LISP ainsi que les nombreux outils de l'environnement, comme le débogueur ou le pisteur, restent disponibles. Comme en LISP ou en Smalltalk, l'interactivité est un point fort de leloup.

Les classes leloup possèdent une ou plusieurs superclasses, des variables d'instances, des variables de classe ainsi que des méthodes. Cette première section présente, à l'aide de l'exemple de la gestion du stock d'un grand magasin, la définition des classes, la définition des méthodes, l'envoi des messages, l'accès aux variables et l'instanciation. Les problèmes liés à l'héritage multiple seront traités ultérieurement (§5.2).

Une classe leloup est définie à l'aide de la fonction *classe* dont le schéma général d'utilisation est donné figure 5.1. Différents mots clés, en caractères italiques, précèdent, la liste des superclasses, *-superClasses*, la description des variables d'instance, *-variables*, la description des variables de classe, *-variablesDeClasse*, et la liste des exceptions à l'héritage *-exceptions*[2].

Sans plus attendre, définissons une première classe, **Article** et ses quatre variables d'instance, **référence**, **désignation**, **prixHt** et **quantité** :

```
( classe Article
    -superClasses (objet)
    -variables
        (référence
         désignation
         prixHT
         (quantité 0)))
```

L'unique superclasse d'**Article** est la classe **objet**. Cette classe est prédéfinie en leloup et ne possède ni superclasse ni variables d'instances. Elle contient quelques méthodes d'intérêt général, comme l'impression d'un objet par exemple[3]

2. L'utilisation de la liste des exceptions est détaillée dans la section consacrée à l'héritage (cf. § 5.2.6).

3. A la manière de la classe **objet**, il est possible de définir une classe sans aucune superclasse et ainsi reconstruire un monde *tout neuf*. En cela, la classe **objet** s'apparente à la classe **vanilla** du système Flavors [Fla 86] [Moon 86] construit pour la machine Lisp [Weinreb 81] [Greenblatt 84].


```
(classe <nomDeClasse>
      -superClasses
          (<nomDeClasse1>
           <nomDeClasse2>
           <nomDeClasse3>
          ...)
      [-variables
          (<descriptionVariable1>
           <descriptionVariable2>
           <descriptionVariable3>
           ... )]
      [-variablesDeClasse
          (<descriptionVariable1>
           <descriptionVariable2>
           <descriptionVariable3>
           ... )]
      [-exceptions
          (<synonyme1>
           <synonyme2>
           <synonyme3>
           ... )])
```

Figure 5.1. La syntaxe de définition d'une classe en leloup.

Les variables ne sont pas typées et possèdent ou non une valeur par défaut. Une variable sans valeur par défaut est décrite simplement par son nom. Lorsque l'on veut indiquer une valeur par défaut, la description de la variable est faite à l'aide d'une liste à deux éléments, le nom de la variable suivi de la valeur par défaut correspondante. Dans l'exemple, les variables **référence**, **désignation** et **prixHT** ne possèdent pas de valeur par défaut. La variable **quantité** est, quant-à elle, initialisée par défaut avec 0.

5.1.1 Instanciation

La classe est l'entité conceptuelle qui décrit l'objet. Sa définition sert de modèle pour construire ses représentants physiques appelés *instances*. Une instance est donc un objet particulier qui est créé en respectant les plans de construction donnés par sa classe. Celle-ci joue le rôle de moule permettant de reproduire autant d'exemplaires que nécessaire.

```
(creer <nomDeClasse>
       [<nomDeVariable1> <valeurInitiale1>]
       [<nomDeVariable2> <valeurInitiale2>]
       [<nomDeVariable3> <valeurInitiale3>]
       ... )
```

Figure 5.2. La fonction d'instanciation de leloup.

En leloup, toutes les classes sont instanciables à l'aide de la fonction *creer* qui retourne la nouvelle instance (cf. Fig. 5.2). Seul le premier argument est obligatoire, le nom de la classe à instancier :

```
? ( creer 'Article)
= Article[() () () 0]
```

L'affichage de l'objet retourné indique la classe d'appartenance de l'objet, **Article**, ainsi qu'entre crochets, les valeurs des différentes variables dans l'ordre de leur définition[4]. Les variables ne possédant pas de valeur par défaut sont initialisées avec la liste vide LISP, ().

Les variables du nouvel objet peuvent être initialisées à l'aide des arguments optionnels de la fonction *creer*, chaque variable à initialiser devant être suivie de sa valeur initiale. Par exemple, pour créer une instance en initialisant les variables **référence**, **désignation** et **prixHT** :

```
? ( creer 'Article
?       'référence    911
?       'désignation  "punaises en boîte de 100"
?       'prixHT       5)))
= Article[911 "punaises en boîte de 100" 5 0]
```

La valeur par défaut précisée lors de la définition de la classe peut être écrasée par une nouvelle valeur initiale :

```
? (setq kimono
?       ( creer 'Article
?             'référence    912
?             'désignation  "kimono coton blanc"
?             'prixHT       700
?             'quantité     50)))
= Article[912 "kimono coton blanc" 700 50]
```

Tous les arguments de la fonction *creer* sont évalués, le nom de la classe à instancier ainsi que les noms et les valeurs des variables à initialiser. Par exemple,

4. Cet affichage est le résultat de la méthode prin de la classe objet, la superclasse d'Article.


```
(send <sélecteur> <receveur> [<arg1> <arg2> <arg3> ...])
```

Figure 5.3. La fonction d'envoi de messages en leloup.

il est possible de passer en argument le nom d'une classe à instancier. Ainsi, dans la suite[5], la variable locale X est d'abord initialisée aléatoirement avec un nom de la classe, **objet** ou **Article**, puis, la fonction *creer* est appelée avec le contenu de cette variable comme nom de classe à instancier :

```
? (let ((X (if (= 5 (random 0 1000))
?              'Article
?            'objet)))
?    ( creer X))
= Article[() () () 0]
? ;;  Quelle chance !!
```

Dernier détail sur la fonction *creer* : les variables sont initialisées dans l'ordre de lecture des arguments de la fonction *creer*. Cet ordre peut être différent de l'ordre de définition des variables. Dans certains cas, ce point de détail peut avoir son importance. En particulier, si un réflexe (cf. § 5.3) est attaché à une des variables.

5.1.2 Envoi des messages

La communication entre les objets s'effectue uniquement par envoi de messages. L'envoi d'un message est réalisé à l'aide de la fonction *send* (cf. Fig. 5.3)[6].

Deux arguments sont indispensables pour effectuer l'envoi d'un message, l'objet receveur auquel est envoyé le message et le nom de la méthode que l'on veut déclencher, le sélecteur. Les arguments supplémentaires de la méthode sont optionnels et, à la manière des fonctions LISP, une méthode peut avoir un nombre variable d'arguments. De même, les méthodes ont un retour.

Par exemple, en supposant que la méthode **prixTTC** soit déjà définie pour la classe **Article**, voici comment on la déclenche :

5. Il vous reste deux lignes pour apprendre LISP... Heureusement, il existe d'excellents ouvrages tels que ceux de Jean-Jacques Girardot [Girardot 85] ou Harald Wertz [Wertz 85], en français, et ceux de David S. Touretzky [Touretzky 84] ou Patrick H. Winston et Berthold K.P. Horn [Winston 84a] ou Robert Wilensky [Wilensky 86], en anglais.

6. La fonction *send* de leloup s'utilise comme la fonction CEYX [Hullot 83b] [Colnet 86a] de même nom.

```
? ( send 'prixTTC ( creer 'Article 'prixHT 100))
= 118.6
```

Dans ce cas, le retour de la méthode constitue le résultat *intéressant* de l'envoi de message, le prix toutes taxes du receveur. Dans d'autres cas, ce retour peut être sans intérêt particulier, la méthode ne travaillant que par effet de bords sur les variables de l'objet receveur par exemple.

Comme pour la fonction *creer*, tous les arguments de la fonction *send* sont évalués[7]. Dans la suite, de nombreux exemples montrent l'utilisation de la fonction *send* qui n'est utilisable que lorsque l'on a défini une méthode...

5.1.3 Accès aux variables

Toutes les variables d'une classe sont accessibles en lecture et en écriture à l'aide de méthodes automatiquement définies à la création de la classe. A chaque variable correspond une méthode de même nom qui permet d'accéder en lecture et en écriture à cette variable. Ainsi, la classe `Article` possède automatiquement quatre méthodes, **référence**, **quantité**, **prixHT** et **désignation**.

Comme toutes les méthodes, les méthodes d'accès aux variables s'invoquent à l'aide de la fonction *send*. S'il s'agit d'un accès en lecture à une variable, *send* est utilisée avec deux arguments : le premier, le sélecteur, est le nom de la variable à lire et, le deuxième argument référence l'instance concernée :

```
? kimono
= Article[912 "kimono coton blanc" 700 50]
?
? ;; Lecture de la variable désignation :
? ( send 'désignation kimono)
= "kimono coton blanc"
```

S'il s'agit d'une écriture, la fonction *send* est utilisée avec un argument supplémentaire, la nouvelle valeur de la variable à écrire :

```
? ;; Ecriture de la variable quantité :
? ( send 'quantité kimono 100)
= 100
? kimono
= Article[912 "kimono coton blanc" 700 100]
```

5.1.4 Définition des méthodes

Les méthodes sont définies à l'aide de la fonction *demethod* (cf. Fig. 5.4). Le premier argument est un couple qui indique le nom de la classe et le sélecteur de la méthode que l'on définit pour cette classe. Le deuxième argument, la liste des paramètres de la méthode, commence par le nom du receveur. La suite, comme pour les fonctions LISP ordinaires, représente le corps de la méthode.

7. Ce qui permet en particulier de réaliser l'équivalent du `perform` Smalltalk [Goldberg 83].


```
(demethod (<classe> <sélecteur>) (<receveur> [<arg1> <arg2> ...])
    <corpsDeMéthode>)
```

Figure 5.4. La fonction d'envoi de messages en leloup.

Complétons par exemple la classe `Article` à l'aide de la méthode `prixTTC` qui calcule le prix toutes taxes comprises de l'objet receveur, `self` :

```
( demethod (Article prixTTC) (self)
    (* ( send 'prixHT self) 1.186))
```

Le corps de cette méthode consiste simplement à multiplier le prix hors taxes par `1.186` ce qui revient à ajouter 18.6 % au prix hors taxes. Comme dans le cas des fonctions LISP ordinaires, la dernière expression évaluée dans le corps d'une méthode constitue son résultat :

```
? ( send 'prixTTC ( creer 'Article 'prixHT 100))
= 118.6
```

Dans certains cas, le retour de la méthode peut être sans importance, en particulier lorsque la méthode ne travaille que par effets de bords en modifiant l'objet receveur par exemple. Les deux méthodes `retirer` et `ajouter` permettent respectivement d'ajouter ou de retirer une quantité, `q`, d'une instance de la classe `Article`. Elles modifient la valeur de la variable **quantité** :

```
( demethod (Article retirer) (self q)
    ( send 'quantité self (- ( send 'quantité self) q)))

( demethod (Article ajouter) (self q)
    ( send 'quantité self (+ ( send 'quantité self) q)))
```

Dans ce cas, le résultat de l'envoi de message correspond à la nouvelle valeur de la variable **quantité** car la méthode d'écriture d'une variable retourne la valeur affectée à cette variable :

```
? kimono
= Article[912 "kimono coton blanc" 700 100]
? ( send 'retirer kimono 50)
= 50
```

La nouvelle valeur de la variable **quantité**, 50, est effectivement mémorisée au niveau de l'objet lui-même :

```
? kimono
= Article[912 "kimono coton blanc" 700 50]
```

5.1.5 Définition des variables de classe

Comme Smalltalk, leloup permet de définir des *variables de classe*. Une variable de classe est partagée par l'ensemble des instances d'une classe et de ses sous-classes. Les variables de classe sont définies lors de la création de la classe grâce au mot clé *-variablesDeClasse* (cf. Fig. 5.1). Comme les variables d'instance, les variables de classe peuvent avoir une valeur par défaut que l'on fixe au moment de leur définition. De même l'accès à une variable de classe est réalisé par envoi de message, une écriture se distinguant d'une lecture par un argument supplémentaire donnant la nouvelle valeur de la variable. Toutefois, le receveur du message peut indifféremment être une instance de la classe ou la classe elle-même désignée par son nom :

Définissons par exemple la classe `ProduitLaitier`, sous-classe d'`Article` et possédant une variable de classe, `températureDeConservation` dont la valeur par défaut est 7 :

```
? ( classe ProduitLaitier
?       -superClasses (Article)
?       -variablesDeClasse
?           ((températureDeConservation 7)))
= ProduitLaitier
```

La classe elle même est interrogeable pour connaître la valeur de la variable `températureDeConservation` :

```
? ;;  Lecture de la variable de classe :
? ( send 'températureDeConservation ProduitLaitier)
= 7
? ;;  Ecriture de la variable de classe :
? ( send 'températureDeConservation ProduitLaitier 7.5)
= 7.5
```

Une instance peut aussi être interrogée pour manipuler la variable :

```
? (setq yogourth
?       ( creer 'ProduitLaitier
?               'référence 4012
?               'désignation "les Bulgares"))
= ProduitLaitier[4012 "les Bulgares" () 0]
?
? ;;  Lecture de la variable de classe :
? ( send 'températureDeConservation yogourth)
= 7.5
? ( send 'températureDeConservation ProduitLaitier 37.5)
= 37.5
? ( send 'températureDeConservation yogourth)
= 37.5
? ;;  Bulgare, vous avez dit Bulgare ?
```


5.1.6 Premier bilan

Seules quatre primitives de base de leloup ont été décrites jusqu'ici : *classe*, *creer*, *send* et *demethod*. Ces quatre fonctions suffisent pour, en LISP, adopter un style de programmation à la Smalltalk.

L'intégration des primitives objets dans un environnement LISP est très naturel et, les systèmes LISP modernes possèdent souvent une couche objet prédéfinie[8]. Un des points importants qui explique la facilité d'intégration d'une couche objet au dessus d'un interprète LISP[9] réside dans le fait que les variables ne sont pas typées. Ainsi, le symbol `X` peut référencer à un instant donné une instance d'`Article` puis, dans la suite une instance de `ProduitLaitier` par exemple, et ainsi de suite.

L'utilisation de la fonction *send* est systématique et uniforme. Qu'il s'agisse de l'accès à une variable d'instance, de l'accès à une variable de classe ou de l'activation d'une méthode ordinaire, il faut utiliser la fonction *send*. Ceci permet, lorsqu'on utilise une classe, de faire abstraction de la nature de la propriété utilisée. Par exemple, syntaxiquement, il est impossible de savoir si un envoi de message va se traduire à l'exécution par un accès à une variable d'instance, par un accès à une variable de classe ou par le déclenchement d'une méthode définie par l'utilisateur. Soit l'envoi de message suivant :

```
...
( send 'prixTTC X)
...
```

Cette instruction peut être imbriquée dans une boucle ou dans un test et il est impossible de dire quoi que se soit sur le type de l'objet désigné par `X`. De plus, si on passe plusieurs fois par cet endroit du programme, l'objet référencé par `X` peut changer. La seule chose que l'on puisse dire pour l'instant, c'est que si `X` désigne une instance de la classe `Article`, l'exécution de l'envoi de message provoque le déclenchement d'une méthode définie par l'utilisateur, la méthode `prixTTC` de la classe `Article`. Dans un cadre de prototypage, l'implantation d'une classe est souvent amenée à changer et, au cours du temps, la propriété `prixTTC` de la classe `Article` peut devenir, pourquoi pas, une donnée mémorisée par l'objet. Pour l'utilisateur de la propriété `prixTTC`, rien ne change car cette propriété s'utilise de la même façon que lorsqu'il s'agissait d'une méthode.

Comme Smalltalk, leloup ne permet pas de définir des méthodes privées, c'est à dire des méthodes dont l'usage est limité à l'intérieur de certaines classes seulement. Dans la pratique, il arrive couramment qu'une méthode soit définie comme une fonction interne, simplement pour faciliter l'implantation des méthodes *exportées* d'une classe. Par exemple, lorsqu'une séquence d'instructions se répète plusieurs

8. C'est le cas de Le-Lisp [Chailloux 86] et de COMMON LISP [Steele Jr 84] par exemple. Certains vont même jusqu'à considérer LISP comme un des pères de la programmation objets [Cointe 82] et placent LISP au même rang que Simula [Dahl 70] dans l'histoire de la programmation objets.

9. *"L'ordinateur doit donc se prêter aux phantasmes les plus divers, servir de support à l'imagination ...*
Aujourd'hui, LISP apparaît comme une base de logiciel, une boule d'argile dont le programmeur fait ce qu'il veut." Pierre Cointe, [Cointe 82]

fois, il est de coutume d'en faire une méthode pour mettre en facteur la séquence d'instructions répétée. Si la méthode créée ne correspond pas à une action à part entière sur l'objet et ne respecte pas la cohérence de l'objet par exemple, il convient de n'utiliser cette méthode qu'à l'intérieur de la classe elle-même. La limitation de l'utilisation d'une méthode n'est pas prise en compte par le langage, toute méthode peut être invoquée à partir de n'importe quelle classe. En leloup comme en Smalltalk, la seule façon de protéger une méthode *dangereuse* contre les utilisations abusives consiste à la marquer d'un commentaire particulier.

Si, dans un cadre de prototypage, cette solution se révèle acceptable, dans un cadre industriel, une solution permettant de définir des procédure à usage interne est souhaitable. Eiffel est de ce point de vue un excellent langage pour l'industrie ou pour les applications nécessitant un haut niveau de sécurité [Meyer 88b] [Meyer 88a].

5.2 Héritage

Une classe peut avoir certaines caractéristiques en commun avec une ou plusieurs autres classes. Leur ensemble forme ainsi une hiérarchie dans laquelle chaque *sous-classe* partage les connaissances des classes dont elle est une spécialisation. On dit qu'elle *hérite* des méthodes et des données de ses *superclasses*.

Il serait tout à fait possible de gérer le stock avec la seule classe créée pour l'instant, chaque article en magasin en devenant une instance particulière. Cette façon de faire ne serait cependant ni pratique, ni économique pour implanter les spécificités de chaque type d'article. En effet, la classe `Article` devrait alors regrouper l'union des variables et des méthodes nécessaires pour caractériser et manipuler tous les types d'articles possibles, chaque instance n'en utilisant effectivement qu'un sous-ensemble. Par exemple, une variable `taille` est nécessaire pour enregistrer la taille d'une chemise, mais elle est superflue pour un téléviseur. Dans le même ordre d'idées, la méthode `prixTTC` ne convient pas pour tous les articles, les produits dits *de luxe* étant soumis à un taux de TVA plus élevé que les produits de consommation courante.

Les variables et les méthodes détenues par la superclasse sont partagées par ses sous-classes. Bien sûr, au niveau physique, les duplications inutiles sont évitées, mais conceptuellement tout se passe *comme si* les informations de la superclasse étaient recopiées dans les sous-classes : une sous-classe *hérite* des informations de sa superclasse.

Enrichissement

La spécialisation d'une classe peut être réalisée selon deux techniques. La première est l'enrichissement : la sous-classe est dotée de nouvelles variables et/ou de nouvelles méthodes représentant les caractéristiques propres au sous-ensemble d'objets ainsi décrit. Par exemple, le problème de la taille des vêtements est réglé en dotant la classe `Article` d'une sous-classe `Vêtement` avec des variables supplémentaires décrivant les spécificités des vêtements, `taille` et `coloris` (cf. Fig. 5.5). Par héritage, la sous-classe `Vêtement` compte en fait six variables d'instance, `référence`, `désignation`, `prixHT`, `quantité`, `taille`, `coloris`, et quatre méthodes, `prixTTC`, `prixTransport`, `retirer` et `ajouter` :


```
(classe Vêtement
  -superClasses
    (Article)
  -variables
    (taille
     coloris))
```

```
(classe ArticleDeLuxe
  -superClasses
    (Article))

(defmethod
  (ArticleDeLuxe prixTTC) (self)
    (* (send 'prixHT self) 1.33))
```

Figure 5.5. Définitions des classes `Vêtement` et `ArticleDeLuxe`.

```
? (setq C17 ( creer 'Vêtement
?                   'désignation "pantalon"
?                   'prixHT 100
?                   'coloris 'blue))
= Vêtement[() "pantalon" 100 0 () blue]
? ( send 'quantité C17)
= 0
? ( send 'prixTTC C17)
= 118.6
```

Substitution – masquage

La seconde technique est la substitution qui consiste à donner une nouvelle définition à une méthode héritée, lorsque celle-ci se révèle inadéquate pour l'ensemble des objets décrits par la sous-classe. Par exemple, en créant une sous-classe pour les articles "de luxe", il est possible de définir une nouvelle méthode pour le calcul du prix TTC, avec le taux de TVA approprié (cf. Fig. 5.5). La sous-classe `ArticleDeLuxe` intègre les mêmes informations que la classe `Article`, si ce n'est que la méthode `prixTTC` est *masquée* par une nouvelle définition :

```
? (setq caviar ( creer 'ArticleDeLuxe
?                      'désignation "Petites billes noires"
?                      'prixHT 1000))
= ArticleDeLuxe[() "Petites billes noires" 1000 0]
? ( send 'prixTTC C17)
= 1330.
```

Dans certains cas, l'héritage simple n'est plus une solution satisfaisante. Des classes appartenant à des branches différentes de l'arbre d'héritage peuvent avoir besoin de partager certaines propriétés. En leloup, une classe peut avoir plusieurs superclasses directes, l'héritage est *multiple*. L'ensemble des classes n'est plus structuré en arborescence mais forme un graphe orienté sans circuit.

5.2.1 De l'héritage multiple

Par exemple, un téléviseur et un aspirateur ont des caractéristiques communes en ce qui concerne l'alimentation électrique (voltage, puissance, etc.), mais, un téléviseur est considéré comme un article de luxe et est soumis à une TVA de 33%, ce qui n'est pas le cas des équipements d'électroménager.

Bien sûr, le problème peut être résolu en dotant les classes **Téléviseur** et **Electroménager** des mêmes variables (**voltage**, **puissance**, etc.), mais cela entraîne des duplications d'informations. Une solution bien plus en rapport avec la philosophie objet, car permettant de profiter au mieux des classes déjà définies, consiste à donner à une classe la possibilité d'hériter directement de plusieurs autres classes.

Une façon de résoudre le problème qui nous préoccupe consiste alors à créer une classe **AlimElectrique** regroupant les informations propres à l'alimentation électrique, puis d'en faire hériter la classe **Téléviseur** d'une part et la classe **ElectroMénager** d'autre part (cf. Fig. 5.6).

Une classe hérite de l'union des variables et des méthodes de ses superclasses. Ainsi, la classe **Téléviseur** hérite de la classe **AlimElectrique** les informations relatives à son alimentation électrique, mais hérite aussi de la classe **ArticleDeLuxe** les informations relatives à ses conditions de taxation :

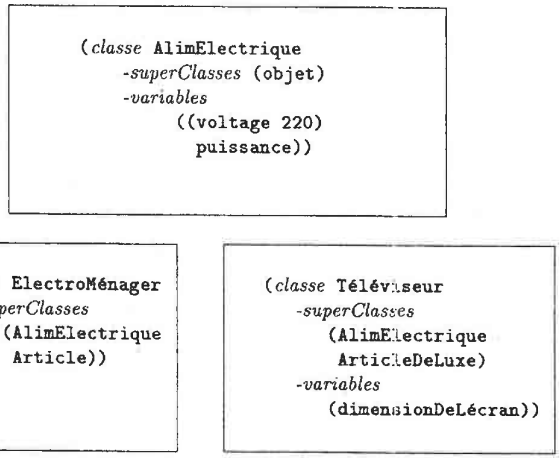


Figure 5.6. Héritage multiple.

```
? (setq A2 ( creer 'Téléviseur 'prixHT 100))
= Téléviseur[() () 100 0 220 () ()]
? ( send 'prixTTC A2)
= 133.
? ;;  Résultat de la méthode définie dans ArticleDeLuxe.
?
? (setq Aspirateur ( creer 'Electroménager 'prixHT 100))
= Electromenager[() () 100 0 220 ()]
? ( send 'prixTTC Apirateur)
= 118.6
? ;;  Résultat de la méthode définie dans Article.
?
? ( send 'dimensionDeLecran Aspirateur)
** send : méthode indéfinie : dimensionDeLecran ElectroMenager
```

L'avantage de l'héritage multiple est d'accroître encore la modularité des programmes, et donc d'en faciliter la mise au point et la maintenance. Il permet également un gain de place appréciable par mise en commun d'informations, programmes et données, qu'il faudrait sinon dupliquer dans plusieurs classes. En contrepartie, le contenu des classes et leur agencement doivent être élaborés avec le plus grand soin, sous peine d'être inutilisables. Lorsque le graphe d'héritage devient complexe, son exploitation requiert un minimum d'expérience (cf. Fig. 5.7).

5.2.2 Le lien d'héritage

La relation d'héritage peut être interprétée de différentes façons selon l'usage qui en est fait. Ronald Brachman en a répertorié plus d'une dizaine [Brachman 83], parmi lesquelles se détachent trois points de vue principaux :

- *Point de vue ensembliste* : la relation d'héritage décrit une inclusion d'ensembles. La classe `ArticleDeLuxe` héritant de la classe `Article`, l'ensemble des éléments (instances) de la classe `ArticleDeLuxe` est inclus dans l'ensemble des éléments de la classe `Article` :

$$\forall\, x,\quad x \in \mathit{ArticleDeLuxe} \;\Rightarrow\; x \in \mathit{Article}$$

- *Point de vue logique* [Hayes 79] : l'appartenance à une classe est exprimée par un prédicat. $\mathit{Article}(x)$ est vrai si l'élément x appartient à la classe `Article`. La formule *un article de luxe est un article* se traduit alors par :

$$\forall\, x,\quad \mathit{ArticleDeLuxe}(x) \;\Rightarrow\; \mathit{Article}(x)$$

- *Point de vue conceptuel* : la relation d'héritage indique une spécialisation. Par exemple, la relation qui lie une espèce animale à ses sous-espèces est de cette nature :

ArticleDeLuxe est une sorte d'Article

Les classes les plus générales se trouvent près de la racine du graphe d'héritage tandis que les classes décrivant des objets particuliers se trouvent près des feuilles. C'est sous cet angle que l'héritage a été considéré jusqu'ici.

Toutes les relations correspondant à ces différentes interprétations sont *transitives* et déterminent un *ordre* sur les classes. Il ne sera donc plus fait mention que de "la" relation d'héritage, indépendamment de la sémantique qui lui est attachée.

Le graphe de la relation d'héritage est sans circuit et possède toujours une "racine", c'est-à-dire un nœud sans père, représentant la classe qui détient le comportement général de tous les objets, la classe `objet` dans le cas de l'exemple. Si le graphe se construit de haut en bas, par spécialisations successives à partir de cette racine, la relation d'héritage lie les objets spécialisés à ceux qui les généralisent, de bas en haut.

Chaque classe possède son propre graphe d'héritage restriction du graphe d'héritage complet à la seule hiérarchie d'héritage de la classe.

En leloup, l'héritage s'applique uniformément à toutes les *propriétés* des objets, variables d'instance, variables de classe et méthodes. En d'autres termes, une classe hérite des méthodes de ses superclasses de la même façon qu'elle hérite des variables d'instance ou de ses variables de classe[10].

10. Cet aspect n'est pas fondamental, même si la plupart des langages imposent des restrictions sur l'héritage de l'une ou l'autre des propriétés. Par exemple, le masquage des variables est fréquemment interdit.

5.2.3 Linéarisation

Avec l'héritage multiple, certains objets ne sont plus comparables, comme les classes `AlimElectrique` et `Article` par exemple (cf. Fig. 5.7). Le masquage ne s'applique donc plus puisque, par exemple, il n'est pas possible de déterminer si `Article` se trouve avant ou après `AlimElectrique` dans le graphe d'héritage de `Téléviseur`. Les éventuelles propriétés homonymes de `AlimElectrique` et `Article` sont alors en conflit pour l'héritage de `Téléviseur`.

En revanche, les propriétés homonymes des objets comparables ne sont pas en conflit, car la propriété du plus petit objet, au sens de la relation d'ordre, masque les autres.

Il n'existe pas de méthode simple pour résoudre de tels conflits : aucun parcours de la hiérarchie de l'objet n'est satisfaisant *a priori*. Bien qu'elle paraisse la plus naturelle, la solution consistant à changer les noms des propriétés directement dans les classes est inacceptable car elle va à l'encontre des principes d'encapsulation et de réutilisabilité qui font la force des langages à objets.

Il faut donc établir des heuristiques permettant de donner une sémantique au mécanisme d'héritage et assurer l'unicité du résultat. En d'autres termes, le parcours de la hiérarchie doit respecter certains critères garantissant que la liste de priorité obtenue respecte au mieux les intentions du concepteur de la hiérarchie des classes. Ces critères relèvent du "bon sens", mais il est certain qu'ils font toujours intervenir une certaine part de subjectivité.

5.2.4 L'ordre de l'héritage

Le premier critère est fondamental.

▷ *La liste de priorité doit être ordonnée par une relation d'ordre qui est une extension linéaire*[11] *de la fermeture transitive de la relation d'héritage.*

En d'autres termes, l'ordre imposé par la relation d'héritage doit être préservé dans la liste de priorité : la classe `ArticleDeLuxe` étant sous-classe de `Article`, elle doit précéder `Article` dans la liste de priorité de la classe `Téléviseur` par exemple.

Ce principe garantit que le masquage est effectif et donc que le mécanisme d'héritage produit bien un élément de l'ensemble des propriétés en conflit.

La multiplicité de l'héritage

Les extensions linéaires d'un ordre partiel étant fort nombreuses, d'autres critères sont nécessaires. Le premier fait intervenir une autre relation sur les classes, qui était implicite jusqu'à présent : la *multiplicité*.

11. Une relation d'ordre total $\mathcal{RE}$ sur un ensemble E est une extension linéaire d'une relation d'ordre partiel $\mathcal{R}$ si et seulement si : $x \leq_{\mathcal{R}} y \;\Rightarrow\; x \leq_{\mathcal{RE}} y$.

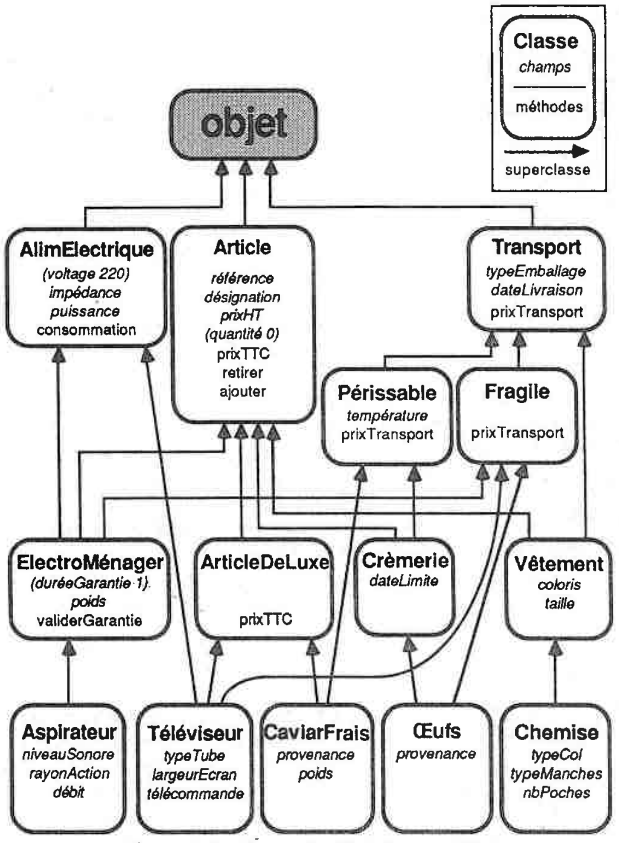


Figure 5.7. Un graphe d'héritage multiple.

```
(classe Téléviseur
   -superClasses
      (AlimElectrique ArticleDeLuxe Fragile)
   . . . )
```

Figure 5.8. La multiplicité locale de la classe `Téléviseur`.

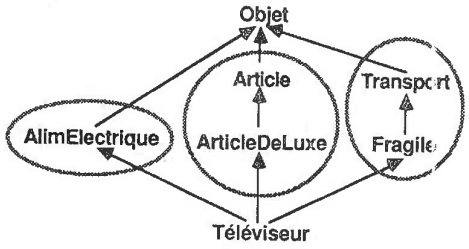


Figure 5.9. Les modules du graphe d'héritage de la classe `Téléviseur`.

La multiplicité locale d'une classe est la relation d'ordre qui existe entre les superclasses directes de cette classe. C'est une relation d'ordre total, qui est définie par l'ordre dans lequel sont données les superclasses lorsque la classe est créée (cf. Fig. 5.8). Par exemple, les superclasses de `Téléviseur` étant dans l'ordre `AlimElectrique`, `ArticleDeLuxe` et `Fragile`, elles doivent apparaître dans cet ordre précis dans la liste de priorité de `Téléviseur`.

La multiplicité globale d'un objet est la réunion des multiplicités locales de ses ascendants. Comme la relation d'héritage, elle constitue une relation d'ordre partiel. Le second critère peut maintenant être formulé en ces termes :

▷ *La liste de priorité doit être ordonnée par une relation d'ordre qui est une extension linéaire de la fermeture transitive de la multiplicité globale.*

La modularité

Le dernier critère est fondé sur une caractéristique essentielle des langages à objets : la modularité. Par exemple, dans le graphe d'héritage de la classe `Téléviseur`, il paraît assez naturel de distinguer trois sous-graphes, ou *modules*, correspondant à trois points de vue sur l'objet : celui de son alimentation électrique (classe `AlimElectrique`), celui de son statut d'article (classes `Article` et `ArticleDeLuxe`), et celui de son transport (classe `Transport` et `Fragile`) (cf. Fig. 5.9). Une décomposition aussi parfaite n'est toutefois pas toujours possible, notamment pour la classe `Œufs`.

Le parcours ne doit pas mélanger les sous-graphes. Autrement dit :

▷ *La liste de priorité doit préserver la modularité du graphe d'héritage.*

Chaque module forme lui-même un graphe d'héritage d'objet, avec sa propre racine correspondant à l'objet le plus général du point de vue qu'il représente. Pour conserver la modularité, le parcours doit être la composition des parcours des différents modules. Dans la liste de priorité finale, les objets d'un même module doivent donc former une sous-liste d'objets connexes, obtenue par parcours du module.

Quelques parcours

Examinons maintenant quelques parcours à la lumière de ces critères, pour les caractériser et faire un choix.

Le parcours en *profondeur d'abord*, prenant les pères d'un objet dans l'ordre de la multiplicité, n'est pas satisfaisant, car il viole le premier principe : en particulier, la racine du graphe n'est pas visitée en dernier. Voici par exemple les listes de priorité obtenues pour les classes **Téléviseur** et **Œufs** de la figure 5.7 :

Téléviseur → **Téléviseur, AlimElectrique, objet, ArticleDeLuxe, Article, Fragile, Transport**

Œufs → **Œufs, Crèmerie, Article, objet, Périssable, Transport, Fragile**

Le parcours en *largeur d'abord* a été adopté pour le langage Mering notamment [Ferber 83]. Même s'il semble mieux résoudre le problème pour la classe **Œufs**, il tombe dans le même travers que le parcours précédent pour la classe **Téléviseur**. De toute façon, il ne respecte pas la modularité :

Téléviseur → **Téléviseur, AlimElectrique, ArticleDeLuxe, Fragile, objet, Article, Transport**

Œufs → **Œufs, Crèmerie, Fragile, Article, Périssable, Transport, objet**

Les recherches en profondeur d'abord permettent néanmoins d'obtenir des extensions linéaires, comme l'ont montré R. Ducournau et M. Habib [Ducournau 89]. Une version simplifiée de l'algorithme proposé a été retenue pour leloup. Elle consiste à retenir les nœuds du graphe dans l'ordre d'empilement du parcours en profondeur d'abord. Les nœuds déjà visités ne sont pas marqués et la liste obtenue contient donc des occurrences multiples. La figure 5.10 schématise le déroulement de cette partie de l'algorithme sur le graphe d'héritage de la classe **Œufs**.

La liste obtenue est ensuite parcourue à l'envers, en supprimant au fur et à mesure les éléments déjà rencontrés au moins une fois. De cette façon, seule la dernière occurrence de chaque élément est conservée dans la liste finale. La liste de priorité de la classe **Œufs** devient alors :

Œufs → **Œufs, Crèmerie, Article, Périssable, Fragile, Transport, objet**

Pour la classe **Téléviseur**, elle a la forme suivante :

Pile		Nœuds retenus
Œufs	→	*Œufs*
Œufs Crèmerie	→	*Crèmerie*
Œufs Crèmerie Article	→	*Article*
Œufs Crèmerie Article objet	→	*objet*
Œufs Crèmerie Article		
Œufs Crèmerie		
Œufs Crèmerie Périssable	→	*Périssable*
Œufs Crèmerie Périssable Transport	→	*Transport*
Œufs Crèmerie Périssable Transport objet	→	*objet*
Œufs Crèmerie Périssable Transport		
Œufs Crèmerie Périssable		
Œufs Crèmerie		
Œufs		
Œufs Fragile	→	*Fragile*
Œufs Fragile Transport	→	*Transport*
Œufs Fragile Transport objet	→	*objet*
Œufs Fragile Transport		
Œufs Fragile		
Œufs		

Figure 5.10. Ordre d'empilement des nœuds pendant le parcours en profondeur d'abord du graphe d'héritage de la classe **Œufs**.

Téléviseur → **Téléviseur, AlimElectrique, ArticleDeLuxe, Article, Fragile, Transport, objet**

Modularité et multiplicité sont bien respectées.

Il existe cependant des cas où la liste produite n'est guère satisfaisante (cf. Fig. 5.11). Le premier cas est un contre-exemple générique pour lequel il n'existe pas d'extension linéaire qui respecte la multiplicité. Les deux autres anomalies sont dues à la présence de circuits, qui reflètent une contradiction entre la relation d'héritage et la relation de multiplicité d'une part (2), une contradiction interne à la relation de multiplicité d'autre part (3). Ces contradictions remettent en cause la sémantique de l'héritage telle que l'ont définis les critères adoptés, et la façon même d'aborder le problème. S'il semble évident d'interdire les circuits pour la relation d'héritage, dans quelle mesure faut-il interdire les contradictions entre multiplicité et héritage ? Une autre approche introduirait de toute façon d'autres restrictions, sans compter que l'utilisateur désirera toujours avoir la possibilité de traiter différemment certains cas particuliers (cf. § 5.2.6).

Une solution universelle demanderait une autre technique, sans doute non al-

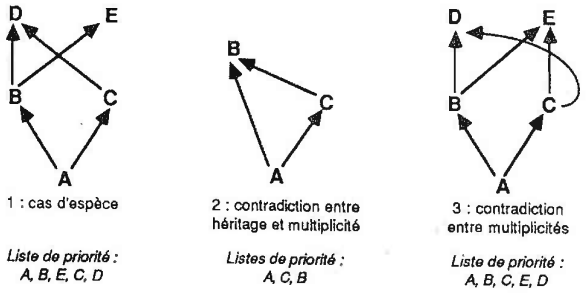


Figure 5.11. exemples de graphes d'héritage où la liste de priorité obtenue ne respecte pas les trois critères donnés.

gorithmique. La résolution des conflits ne peut être fiable que si elle prend en compte les connaissances liées à l'application. Seul le concepteur, ou un expert du domaine, possède la compétence requise. Appliquer systématiquement une solution par défaut ne peut jamais régler correctement chaque cas particulier.

Si la technique présentée ne donne pas toujours un résultat satisfaisant, elle a au moins le mérite d'exister et de posséder une sémantique cohérente.

5.2.5 Héritage des méthodes et des variables

Toutes les propriétés d'une classe, méthodes, variables d'instance et variables de classe sont héritées en respectant l'ordre de la liste de priorité obtenue avec l'algorithme précédant. La méthode `listeDePriorite` de la classe `objet` retourne la liste de priorité de l'objet receveur qui peut être indifféremment une instance ou le nom d'une classe :

```
? ( send 'listeDePriorite ( creer 'Téléviseur))
= (Téléviseur AlimElectrique ArticleDeLuxe Article Fragile Transport
  objet)
?
? ( send 'listeDePriorite Téléviseur)
= (Téléviseur AlimElectrique ArticleDeLuxe Article Fragile Transport
  objet)
?
? ( send 'listeDePriorite objet)
= (objet)
```

La fonction *send*, chargée du déclenchement des méthodes, parcourt la liste de priorité et déclenche la première méthode trouvée. Ainsi, lorsqu'un message est

envoyé à une instance de la classe `Téléviseur`, la méthode à exécuter est d'abord recherchée dans le dictionnaire des méthodes de `Téléviseur`, puis, dans l'ordre, les classes `AlimElectrique`, `ArticleDeLuxe`, `Article` et enfin `objet` sont inspectées. Seule la première méthode trouvée est exécutée.

Pour les variables, la règle d'héritage est la même que pour les méthodes. Si deux variables provenant de deux superclasses différentes portent le même nom, elles sont fusionnées en une seule. Lorsque plusieurs classes sont concurrentes pour fixer la valeur par défaut de la variable, la classe la plus proche du début de la liste de priorité est choisie. Par exemple, si on veut redéfinir la valeur par défaut de `prixHT` pour la classe `ArticleDeLuxe` :

```
? ( classe ArticleDeLuxe
?       -variables
?           ((prixHT 1000)))
= ArticleDeLuxe
?
? (setq parfum ( creer 'ArticleDeLuxe))
= ArticleDeLuxe[() () 1000 0]
```

5.2.6 Exceptions à l'héritage

Le mécanisme d'héritage, tel que nous l'avons vu jusqu'à présent est un mécanisme global, dans le sens où, en choisissant d'hériter d'une classe, on hérite de l'ensemble de ses propriétés. Par exemple, il n'est pas possible que la classe des autruches hérite de toutes les propriétés de la classe des oiseaux, *sauf* de celles qui concernent le vol. Disposer d'un tel mécanisme est cependant primordial en représentation des connaissances et a motivé de nombreuses études, notamment dans le cadre des réseaux sémantiques [Etherington 87] [Fikes 87] et des langages de frames [Touretzky 86] [Ducournau 89].

Le mécanisme d'exception à l'héritage de leloup permet de traiter certaines exceptions au cas par cas en les énumérant dans la liste qui suit le mot clé *exceptions* de la définition d'une classe (cf. Fig. 5.1).
Une exception a la forme suivante :

```
(<sélecteurException> (<nomDeClasse> <sélecteur>))
```

Le mécanisme d'exception ne remet pas en cause le calcul des listes de priorité. Chaque exception se traduit par la création d'une méthode relais, de sélecteur `<sélecteurException>`, définie dans la classe elle-même. Le corps de cette méthode relais active la méthode `<sélecteur>` de la classe `<nomDeClasse>` en lui transmettant tous ses arguments, receveur compris.

Par exemple, pour définir dans la classe `ArticleDeLuxe` la méthode `prixAvecTaxeNormale` devant donner le même résultat que la méthode `prixTTC` de la classe `Article` :

```
? ( classe ArticleDeLuxe
?       -superClasses (Article)
?       -exceptions
?           ((prixAvecTaxeNormale (Article prixTTC))))
= ArticleDeLuxe
?
? (setq A2 ( creer 'Téléviseur 'prixHT 100))
= Téléviseur[() () 100 0 220 () ()]
? ( send 'prixTTC A2)
= 133.
? ;; Résultat de la méthode définie dans ArticleDeLuxe.
?
? ( send 'prixAvecTaxeNormale A2)
= 118.6
? ;; Résultat de la méthode définie dans Article.
```

Inversement, la classe **Article** peut utiliser la méthode **prixTTC** de la classe **ArticleDeLuxe** :

```
? ( classe Article
?       -superClasses (objet)
?       -variables
?           (référence
?            désignation
?            prixHT
?            (quantité 0))
?       -exceptions
?           ((prixAvecTaxeDeLuxe (ArticleDeLuxe prixTTC))))
= Article
? ( send 'prixAvecTaxeDeLuxe ( creer 'Article 'prixHT 100))
= 133.
```

Ou encore, la classe **ArticleDeLuxeDétaxé** qui utilise la méthode **prixTTC** des articles ordinaires :

```
? ( classe ArticleDeLuxeDétaxé
?       -superClasses (ArticleDeLuxe)
?       -exceptions
?           ((prixTTC (Article prixTTC))))
= ArticleDeLuxeDétaxé
?
? ( send 'prixTTC ( creer 'ArticleDeLuxeDétaxé prixHT 100))
= 118.6
```

5.3 Réflexes et contraintes de type

5.3.1 Les méthodes de vérification de type

Lorsqu'on écrit le corps d'une méthode, le type du receveur est déterminé par la classe où est définie cette méthode et, naturellement, on écrit le corps de la méthode en se limitant au cas traité par la méthode.

Dans certains cas, malgré le mécanisme d'envoi de message, il est utile de tester explicitement le type d'un objet. Par exemple pour les arguments optionnels d'une méthode, ceux qui suivent le receveur, aucune supposition ne peut être faite en ce qui concerne leur classe d'appartenance. La méthode **type**, de la classe **objet** permet de connaître la classe d'appartenance d'un objet :

```
? ( send 'type ( creer 'Article))
= Article
? ( send 'type ( creer 'objet))
= objet
? (equal 'objet ( send 'type ( creer 'Article)))
= ()
? (equal 'Article ( send 'type ( creer 'Article)))
= t
```

En programmation objets, il est utile de considérer le type qui est associé à un objet dans un sens plus large. Du fait de l'héritage, une classe hérite des propriétés de ses superclasses et, il est de coutume de considérer qu'une instance est simultanément du type de toutes ses superclasses. Ainsi, une instance de la classe **ArticleDeLuxe** est du type **ArticleDeLuxe**, mais est aussi du type **Article** et du type **objet**.

Lorsque l'on veut tester explicitement le type d'un objet, il vaut mieux utiliser cette notion de type au sens large. En effet, l'évolution de l'arbre des classes peut faire apparaître de nouvelles classes et ainsi remettre en cause un test de type au sens strict de l'appartenance à une classe. Par exemple, on peut très bien imaginer la création d'une nouvelle sous-classe de la classe **ArticleDeLuxe**, **ArticleDeLuxeDétaxé**.

Pour tester le type au sens large d'un objet, leloup associe à chaque classe un sélecteur particulier. Les méthodes correspondantes permettent de savoir si une instance est soit une instance de la classe elle-même soit une instance de ses sous-classes. Le nom de sélecteur est crée systématiquement en suffixant le nom de la classe par le caractère point d'interrogation. Par exemple, le sélecteur **Article?** permet de savoir si un objet est instance de la classe **Article** ou d'une de ses sous-classes :

```
? ( send 'Article? ( creer 'Article))
= Article[() () () 0]
?
? ( send 'Article? ( creer 'ArticleDeLuxe))
= ArticleDeLuxe[() () () 0]
?
? ( send 'ArticleDeLuxe? ( creer 'Article))
= ()
?
? ( send 'objet? ( creer 'Article))
= Article[() () () 0]
```

L'implantation de ce mécanisme de test repose sur l'utilisation de la liaison dynamique. Pour chaque classe, deux méthodes de même sélecteur sont définies,

```
(reflexe (<nomDeClasse> <sélecteur>)
     [-lecture
          (lambda (self valeur)
               ...)]
     [-avantEcriture
          (lambda (self valeurAvant valeurApres)
               ...)]
     [-apresEcriture
          (lambda (self valeurAvant valeurApres)
               ...)])
```

Figure 5.12. Définition des réflexes en leloup.

une dans la classe racine de l'arbre d'héritage, `objet`, et l'autre dans la classe elle même. Le corps de la méthode définie dans la classe `objet` retourne invariablement la valeur logique fausse, `nil`. Celui de la méthode définie dans la classe concernée retourne invariablement une valeur logique vraie, le receveur lui même en l'occurrence[12]. Par exemple, pour la classe `Article`, les deux méthodes suivantes sont automatiquement définies :

```
( demethod (objet Article?) (self)
    nil)

( demethod (Article Article?) (self)
    self)
```

5.3.2 Définition de réflexes

Les réflexes sont des séquences d'instructions qui sont déclenchées automatiquement à la lecture ou à l'écriture d'une variable. En particulier, ils permettent d'implanter simplement des vérifications ou des contraintes de type sur les variables d'un objet.

La fonction *reflexe* permet d'attacher des réflexes à une variable d'instance ou à une variable de classe (cf. Fig. 5.12). On distingue trois catégories de réflexes, celui qui se déclenche à la lecture d'une variable, *-lecture*, celui qui se déclenche avant l'écriture, *-avantEcriture* et enfin celui qui se déclenche après l'écriture de la variable, *-apresEcriture*.

Il est possible d'utiliser simultanément les trois variantes de réflexes. Dans le corps des *lambda-expressions* correspondantes, on peut manipuler l'objet qui subit la lecture ou l'écriture à l'aide du premier argument, `self`. Le deuxième

12. En LISP, tout objet différent de `nil` est assimilé à une valeur logique vraie. Ceci permet souvent d'utiliser directement le résultat d'un prédicat comme argument d'un calcul.

argument de la lambda-expression désigne la valeur de la variable. En cas de réflexe d'écriture, il s'agit de la valeur précédant l'écriture. Le troisième argument des réflexes d'écriture désigne la valeur devant être écrite. Le retour des lambda-expression *-lecture* et *-apresEcriture* constituent le nouveau retour de la méthode en cas de lecture ou d'écriture.

5.3.3 Contraindre une variable

Par exemple, il est possible de contraindre une variable en indiquant l'ensemble des valeurs qu'elle peut prendre. Si on suppose que la taille d'un vêtement se limite à un ensemble de valeurs particulières, il est possible d'indiquer en extension cet ensemble de valeurs dans le corps du réflexe en écriture de la variable à restreindre. A chaque écriture de la variable `taille`, le réflexe vérifie l'appartenance de la valeur à écrire dans l'ensemble des valeurs possibles :

```
? ( reflexe (Vêtement taille)
?     -avantEcriture
?         (lambda (self valeurAvant valeurApres)
?             (unless (member valeurApres
?                             '(1 2 3 4 5 36 37 38 39 40 41 42 43 44 45))
?                 (error 'taille
?                        "Taille inconnue !."
?                        (list self valeurApres)))))
= taille

? (setq pantalon ( creer 'Vêtement 'taille 36))
= Vêtement[() () () () () 0 () 36]

? ( send 'taille pantalon 6)
** taille : Taille inconnue !. : (Vêtement[() () () () () 0 () 36] 6)

? ( send 'taille pantalon 37)
= 37
? pantalon
= Vêtement[() () () () () 0 () 37]
? ( creer 'Vêtement 'taille 12)
** taille : Taille inconnue !. : (Vêtement[() () () () () 0 () ()] 12)
```

Ou, par exemple, pour vérifier que la variable `quantité` de la classe `Article` est bien un nombre (`numberp`) positif :

```
? ( reflexe (Article quantité)
?     -avantEcriture
?         (lambda (self valeurAvant valeurApres)
?             (unless (and (numberp valeurApres)
?                          (>= 0 valeurApres))
?                 (error 'quantité
?                        "Quantité étrange !"
?                        (list self valeurApres)))))
= quantité
```

```
? ( creer 'Article 'quantité 100)
= Article[() () () 100]
?
? ( creer 'Article 'quantité -1)
** quantité : Quantité étrange ! : (Article[() () () 100] 6)
```

5.3.4 Cohérence des variables

Un réflexe peut aussi être utilisé pour lier deux variables d'un même objet. Par exemple, supposons que l'on veuille, pour des raisons d'efficacité, mémoriser aussi le prix toutes taxes d'un `Article` afin d'éviter de le recalculer plusieurs fois. On modifie la définition de la classe `Article` en lui ajoutant la variable `prixTTC` :

```
( classe Article
    -superClasses (objet)
    -variables
       (référence
        désignation
        prixHT
        prixTTC
        (quantité 0)))
```

Puis, pour être sûr qu'à chaque modification de la variable `prixHT`, la variable `prixTTC` soit à jour, on place un réflexe en écriture sur la variable `prixHT` qui effectue la mise à jours de la variable `prixTTC` :

```
? ( reflexe (Article prixHT)
?     -avantEcriture
?        (lambda (self valeurAvant valeurApres)
?            ( send 'prixTTC self (* 1.18 valeurApres))))
= prixHT

(setq sucre ( creer 'Article 'prixHT 1))
= Article[() () 1 1.18 0]

? ( send 'prixHT sucre 2)
= 2
? sucre
= Article[() () 2 2.36 0]
```

Dans cet exemple, l'implantation de la classe `Article` est modifiée : une méthode est devenue une variable. Les programmes qui utilisent cette classe n'ont pas à être modifiés car l'utilisation d'une méthode est indistinguable syntaxiquement de l'accès à une variable (cf. § 5.1.6).

5.3.5 Propriétés d'un objet

Il est possible d'utiliser les méthodes qui testent le type d'une variable à l'intérieur des réflexes. Une autre façon de contraindre une variable consiste à

s'assurer que l'objet qu'elle désigne est capable de répondre à un ou plusieurs messages. On peut ainsi faire abstraction de la classe d'appartenance d'un objet en ne s'intéressant qu'à ses propriétés.

La méthode `saisTuRepondre?` définie dans la classe `objet` permet de savoir si un objet possède ou non une propriété :

```
? ( send 'saisTuRepondre? ( creer 'Article) 'prixTTC)
= t
? ( send 'saisTuRepondre? ( creer 'Article) 'taille)
= ()
? ( send 'saisTuRepondre? ( creer 'Vêtement) 'taille)
= t
```

La méthode `saisTuRepondre?` possède un nombre variable d'arguments. Plusieurs propriétés peuvent suivre le receveur :

```
? ( send 'saisTuRepondre?
?        ( creer 'Vêtement)
?        'prixTTC
?        'ajouter
?        'taille)
= t
```

5.3.6 Modification du retour de l'accès aux variables

Les réflexes *-lecture* et *-apresEcriture* permettent de modifier le retour d'une méthode d'accès à une variable. Par exemple, il est possible de modifier, sans changer le contenu de la variable `prixHT`, le retour de la méthode d'accès :

```
? (setq caviar
?        ( creer 'ArticleDeLuxe
?                'prixHT 10))
= ArticleDeLuxe[() () 10 0]

? ;;  Le caviar n'est pas cher ces temps-ci :
? ( reflexe (ArticleDeLuxe prixHT)
?     -lecture
?        (lambda (self valeur)
?            (* 100 valeur)))
= prixHT

? ( send 'prixHT caviar)
= 1000
? caviar
= ArticleDeLuxe[() () 1000 0]
```

Les réflexes en lecture peuvent aussi être utilisés pour effectuer des mesures sur l'accès à une variable pour une classe donnée. Par exemple, si on désire mesurer le nombre des accès en lecture à la variable `quantité` de la classe `Article` :

```
? (setq compteurDacces 0)
= 0
? ( reflexe (Article quantite'')
?     -lecture
?         (lambda (self valeur)
?             (incr compteurDacces)
?             valeur))
= quantité

? (setq truc ( creer 'Article))
= Article[() () () 0]
? compteurDacces
= 1
( send 'quantité truc)
= 0
? compteurDacces
= 2
```

5.4 L'environnement de développement

Les outils de l'environnement Le-Lisp [Chailloux 86] sont utilisables en leloup. En particulier, le debogueur et le pisteur pas à pas continuent de fonctionner à merveille.

De même, le ramasse miette n'est pas modifié, les instances des classes leloup non référencées sont comme les objets LISP ordinaires, automatiquement collectées.

5.4.1 Classes externes

Le-Lisp possède sa propre couche objet qui permet dans un contexte d'héritage simple de définir des classes possédant des variables d'instance et des méthodes. Ces classes sont réutilisables en leloup et, une classe leloup peut être sous-classe d'une classe prédéfinie Le-Lisp.

Chaque classe Le-Lisp est définie par un *package* particulier. Pour utiliser une classe Le-Lisp, il faut préalablement la *déclarer* à l'aide de la fonction *classeExterne* :

```
( classeExterne <packageLe-Lisp>)
```

L'implantation de GÉPI utilise des classes prédéfinies Le-Lisp. Un exemple d'utilisation est donné au paragraphe 6.4.1.

5.4.2 Compilation

Un programme leloup, constitué d'une ou plusieurs classes peut être compilé. Une compilation leloup s'effectue en deux passages, le premier est un traitement spécifique des classes et des méthodes écrites en leloup, le deuxième est effectué

par le compilateur standard Le-Lisp. Après la compilation, aucune nouvelle classe ou nouvelle méthode ne peut être définie. Toute création soit d'une classe, soit d'une méthode après compilation se solde par une erreur fatale.

Cette restriction est courante dans les langages à objets compilés comme Eiffel ou Objective-C [Cox 86]. De même, en leloup, seules les applications dont l'arbre des classes est *figé* peuvent être compilés. Les programmes qui définissent des classes dynamiquement, c'est à dire au cours de leur exécution, ne doivent pas être compilés. Le compilateur utilise la structure de l'arbre des classes au moment de la compilation pour traduire les envois de messages.

La fonction *leloupCompile*, sans arguments, permet de compiler un programme leloup. Toutes les classes et toutes les méthodes en mémoire au moment de l'appel de cette fonction sont compilées.

Dans le pire des cas, la compilation rend constant le temps de recherche d'une méthode quelle que soit la classe où débute la recherche. L'accès à une méthode est réalisé à l'aide d'une seule indirection. l'héritage des méthodes est statique.

Dans certains cas, il est possible de connaître statiquement le type exact du receveur d'un envoi de message. Le compilateur peut alors substituer l'envoi de message correspondant par un appel fonctionnel direct à la fonction LISP qui réalise la méthode correspondante.

Mis à part le cas anecdotique suivant :

```
( send 'X ( creer 'Y))
```

il est possible de connaître le type exact du receveur dans d'autres cas de figure.

Soit `C`, une classe sans sous-classe et la méthode `M1` définie dans la classe `C` :

```
( demethod (C M1) (self ...)
    ...
    ( send 'M2 self ...)
    ...)
```

S'il n'existe aucune exception se redirigeant sur le sélecteur `M1` de la classe `C`, on connaît exactement le type du receveur `self`. On peut dans ce cas remplacer l'envoi de message de sélecteur `M2` par un appel fonctionnel direct à la fonction LISP qui réalise la méthode correspondante recherchée à partir de la classe `C`.

Lorsque la classe `C` possède une ou plusieurs sous-classes, le déclenchement de la méthode `M1` peut être provoqué par une instance de ces sous-classes. Cependant, lorsque le sélecteur `M1` est masqué dans *toutes* les sous-classes directes de `C`, on est ramené au cas précédent.

Lorsque l'on connaît exactement le type du receveur d'un envoi de message, il est possible de savoir si l'envoi de message est ou non un accès à une variable. S'il s'agit d'un accès à une variable, on peut savoir s'il s'agit d'une lecture ou d'une écriture en comptant le nombre d'arguments de la fonction *send* :

```
;; C'est une lecture :
( send 'prixHT ( creer 'Article))

;; C'est une écriture :
( send 'prixHT ( creer 'Article) nouvelleValeur)
```

Selon qu'il s'agit d'une lecture ou d'une écriture, l'envoi de message est traduit par une séquence d'accès différente.

Le gain en temps d'exécution d'un programme compilé par rapport à un programme interprété est très difficile à évaluer car plusieurs facteurs entrent en jeu :

- la structure du graphe d'héritage,
- la nature des instances manipulées à l'exécution,
- la fréquence des exceptions,
- et la fréquence d'utilisation de classes externes, non traitées par le compilateur.

Il est difficile dans ces conditions de donner des chiffres sans risquer de commettre une erreur. Une chose est sûre : un programme compilé va au moins aussi vite qu'un programme interprété et ...les résultats du programme compilé et du programme interprété sont identiques.

5.5 Pour conclure

Comme en Smalltalk, la programmation en leloup est essentiellement interactive et ainsi propice au prototypage : une méthode écrite peut être immédiatement testée. Les chapitres suivants contiennent la description de l'implantation de GÉPI, un exemple en grandeur nature qui utilise leloup.

L'utilisation des réflexes peut, dans certains cas, être une solution élégante pour modifier localement l'implantation d'une classe (cf. § 5.3.4) ou pour effectuer des mesures sur l'utilisation des objets (cf. § 5.3.6). Dans l'implantation actuelle de GÉPI, les réflexes ne sont que très rarement utilisés.

L'utilisation de métaclasses ne nous est pas apparue indispensable tout au moins en ce qui concerne le projet GÉPI. A notre avis, les métaclasses [Briot 85] [Briot 86] [Cointe 87] sont plus souvent destinées au concepteur du langage lui-même qu'aux utilisateurs finaux. Une version simplifiée de Smalltalk, Deltatalk [Borning 87], sans métaclasses, est qualifiée d'esthétique par ses auteurs.

leloup est plus proche de Smalltalk que d'Eiffel [Meyer 86c] dans la mesure où aucun typage statique des variables ou des arguments des méthodes n'est effectué. Dans un cadre de prototypage, de par notre expérience, nous pensons que les vérifications statiques effectuables sur un programme Eiffel [Meyer 87b], ou Simula [Meyer 79] [Birtwistle 73] par exemple n'accélèrent pas la détection des erreurs bénignes. En particulier, la détection d'un envoi de message auquel le receveur ne

sait pas répondre a souvent lieu lors du premier essai du programme. La vérification stricte du respect de l'interface d'une classe est l'avantage indiscutable des langages à objets à typage statique. La taille de l'équipe de développement d'un logiciel est selon nous le seul critère qui doit faire préférer un langage comme Eiffel à un langage objet de la famille Smalltalk comme leloup par exemple.

6
Le couplage arbre–texte

" La question fondamentale à laquelle se heurte tout débutant en informatique est de savoir comment l'électricité peut être domestiquée à un point de réalisation si spectaculaire. "

Francis Scheid [Scheid 74]

Afin d'intégrer dans un même éditeur les commandes de manipulation structurelle et les commandes de manipulation textuelle, il est nécessaire de réaliser un couplage entre la représentation textuelle et la représentation arborescente du document (chap. 1.2). Ce chapitre, consacré à la réalisation de ce couplage, contient la description des principales classes de GÉPI.

Plan

Après avoir présenté les problèmes liés au couplage (§6.1), les outils qui permettent de désigner une zone de texte sont décrits (§6.2). Les classes représentatives des différentes constructions d'une description de langage sont ensuite passées en revue (§6.3). Puis, la représentation des nœuds de l'arbre est étudiée (§6.4). Enfin, l'algorithme qui permet de retrouver la cohérence du couplage après une modification est détaillé (§6.5). Pour conclure, nous effectuons un bilan des différentes techniques objets utilisées dans ce chapitre (§6.6).

6.1 Principes du couplage arbre–texte

Pour l'implantation du manipulateur de document de GÉPI, le problème principal consiste à effectuer un couplage entre deux représentations, le texte d'un document et sa représentation arborescente. Ces deux représentations sont continuellement mises en correspondance.

6.1.1 Structure de la représentation arborescente

La représentation arborescente d'un document correspond *plus ou moins* à un arbre de syntaxe abstraite. En effet pour permettre une manipulation plus souple du manipulateur de document, cet arbre peut comporter des feuilles contenant du texte entré librement par l'utilisateur. En l'absence de telles feuilles, l'arbre représentatif d'un document est construit en regard de la syntaxe abstraite intégrée à la description du langage manipulé.

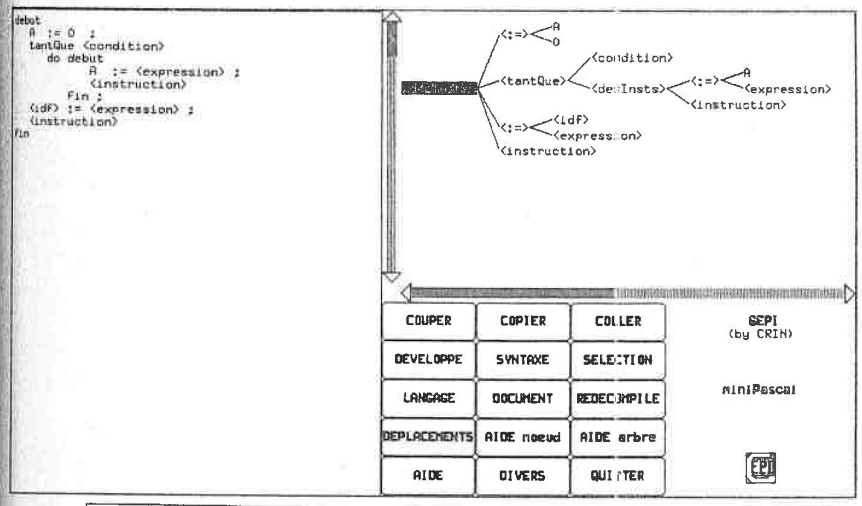
L'utilisation d'un arbre de syntaxe abstraite plutôt qu'un arbre de syntaxe concrète présente deux avantages. D'une part, l'arbre de syntaxe abstraite est moins encombrant que l'arbre de syntaxe concrète. D'autre part, l'arbre de syntaxe abstraite reflète de façon plus naturelle la structure syntaxique du document, ce deuxième argument étant néanmoins subjectif.

L'information contenue dans l'arbre ne permet pas à elle seule de reconstruire par décompilation tout le texte du document. En effet, les bruits du langage ne faisant pas partie de la description syntaxique à proprement parler, ces derniers ne sont pas mentionnés dans l'arbre. Par exemple, les commentaires d'un programme, dont la structure est décrite à l'aide de la rubrique *bruits* dans la description du langage (cf. § 2.2.4) ne font pas partie de l'arbre. Inversement, lorsque certaines parties du texte sont syntaxiquement incorrectes, ce texte ne permet pas *a priori* de reconstruire la représentation arborescente, la présence d'erreurs de syntaxe empêchant le bon fonctionnement de l'analyseur. Les deux représentations sont donc complémentaires.

Le principe du couplage consiste à étiqueter chaque nœud et chaque feuille de l'arbre avec une information permettant de désigner une zone dans le texte correspondant. Etant donnée la structure hiérarchique induite par la représentation arborescente, les zones d'influence associées aux différents nœuds possèdent les propriétés suivantes :

1. la zone d'influence d'un nœud englobe les zones d'influence de ses fils,
2. les zones d'influence de deux nœuds frères sont disjointes,
3. les zones d'influence de deux nœuds frères ne constituent pas nécessairement une zone continue.

La figure 6.1 visualise sur un exemple les différentes zones d'influence associées aux nœuds et aux feuilles d'un arbre. Le texte du programme manipulé avec l'éditeur de GÉPI est repris dans les deux parties inférieures de cette figure. Celle de gauche contient le texte sans la visualisation des zones d'influence des différents nœuds. Dans celle de droite, les différentes zones d'influence sont représentées entre crochets. Un crochet ouvrant marque le début d'une zone et, le crochet fermant associé marque la fin de cette zone.




```
debut
    A := 0  ;
    tantque <condition>
        do debut
              A := <expression>  ;
              <instruction>
        fin  ;
    <idf> := <expression>  ;
fin
```

```
[debut
    [[A]:=[0]] ;
    [tantque[<condition>]
        do[debut
              [[A]:=[<expression>]] ;
              [<instruction>]
        fin]] ;
    [[<idf>]:=[<expression>]] ;
fin]
```

Figure 6.1. Les zones d'influence associées aux nœuds et aux feuilles de l'arbre représentatif.

Cette mise en correspondance permet de calculer à quel nœud ou à quelle feuille est associée une position du document. Lorsque l'on pointe une position dans le document, le nœud ou la feuille correspondante est celui ou celle dont la zone d'influence est la plus petite zone qui contient cette position. Bien entendu, il n'est pas nécessaire d'explorer systématiquement toutes les zones pour trouver la bonne, un parcours descendant de l'arbre permettant de l'atteindre rapidement. L'algorithme est fort simple et consiste à partir de la racine de l'arbre représentatif à descendre le plus bas possible, tant que la position sélectionnée est comprise dans la zone associée au nœud exploré. Les trois propriétés mises en évidences précédemment permettent d'aller directement sur le nœud ou sur la feuille concernée.

Le même principe est utilisable lorsqu'il s'agit, partant d'une zone du texte du

document, de trouver le nœud ou la feuille concernée par cette zone. Dans tous les cas, le chemin à suivre en descendant l'arbre est direct.

6.1.2 L'environnement Aida

Comme pour la réalisation de l'éditeur interactif de livres (cf. § 4.3), la réalisation du manipulateur de documents de GÉPI doit commencer par la réalisation de plusieurs composants élémentaires : un éditeur pleine page, un éditeur d'arbres, des menus déroulants, des boutons... Heureusement pour nous, l'environnement Aida [Devin 88] [Aid 88] fournit des outils similaires à ceux de Smalltalk (cf. § 4.3.2).

Les différents composants de l'interface de GÉPI peuvent ainsi être décrits en réutilisant les classes prédéfinies de l'environnement Aida : éditeur pleine page, `medite`, éditeur d'arbres, `treeeditor`, barres de défilement, `scrollbar`, boutons, `button`... Comme en Smalltalk, la définition du manipulateur de document consiste à définir un *modèle* permettant d'assurer la communication entre les différents composants de l'interface de GÉPI. Ce modèle est chargé d'avertir les différents composants de l'interface, jouant en quelque sorte le rôle de chef d'orchestre. Cette partie du manipulateur de document est construite de la même manière que l'éditeur de livre présenté précédemment.

Néanmoins, certaines classes propres à l'édition syntaxique doivent être définies pour par exemple représenter les différentes constructions d'une description de langage ou pour représenter les nœuds de l'arbre d'un document. Ces nouvelles classes sont décrites dans la suite en leloup. Leur implantation permet d'exhiber quelques techniques classiques en programmation objets.

L'ordre que nous avons choisi pour présenter ces classes permet (nous l'espérons) de comprendre l'implantation de GÉPI tout en présentant de façon progressive les mécanismes objets utilisés. Nous avions pensé dans un premier temps utiliser l'ordre chronologique de définition des classes, mais cet ordre est lui-même difficile à retrouver. En effet, s'il est possible de fixer plus ou moins l'ordre d'apparition des classes, il est bien difficile d'en faire autant pour les méthodes qui leur sont associées. Ces dernières apparaissent souvent par nécessité lors de la définition d'autres classes.

6.2 Désignation d'une zone de texte

Avant de passer à la description de classes plus complexes, commençons par définir celles qui permettent de désigner une position dans le texte ou une zone de texte.

6.2.1 La classe `position`

Dans le texte de l'éditeur pleine page de l'environnement Aida, `medite`, un caractère est désigné à l'aide de ses coordonnées cartésiennes.


```
(classe position
     -superClasses (objet)
     -variables
        ((x 0)
         (y 0)))

(demethod (position ligne) (self)
     (send 'y self))

(demethod (position colonne) (self)
     (send 'x self))
```

Figure 6.2. La classe `position` en leloup.

Pour ne pas travailler directement sur un couple, coordonnée en `x`, coordonnée en `y`, il est préférable de définir la classe `position` dont les instances désignent la position d'un caractère dans le texte. Outre le fait qu'il est possible de manipuler globalement une position dans le texte à l'aide d'un seul objet, cette technique permet de faire abstraction de la représentation d'une position en n'utilisant que les méthodes définies dans cette nouvelle classe.

La classe `position` est ainsi définie avec deux variables, `x`, la colonne où se trouve ce caractère et `y`, la ligne où se trouve ce caractère (cf. Fig. 6.2). Les valeurs par défaut des variables `x` et `y` correspondent aux coordonnées du premier caractère d'un texte, dans le coin supérieur gauche.

Il est désormais possible de définir d'autres méthodes de plus haut niveau pour manipuler les positions du texte correspondant au document. Par exemple, `surLaLigne?`, un prédicat qui est vrai si le receveur est sur la ligne donnée en argument:

```
( demethod (position surLaLigne?) (self ligne)
    (= ( send 'ligne self) ligne))
```

Les méthodes = et < permettent quant à elles de comparer deux positions dans un texte, l'ordre total fixé entre les positions étant l'ordre naturel de lecture d'un texte, de haut en bas, de gauche à droite :

```
( demethod (position =) (positionA positionB)
    (and (= ( send 'ligne positionA)
            ( send 'ligne positionB))
         (= ( send 'colonne positionA)
            ( send 'colonne positionB))))

( demethod (position <) (positionA positionB)
    (let ((ligneA ( send 'ligne positionA))
          (ligneB ( send 'ligne positionB)))
      (or (< ligneA ligneB)
          (and (= ligneA ligneB)
               (< ( send 'x positionA) ( send 'x positionB))))))
```

Il est intéressant de remarquer ici que le fait de passer par l'intermédiaire d'une classe particulière pour manipuler des objets simples comme les positions permet de choisir plus librement le noms des opérations que l'on associe à ces objets. Les sélecteurs = et <, habituellement utilisés pour comparer des nombres peuvent être réutilisés pour manipuler des instances de la classe `position`.

Les méthodes `dx` et `dy` permettent de savoir quel est le décalage en `x`, respectivement `y` entre deux positions :

```
( demethod (position dx) (positionA positionB)
    (- ( send 'colonne positionB)
       ( send 'colonne positionA)))

( demethod (position dy) (positionA positionB)
    (- ( send 'ligne positionB)
       ( send 'ligne positionA)))
```

Comme c'est souvent le cas lorsque l'on développe un logiciel de façon incrémentale, la définition de ses dernière méthodes, dont l'intérêt est moins évident, a été effectué par nécessité lors de la définition d'autres classes. Il en est de même pour d'autres méthodes de la classe `position` que nous ne présenterons pas. Les méthodes `dx` et `dy` sont cependant nécessaires à la bonne compréhension des sections suivantes...

6.2.2 La classe zone

Les instances de la classe `zone`, guère plus complexe que la classe `position`, permettent de désigner une portion de texte. La classe `zone` est construite par composition à l'aide de la classe définie précedemment. Ses deux variables, `p1` et `p2`, désignent respectivement la position du premier caractère de la zone et celle du caractère immédiatement à droite du dernier caractère de la zone (cf. Fig. 6.3).

Comme pour la classe `position`, des méthodes permettant de manipuler les zones peuvent être définies. Par exemple, le prédicat `zoneVide?` qui est vrai si le receveur est une zone ne contenant aucun caractère, c'est à dire lorsque les deux positions `p1` et `p2` du receveur sont identiques :


```
(classe zone
     -superClasses (objet)
     -variables
        ((p1 (creer 'position))
         (p2 (creer 'position))))
```

Figure 6.3. La classe `zone` en leloup.

```
( demethod (zone zoneVide?) (self)
    ( send '=
          ( send 'p1 self)
          ( send 'p2 self)))
```

Définissons maintenant une opération un petit peu plus complexe qui permet de savoir si une position est à l'intérieur d'une zone. Soit la position `P` et la zone désignée par ses deux positions, `Z.P1` et `Z.P2`, l'algorithme suivant réalise cette opération :

```
Si (P < Z.P1)
  alors retourner faux
  sinon Si (P = Z.P1)
         alors retourner vrai
         sinon Si (P < Z.P2)
                alors retourner vrai
                sinon retourner faux
               finSi
        finSi
finSi
```

On peut maintenant écrire la méthode correspondante, `dansLaZone?`, en leloup. Le receveur, `self`, désigne simultanément `Z.P1` et `Z.P2`. Une variable locale, `p1`, permet de mémoriser temporairement la première position associée au receveur :

```
( demethod (zone dansLaZone?) (self P)
    (let ((p1 ( send 'p1 self)))
      (if ( send '< P p1)
          nil
        (if ( send '= P p1)
            self
          (if ( send '< P ( send 'p2 self))
              self
            nil)))))
```

De même, le prédicat `englobante?` qui est vrai si l'argument, une deuxième zone, est englobée dans la zone désignée par le receveur peut être définit ainsi :

```
(classe construction
      -superClasses (objet)
      -variables
         (nom
          extern))
```

Figure 6.4. La classe `construction` en leloup, un *réservoir* à héritage.

```
( demethod (zone englobante?) (self zone)
    (and ( send '<= ( send 'p1 self) ( send 'p1 zone))
         ( send '<= ( send 'p2 zone) ( send 'p2 self))))
```

Même si, dans le cas des classes `zone` et `position`, on ne profite pas de l'héritage, la définition de classes permet l'encapsulation des données et, les objets manipulés sont de plus haut niveau que les objets primitifs du langage LISP.

6.3 Représentation des constructions

Comme dans toutes les applications, le problème crucial est de trouver les classes plutôt que les objets. Un objet isolé est moins intéressant qu'une famille d'objets partageant des propriétés communes. Aucune méthode systématique ne permet de trouver les meilleurs classes. S'il y en avait une, on pourrait dire que l'on a une méthode infaillible pour développer un logiciel. En fait, le talent et l'expérience constitue un facteur important dans la réussite du développement [Meyer 88b].

Souvent, la bonne technique pour trouver les classes consiste simplement à décrire une classe pour chaque catégorie du monde réel. C'est cette technique qui est employée pour représenter les différentes constructions du langage. Ainsi, la classe `constructionAtomique` représente les constructions *atomique*, la classe `constructionGroupe` les constructions *groupe*, la classe `constructionListe` les constructions *liste*, la classe `constructionSucre` les constructions *sucre* et enfin la classe `constructionChoix` les constructions *choix*.

6.3.1 Classe `construction`

Avant de définir chacune de ces classes, la classe `construction` est définie pour regrouper les propriétés communes aux différentes catégories de constructions. Pour cela, la classe `construction` est la superclasse de toutes les classes décrivant les constructions. Elle ne correspond à aucune construction particulière et n'est que le *réservoir* des propriétés communes à l'ensemble des catégories de constructions.

Toutes les constructions possèdent un nom et peuvent comporter une clause *extern*. Aussi, la classe `construction` est-elle définie avec deux variables, `nom` pour contenir le nom de la construction et `extern` pour contenir la chaîne de caractères qui représente la clause *extern* (cf. Fig. 6.4). Ces deux variables seront héritées par les sous-classes `constructionChoix`, `constructionGroupe`, `constructionListe`, `constructionAtomique` et `constructionSucre`.

Cette technique qui consiste à créer dans un premier temps une superclasse générale pour factoriser des propriétés est courante en programmation objets. Généralement, les instances de la classe qui sert de réservoir à héritage ne sont pas utilisables en tant que telle car trop générales. C'est le cas des instances de la classe `construction` elle-même. On parle dans ce cas d'une classe abstraite et, certains langages comme Smalltalk [Goldberg 83] ou Objective-C [Cox 83b] permettent de préciser qu'une classe est abstraite et qu'il n'y a pas lieu de l'instancier. En leloup, cette notion de classe abstraite n'existe pas et toutes les classes sont instanciables.

Afin de vérifier la bonne utilisation des variables `nom` et `extern`, des réflexes peuvent être attachés à ces variables. Pour la variable `nom`, on vérifie qu'il s'agit bien d'un symbole Le-Lisp à l'aide du prédicat prédéfini `symbolp`. La variable `extern` peut soit contenir une chaîne de caractères, `stringp`, soit rien du tout, `nil`, car la clause *extern* est facultative dans tous les cas de construction.

```
( reflexe (construction nom)
    -avantEcriture
       (lambda (self valeurAvant valeurApres)
          (unless (symbolp valeurApres)
             (error 'nom
                    "Mauvais nom de construction."
                    (list self valeurApres)))))

( reflexe (construction extern)
    -avantEcriture
       (lambda (self valeurAvant valeurApres)
          (unless (or (null valeurApres) (string valeurApres))
             (error 'extern
                    "Mauvaise clause extern."
                    (list self valeurApres)))))
```

Ces réflexes de vérification de type des variables `nom` et `extern` sont, comme toutes les propriétés d'une classe, hérités par les sous-classes de `construction`. La vérification de la bonne utilisation des variables `nom` et `extern` sera également effectuée pour les instances des sous-classes de `construction`.

6.3.2 Les constructions *choix*, *groupe* et *sucre*

La représentation des constructions *choix*, *groupe* et *sucre* est immédiate. Les trois classes correspondantes, `constructionChoix`, `constructionGroupe` et `constructionSucre` sont toutes les trois sous-classes de `construction` et possèdent donc par héritage les deux variables `nom` et `extern` (cf. Fig. 6.5).

```
(classe constructionChoix
     -superClasses (construction)
     -variables
        (choix))

(classe constructionGroupe
     -superClasses (construction)
     -variables
        (groupe))

(classe constructionSucre
     -superClasses (construction)
     -variables
        (sucre))
```

Figure 6.5. Les classes `constructionChoix`, `constructionGroupe` et `constructionSucre` en leloup.

La classe `constructionChoix` représente les constructions *choix* et possède une variable propre, `choix`, qui mémorise le contenu de la clause *choix*. Par exemple, pour une construction *choix* donnée, comme `instruction`,

```
...
(instruction
    ( choix affectation  siAlors tantQue))
...
```

l'instanciation suivante est déclenchée lors de la saisie du langage :

```
? ( creer 'constructionChoix
?         'nom 'instruction
?         'choix '(affectation  siAlors tantQue ))
= constructionChoix[instruction () (affectation  siAlors tantQue)]
```

Pour sa part, la classe `constructionGroupe` possède une variable, `groupe`, qui mémorise le contenu de la clause *groupe*. Comme pour une construction *choix*, l'instanciation de la classe `constructionGroupe` est directement calquée sur la représentation externe des constructions *groupe* :

```
? ( creer 'constructionGroupe
?         'nom 'tantQue
?         'groupe '("tantQue" condition (i+) "do" instruction))
= constructionGroupe[tantQue () ("tantQue" condition (i+) "do"
                                           instruction)]
```



```
(classe langage
     -superClasses (objet)
     -variables
        (nom
         terminaux
         terminauxSeparateurs
         abreviations
         bruits
         (actionParDefaut '(b1))
         (indentationParDefaut 3)
         constructions))
```

Figure 6.6. Représentation d'un langage, la classe `langage`.

Sur le même modèle que les deux classes précédentes, la classe `constructionSucre` possède une variable propre, `sucre`, qui mémorise le contenu de la clause *sucre* :

```
? ( creer 'constructionSucre
?         'nom 'unProgramme
?         'extern "joli programme"
?         'sucre '(bloc (cr) "."))
= constructionSucre[unProgramme "joli programme" (bloc (cr) ".")]
```

L'opération de saisie du langage est donc très simple et ne consiste qu'à lire la liste représentative du langage pour instancier les différentes constructions. Chaque construction est représentée à l'aide d'une instance de la classe correspondante selon les mots clés *groupe*, *sucre*, *choix*, *atomique liste*.

Une construction n'est instanciée qu'une seule fois lors de la saisie du langage. Toutes les instances correspondantes sont regroupées dans un objet chargé de représenter le langage dans son ensemble. La classe `langage` (cf. Fig. 6.6) permet de créer de tels objets dont la structure est elle même directement issue de la forme externe d'une description de langage (cf. chap. 2).

La méthode `construction`, définie dans la classe `langage` prend en argument un nom de construction et retourne l'instance associée. Ainsi, si l'on suppose que la variable `miniPascal` référence une instance de la classe `langage`, l'envoi de message suivant permet de retrouver l'instance correspondant à la construction `tantQue` :

```
? ( send 'construction miniPascal 'tantQue)
= constructionGroupe[tantQue () ("tantQue" condition (i+) "do"
                                           instruction)]
```

Définissons maintenant, à titre d'exemple, la méthode `constructionInterne`

```
(classe constructionAtomique
     -superClasses (construction)
     -variables
        (atomique
         (automate '(lambda (texteSource)
                            (debug t)
                            (error 'automate
                                   "Automate indéfini." nil)))))
```

Figure 6.7. La classe `constructionAtomique` en leloup.

de la classe `constructionSucre` qui retourne une construction englobée dans du sucre syntaxique. Le receveur est la construction *sucre* dont on veut obtenir la construction interne. Le deuxième argument est l'objet qui représente le langage tout entier. L'algorithme est très simple et consiste à parcourir le contenu de la clause *sucre* pour, dans un premier temps, retrouver le nom de la construction interne. Puis, dans un deuxième temps, le message `construction` est envoyé au langage qui se charge de retrouver l'instance correspondante :

```
( demethod (constructionSucre constructionInterne) (self langage)
    (let ((nomInterne
              (any (lambda (s)
                           (and (symbolp s) s))
                   ( send 'sucre self))))
        ( send 'construction langage nomInterne)))
```

6.3.3 Les constructions *atomique*

La classe `constructionAtomique` représente les constructions *atomique*. En plus des variables héritées de la classe `construction`, elle comprend deux variables propres, `atomique` et `automate` (cf. Fig. 6.7).

La variable `atomique` sert à mémoriser le contenu de la clause *atomique*, une expression régulière (cf. chap. 2). La variable `automate` est destinée à contenir l'automate de reconnaissance correspondant, une *lambda-expression* LISP.

```
? (setq idf
?       ( creer 'constructionAtomique
?               'nom 'idf
?               'atomique '(lettre
?                           ( 0-n lettreOuChiffre))))
= constructionAtomique[idf () (lettre ( 0-n lettreOuChiffre)) ()]
```

Dans le cas de l'instanciation d'une construction atomique, il ne faut pas oublier

de lancer le calcul du champ `automate` en envoyant le message `calculeAutomate` à la nouvelle instance :

```
( send 'calculeAutomate idf langage)
```

L'argument `langage` permet d'expanser les abréviations contenues dans l'expression régulière de la variable `atomique`. La mise en place d'un réflexe en écriture qui met à jour automatiquement la variable `automate` en cas de modification de la variable `atomique` est délicate dans le cas présent. En effet, il faut connaître le langage auquel la construction appartient pour, à l'intérieur du réflexe, envoyer le message `calculeAutomate` à l'objet dont on modifie la variable `atomique`.

Comme GÉPI manipule simultanément plusieurs langages il est trop dangereux de désigner le langage concerné à l'intérieur du réflexe en utilisant une variable globale par exemple. La solution choisie dans ce cas consiste à chaque écriture de la variable `atomique` à réécrire la variable `automate` avec sa valeur par défaut, une lambda-expression qui déclenche le débogueur :

```
( reflexe (constructionAtomique atomique)
    -avantEcriture
        (lambda (self valeurAvant valeurApres)
            ( send 'automate
                   self
                   '(lambda (texteSource)
                        (debug t)
                        (error 'automate
                               "Automate indéfini." nil)))))
```

Dans ce cas, un réflexe ne permet pas d'assurer proprement l'intégrité de l'objet. Le réflexe est cependant utilisé pour, faute de mieux, déclencher le plus tôt possible la recherche de l'erreur liée à la mauvaise utilisation de l'objet.

6.3.4 Les constructions *liste*

Trois classes représentent les constructions *liste* : `constructionListe`, `constructionListe0-n` et `constructionListe1-n`. La classe `constructionListe` est sous-classe de `construction` tandis que les classes `constructionListe0-n` et `constructionListe1-n` sont des sous-classes de `constructionListe` (cf. Fig. 6.8).

Toutes les variables propres aux constructions *liste* sont regroupées dans la classe `constructionListe`. Elles servent à mémoriser le contenu des différentes clauses qui constituent une construction *liste*. La variable `tete` contient s'il existe le marqueur de tête de liste, de même, `queue` le marqueur de fin de liste, `separateur` le séparateur, `indent` les actions d'indentation et enfin `item` le nom de la construction que l'on peut répéter.

Les deux sous-classes `constructionListe0-n` et `constructionListe1-n` n'ont aucune variable propre et ne se distingueront que dans la suite lorsque nous définirons des méthodes pour ces classes. Par exemple, pour la décompilation ou

```
(classe constructionListe
    -superClasses (construction)
    -variables
       (item
        tete
        separateur
        queue
        indent))
```

```
(classe constructionListe0-n
    -superClasses
        (constructionListe))
```

```
(classe constructionListe1-n
    -superClasses
        (constructionListe))
```

Figure 6.8. Représentation des constructions *liste* en leloup.

l'analyse syntaxique (cf. chap. 7), les constructions *liste0-n* se distinguent des constructions *liste1-n*, ces dernières imposant la présence d'au moins un item.

La classe `constructionListe`, comme la classe `construction`, peut être qualifiée d'abstraite et il n'y a pas lieu de l'instancier. Ce n'est qu'un réservoir à héritage, local cette fois-ci, aux constructions *liste*.

6.4 Représentation des nœuds de l'arbre

Pour la représentation des nœuds de l'arbre, il existe déjà une classe prédéfinie dans l'environnement Aida, la classe `node` qui permet de représenter des arbres N-aires.

6.4.1 Réutilisation de la classe node

Un instance de la classe node permet de représenter un nœud ou une feuille d'un arbre. Cette classe contient toutes les primitives habituelles de manipulation d'arbres, accès au père, accès aux fils, ajout d'un fils, supression d'un fils ...
En outre, cette classe est utilisée pour la représentation des arbres manipulés

par l'éditeur d'arbres, `treeeditor`, lui aussi prédéfini dans l'environnement Aida. L'utilisation de la classe `node` pour représenter les arbres du manipulateur de documents offre donc le double avantage de résoudre les problèmes de la gestion d'arbres tout en permettant l'utilisation de l'éditeur d'arbres `treeeditor`.

Pour utiliser une classe existante, deux techniques sont possibles, la composition ou l'héritage. La composition consiste à définir une classe dont les variables sont des instances d'autres classes. Cette technique a déjà été utilisée pour la classe `zone` dont les variables référencent des instances de la classe `position`. L'utilisation d'une classe par héritage consiste à définir la nouvelle classe comme étant une sous-classe de la classe que l'on veut réutiliser.

La classe `node` ne peut pas servir directement pour la représentation des nœuds de l'arbre d'un document car de nouveaux attributs doivent être associés a ces nœuds comme par exemple la construction qui correspond à un nœud. Si l'on désire utiliser l'éditeur d'arbres `treeeditor`, il faut en outre que les nœuds de l'arbre représentatif du document possèdent *certaines* propriétés de la classe `node`, la classe treeeditor étant conçue pour manipuler de tels arbres. Dans le cas présent, la solution la plus simple consiste à réutiliser la classe `node` par héritage, les propriétes de la classe `node` nécessaires au bon fonctionnement de l'éditeur d'arbres étant héritées.

Les nouveaux attributs, propres aux nœuds d'un arbre représentatif de documents, peuvent être définis à l'aide de nouvelles variables définies dans les sous-classes de la classe `node`. Pour l'éditeur d'arbres, `treeeditor`, rien ne change et celui-ci continue de fonctionner à merveille car les arbres qu'il manipule sont bien du type `node`, une sous-classe héritant des champs et des méthodes de sa superclasse.

La classe `node`, prédéfinie avec l'environnement Aida, n'a pas été définie en leloup, mais avec la couche objets de Le-Lisp [Chailloux 86]. Grâce à la fonction `classeExterne`, il est possible d'intégrer la classe `node` parmi les classes de leloup (cf. § 5.4.1) :

```
( classeExterne '#:tclass:node)
```

La classe `node` peut maintenant être utilisée comme superclasse d'une classe leloup.

6.4.2 Classe noeud

Les différentes catégories de nœuds qui constituent l'arbre représentatif d'un document possèdent des points communs que nous regroupons dans la classe `noeud` (cf. Fig. 6.9). Cette classe est sous-classe de la classe `node`. Afin d'hériter des méthodes de la classe `objet`, prédéfinie en leloup, la deuxième superclasse directe de `noeud` est `objet`. En effet, la classe `node` définie avec la couche objet de Le-Lisp n'hérite pas des méthodes de la classe `objet`.

Les attributs spécifiques aux nœuds de l'arbre représentant un document sont aux nombre de trois : `construction`, `zone` et `optionnel?`. La variable cons-

```
(classe noeud
     -superClasses (node objet)
     -variables
        (construction
         (zone (creer 'zone))
         (optionnel? nil)))
```

Figure 6.9. La classe noeud en leloup.

truction référence l'instance de construction qui est associée au nœud. La zone d'influence associée à un nœud est représentée à l'aide de la variable zone qui référence une instance de la classe zone définie précédemment.

La troisième variable, optionnel?, permet de représenter le fait qu'un nœud est optionel ou non. Cet indicateur peut prendre trois valeurs, nil, visible ou invisible. Lorsque cet indicateur vaut nil, cela signifie que le nœud n'est pas optionnel. Dans ce cas, une opération détruisant le nœud correspondant rend le document syntaxiquement incorrecte. Lorsque la variable optionnel? vaut visible ou invisible, elle indique que le nœud est optionnel et qu'il est ou non visible dans le texte du document. Pour vérifier que la variable optionnel? prend bien sa valeur dans le triplet, nil, visible et invisible, un réflexe s'avère utile :

```
( reflexe (noeud optionnel?)
    -avantEcriture
       (lambda (self valeurAvant valeurApres)
          (unless (member ( send 'optionnel? self)
                          '(nil visible invisible))
             (error 'optionnel? "Mauvaise valeur." valeurApres))))
```

Dans les premières versions du manipulateur de documents, la classe noeud était l'unique classe permettant de représenter les nœuds de l'arbre du document. Progressivement, de nouvelles sous-classes, plus spécifiques, furent définies pour décrire séparément les propriétés des nœuds portant une construction *groupe*, une construction *liste*, une construction *atomique*, du texte entré librement etc.
Du rôle de classe instanciable, la classe nœud est progressivement passée au rôle de classe *abstraite*, c'est-à-dire de classe dont l'instanciation ne présente guère d'intérêt : la classe noeud est devenue pour ses sous-classes l'équivalent de la classe construction pour les sous-classes de constructions, un *réservoir à héritage*.

Le graphe d'héritage actuel du projet GÉPI est représenté sur la figure 6.10 . La classe noeud est maintenant la superclasse des classes noeudAtomique, noeudGroupe, noeudListe, noeudSucre, noeudChoix et enfin noeudTexte. Pour la réalisation des commandes du manipulateur de document, cette technique facilite considérablement la tâche. Par exemple, l'opération de décompilation est spécifique à chaque catégorie de nœuds et nous verrons (cf. § 7.2) comment nous avons réparti l'algorithme de décompilation dans les différentes classes associées à ces catégories de nœuds.

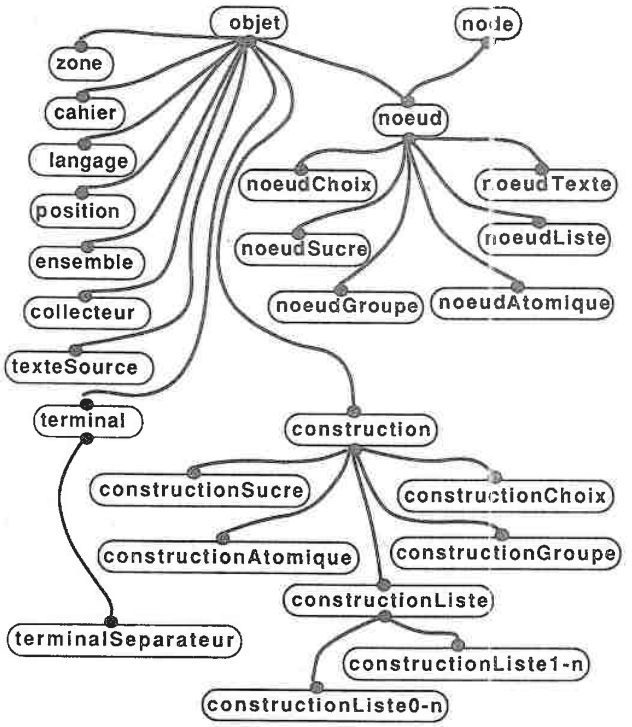


Figure 6.10. Le graphe d'héritage des classes du projet GÉPI.

6.4.3 Les nœuds *Choix*, *liste*, *groupe* et *texte*

Les instances des classes noeudChoix, noeudListe, noeudGroupe et noeudTexte (cf. Fig. 6.11) permettent de représenter les nœuds portant respectivement, une construction *choix*, une construction *liste*, une construction *groupe* ou encore un nœud portant du texte frappé librement par l'utilisateur.

Le champs hérité, construction, contient, dans le cas des noeudChoix, noeudListe et noeudGroupe, la construction correspondante.

```
(classe noeudChoix
     -superClasses (noeud))

(classe noeudGroupe
     -superClasses (noeud))

(classe noeudTexte
     -superClasses (noeud))

(classe noeudListe
     -superClasses (noeud))
```

Figure 6.11. Les classes `noeudChoix`, `noeudGroupe` `noeudTexte` et `noeudListe` en leloup.

Dans le cas des nœuds *texte*, la variable `construction` désigne la construction qui a précédé à la création du nœud *texte* correspondant. Lorsque l'utilisateur frappe du texte libre dans la zone d'influence d'un nœud *groupe* par exemple, ce nœud est transformé en un nœud *texte* (cf. § 1.2). Le nœud *texte* porte alors la construction du nœud qu'il remplace. Cette construction sert de point de départ en cas d'analyse syntaxique du nœud *texte* (cf. § 7.4).

Si ces classes ne possèdent pas de variables propres, elles se distinguent par les méthodes qu'elles comportent. Par exemple, l'implantation de la commande *couper*, *copier*, *coller* est réalisée à l'aide de méthodes de même sélecteur dont la définition est spécifique à chaque classe. L'intérêt d'avoir des classes différentes permet de limiter le nombre de tests à écrire pour la réalisation de ces opérations. Par exemple, lorsqu'on écrit la méthode `coller` dans la classe `noeudTexte`, il est inutile de tester s'il s'agit effectivement d'un nœud portant du texte. Bien entendu, si une opération peut être réalisée de façon identique pour ces quatre classes, celle-ci peut être définie au niveau de la classe `noeud`.

6.4.4 Classe `noeudAtomique`

Les nœuds qui portent des constructions *Atomique* sont représentés à l'aide de la classe `noeudAtomique` (cf. Fig. 6.12).

La variable héritée, `construction`, référence l'instance de `constructionAtomique` associée au nœud *atomique*. Si le nœud *atomique* est expansé et est lexicographiquement correct, la variable `listeDeCaracteres` contient la liste de caractères correspondante. Lorsqu'un nœud *atomique* n'est pas expansé, la variable `listeDeCaracteres` contient la liste vide LISP, `nil`.


```
(classe noeudAtomique
     -superClasses (noeud)
     -variables
        (listeDeCaracteres nil))
```

Figure 6.12. La classe `noeudAtomique` en leloup.

```
(classe noeudSucre
     -superClasses (noeud)
     -variables
        ((zoneInterne (creer 'zone))
         constructionsInvisibles))
```

Figure 6.13. La classe `noeudSucre` en leloup.

6.4.5 Classe `noeudSucre`

La classe `noeudSucre` représente les nœuds qui portent une construction *sucre* (cf. Fig. 6.13). La variable `construction` héritée référence la construction *sucre* associée au nœud.

Une construction sucre permet d'élider des ramifications unaires dans l'arbre représentatif du document (cf. § 2.3.5). Comme pour toutes les classes de nœuds précédentes, la variable héritée, `zone`, référence la zone d'influence associé au nœud. La variable `zoneInterne`, propre à la classe `noeudSucre` référence une zone englobée dans celle référencée par la variable `zone`. Cette zone interne correspond à la zone d'influence de la construction interne.

Par exemple, soit la construction *sucre* suivante :

```
(unProgramme
    ( sucre bloc (cr) "."))
```

Le nœud représentatif d'une telle construction est instance de la classe `noeudSucre` et, les eventuels fils associés à la construction `bloc` sont portés directement par ce nœud *sucre*. Dans ce cas, la variable `zone` du nœud *sucre* désigne la zone d'influence associée à la construction `unProgramme` dans son ensemble. La place occupée dans le texte par l'effet de l'action de paragraphage, `(cr)`, et par le caractère point, `"."`, est comprise dans cette zone. La variable `zoneInterne` référence une sous-zone qui correspond à la place occupée dans le texte par la construction `bloc` seule.

Ainsi, les instances de `noeudSucre` vérifient toujours la condition suivante :

```
? ( send englobante?
?             ( send zone unNoeudSucre)
?             ( send zoneInterne unNoeudSucre))
= t
```

La variable **constructionsInvisibles** contient la liste des constructions pour lesquelles une ramification de l'arbre a été élidée. Lorsque la construction interne d'une construction *sucre* ne peut dériver sur une autre construction *sucre*, le contenu de cette liste est limité à une seule construction. C'est le cas par exemple de la construction **unProgramme** qui dérive sur la construction *groupe* **bloc**. Cependant, lorsque la construction interne d'une construction *sucre* dérive sur une autre construction *sucre*, plusieurs ramifications peuvent être élidées. La variable **constructionsInvisibles** s'allonge alors des constructions correspondantes.

6.5 Propagation d'une modification

Selon la commande déclenchée par l'utilisateur, la zone d'influence d'un nœud ou d'une feuille de l'arbre peut changer. Par exemple, lorsque l'on frappe un caractère dans la zone d'influence d'un nœud *texte*, la zone associée à ce nœud s'agrandit. Inversement, si l'on efface un caractère, cette zone diminue. Ou encore, lorsque l'on développe un nœud *groupe*, la zone du nœud développé s'agrandit ou se réduit selon la place occupée par le nouveau sous-arbre.

Lorsque l'on modifie la zone d'influence d'un nœud ou d'une feuille de l'arbre, il est nécessaire de mettre à jour les zones d'influence de certains nœuds. Le problème consiste donc à répercuter dans le reste de l'arbre une modification de zone effectuée sur un des nœuds ou sur une des feuilles.

Cette section décrit l'algorithme utilisé pour faire cette mise à jour. Son implantation en leloup utilise une propriété importante des langages à objets : la liaison dynamique. En outre, l'implantation contient des envois de message où le sélecteur est lui-même calculé dynamiquement.

6.5.1 Principes de l'algorithme de propagation

L'algorithme de propagation est basé sur les propriétés des zones d'influence associées aux nœuds[1] de l'arbre (page 134). La première question que l'on peut se poser est de savoir quels sont les nœuds concernés par la modification d'une zone d'influence d'un nœud particulier. La réponse est évidente une fois que l'on a constaté que la représentation des zones d'influence des différents nœuds est une structure correctement parenthésée. Cette structure peut être représentée à l'aide de crochets (cf. Fig. 6.1, page 135), un crochet ouvrant correspondant au début d'une zone et un crochet fermant à la fin de la zone. Sur une telle représentation, il est clair que la modification d'une zone concerne tous les crochets ouvrants ou fermants rencontrés à partir de cet endroit jusqu'à la fin du texte.

1. Quand nous parlons des nœuds, il s'agit indifféremment des nœuds et des feuilles.

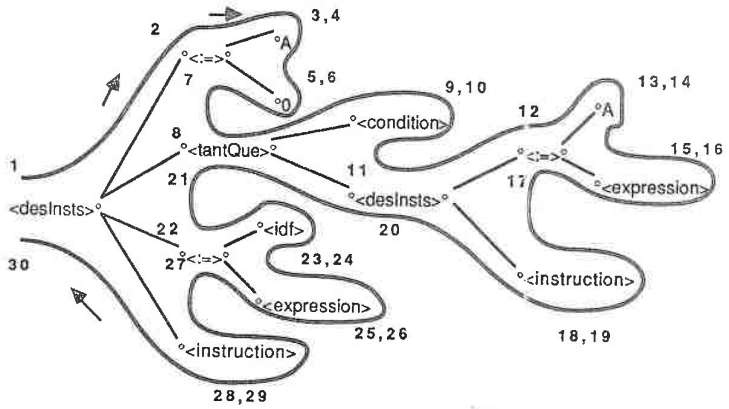


Figure 6.14. L'ordre de parcours des nœuds pour l'algorithme de propagation des modifications. Le parcours de l'arbre est numéroté dans l'ordre de visite des nœuds. Attention, comme pour le manipulateur de documents, cet arbre est représenté couché, sa racine est à gauche et les fils sont imprimés dans l'ordre de haut en bas.

Plus intuitivement, lorsque vous entrez un caractère retour chariot dans le texte d'un éditeur pleine page, toute la suite du texte *se décale*.

Dans l'arbre, le parcours permettant de rencontrer tout les crochets à partir de l'endroit de modification est un parcours en profondeur d'abord chaque nœud étant visité deux fois. La première visite d'un nœud a lieu avant le parcours de ses fils et, la deuxième a lieu une fois le parcours effectué sur tous ses fils. La figure 6.14 schématise ce parcours sur une image agrandie de l'arbre de la figure 6.1.

La première visite d'un nœud provoque la mise à jour de la position du début de sa zone d'influence (le crochet ouvrant). La position de fin de la zone d'influence (le crochet fermant) est mise à jour lors de la deuxième visite. Ainsi, l'ordre de mise à jour des positions correspond à l'ordre dans lequel on les rencontre en lisant le texte du document.

L'algorithme consiste donc à écrire un tel parcours, ce parcours ne commençant pas forcément sur la première position de la racine, mais à partir de l'endroit où a eu lieu la modification. Une première version, non raffinée de l'algorithme peut maintenant être écrite :

```
Initialiation avec le premier point de visite.

tantQu'il reste des points à visiter :
    faire
        (1) Décalage du point de visite courant.
        (2) Passage au point de visite suivant.
    fait
```

6.5.2 Décalage d'une zone

Intéressons nous maintenant à l'opération principale de l'algorithme, celle qui est au cœur de la boucle, (1), le décalage d'une position associée à un nœud. L'opération réalisant ce décalage doit disposer de cinq informations, le nœud concerné par le décalage, la position qu'il faut décaler, la ligne sur laquelle a eu lieu la modification ainsi que la valeur du décalage en x et en y.

Une première méthode, **decale**, peut être définie dans la classe **noeud**. En plus du receveur qui désigne le nœud concerné par le décalage, les arguments supplémentaires sont donc au nombre de quatre :

- **p1Ou2** permet d'indiquer s'il faut modifier la première ou la deuxième position selon que sa valeur est le sélecteur p1 ou le sélecteur p2,
- **ligneDeDepart**, le numéro de la ligne où a eu lieu la modification,
- **dx**, le décalage en x provoqué par la modification,
- et dy, le décalage en y provoqué par la modification.

Le corps de la méthode **decale** de la classe **noeud** ne fait que transmettre le même message à la zone d'influence du nœud receveur :

```
( demethod (noeud decale) (self p1Ou2 ligneDeDepart dx dy)
    ( send 'decale ( send 'zone self) p1Ou2 ligneDeDepart dx dy))
```

Pour sa part, dans la classe zone, la méthode decale envoie un message de même sélecteur à la position de début ou à la position de fin selon que la valeur de **p1Ou2** est p1 ou p2. Le sélecteur étant aussi un argument calculé de la fonction **send** (cf. § 5.1.2), il n'est pas nécessaire de tester la valeur de la variable **p1Ou2** :

```
( demethod (zone decale) (self p1Ou2 ligneDeDepart dx dy)
    ( send 'decale ( send p1Ou2 self) ligneDeDepart dx dy))
```

Enfin, dans la classe **position**, la méthode **decale** et ses deux méthodes de *service*, **decaleDxDy** et **decaleDx** peuvent être définies. La méthode **decale** teste si la position désignée par le receveur est ou non sur la ligne où la modification a eu lieu. Selon le cas, la position est décalée en x et en y avec la méthode **decaleDxDy** ou simplement en y avec la méthode **decaleDy** :


```
( demethod (position decale) (self ligneDeDepart dx dy)
    (if ( send 'surLaLigne? self ligneDeDepart)
        ( send 'decaleDxDy self dx dy)
      ( send 'decaleDy self dy)))

( demethod (position decaleDxDy) (self dx dy)
    ( send 'x self (+ dx ( send 'x self)))
    ( send 'y self (+ dy ( send 'y self))))

( demethod (position decaleDy) (self dy)
    ( send 'y self (+ dy ( send 'y self))))
```

6.5.3 Implantation et optimisation du parcours

L'implantation du parcours des points de visite fait appel à un algorithme de parcours d'arbres classique [Remy 81]. La méthode qui le réalise, **propage**, est définie dans la classe **noeud**. Le receveur est le nœud où il faut commencer le parcours, **self**. Les autres arguments, **p1Ou2**, **ligneDeDepart**, **dx** et **dy** ont la même signification que ceux des méthodes **decale** définies précédemment. La boucle **while** principale ne s'arrête que lorsqu'il n'y a plus de nœuds à traiter, c'est à dire lorsque **self** prend la valeur **nil**. Le corps de la boucle contient les deux traitements principaux, le décalage de la position courante (1), puis le calcul de la position suivante (2). Le décalage de la position courante est effectué à l'aide de la méthode **decale** détaillée précédemment.

Le calcul de la position suivante est effectué à l'aide de la variable **p1Ou2** qui donne la direction à suivre dans le parcours d'arbre. Les méthode de manipulation d'arbres, **fils** et **freres** retournent respectivement si elles existent la liste des fils et la liste des frères du receveur. La méthode **pere** retourne s'il existe le père du nœud receveur :

```
( demethod (noeud propage) (self p1Ou2 ligneDeDepart dx dy)
    (while self
      ;; (1) Décalage du point de visite courant.
      ( send 'decale self p1Ou2 ligneDeDepart dx dy)
      ;; (2) Passage au point de visite suivant.
      (if (eq 'p1 p1Ou2)
          (let ((fils ( send 'fils self)))
            (if fils
                (progn
                  (setq self (car fils))
                  (setq p1Ou2 'p1))
                (setq p1Ou2 'p2)))
          (let ((freres ( send 'freres self)))
            (if freres
                (progn
                  (setq self (car freres))
                  (setq p1Ou2 'p1))
                (setq self ( send 'pere self))
                (setq p1Ou2 'p2))))))
```

Comme un parcours d'arbre en profondeur d'abord, la complexité de l'algorithme précédent croît linéairement avec le nombre de nœuds de l'arbre à parcourir. On peut cependant limiter la longueur du parcours et réduire la complexité de l'opération de mise à jour d'une position après avoir fait les constatations suivantes :

- la propagation du décalage en x ne concerne que les positions situées sur la ligne où a eu lieu la modification,
- lorsque le décalage en y est nul, la propagation du décalage ne concerne que les positions situées sur la ligne où a eu lieu la modification.

Par conséquent, il est inutile de propager le décalage en x dès que la position atteinte sort de la ligne où a eu lieu la modification. En outre, après la ligne sur laquelle a eu lieu la modification, on peut arrêter le parcours si le décalage en y est nul.

L'algorithme de propagation initial peut ainsi être décomposé en deux phases. La première réalise le début du parcours et ne concerne que les positions qui sont sur la ligne où la modification a eu lieu. La deuxième phase du parcours ne concerne que les positions qui sont situées après cette ligne.

La nouvelle version de la méthode propage de la classe noeud est maintenant chargée de la première phase du parcours. A peu de choses près, sa structure est la même que précédemment. En particulier, les actions (1) et (2) sont identiques. Seule la boucle principale est modifiée pour pouvoir abandonner cette première phase du parcours lorsque la fin de la ligne est atteinte. La boucle `while` est ainsi remplacée par une *boucle à échappement*, `untilexit`, dont l'exécution s'arrête dès que l'instruction `exit` est exécutée. Comme précédemment, la position courante est traitée par l'envoi du message decale au nœud courant. Cependant, avant de

traiter cette position, on s'assure qu'elle est sur la ligne où a eu lieu la modification. Si ce n'est pas le cas, la méthode chargée de la suite du parcours est invoquée :

```
( demethod (noeud propage) (self p1Ou2 ligneDeDepart dx dy)
    (untilexit propage
      (unless ( send 'surLaLigne? self p1Ou2 ligneDeDepart)
          ;; Arrêt de la propagation en x.
          ( send 'propageDy self p1Ou2 dy)
          (exit propage))
      ;; (1) Décalage du point de visite courant.
      ( send 'decale self p1Ou2 ligneDeDepart dx dy)
      ;; (2) Passage au point de visite suivant.
      (if (eq 'p1 p1Ou2)
          ...
          ...)))
```

La méthode propageDy est chargée de la deuxième phase du parcours et continue le travail commencé par la méthode propage. Cette fois, pour traiter le nœud courant, la méthode decaleDy qui n'effectue que le décalage en y est déclenchée. Cette méthode est définie sur le modèle de la méthode decale de la classe noeud :

```
( demethod (noeud decaleDy) (self p1Ou2 dy)
    ( send 'decaleDy ( send 'zone self) p1Ou2 dy))
```

La méthode propageDy de la classe noeud est construite sur le modèle de la première version de la méthode propage. Le calcul du point de visite suivant (2) est identique. Cette fois, le décalage de la position courante, (1 bis), utilise la méthode decaleDy. Un test sur la valeur de dy permet d'arrêter le traitement si le décalage en y est nul :

```
( demethod (noeud propageDy) (self p1Ou2 dy)
    (when (<> 0 dy)
      (while self
        ;; (1 bis) Décalage du point de visite courant.
        ( send 'decaleDy self p1Ou2 dy)
        ;; (2) Passage au point de visite suivant.
        (if (eq 'p1 p1Ou2)
            ...
            ...))))
```

Cette optimisation est très efficace lorsque le décalage en y est nul. En fait, avec les manipulations textuelles du document, ce cas est très fréquent : la frappe d'un caractère autre que le retour chariot ne provoque qu'un décalage en x.

6.5.4 Traitement des nœuds *sucre*

L'algorithme de propagation décrit précédemment est implanté à l'aide de la méthode propage définie dans la classe noeud. S'il permet de prendre en compte

le cas des nœuds *texte*, *groupe*, *liste atomique* et *choix*, il apparaît inadapté au traitement des nœuds *sucre*. Ces nœuds possèdent en effet une zone spécifique, **zoneInterne** (cf. § 6.4.5), qu'il convient de mettre à jour au même titre que la zone héritée de la classe **noeud**.

Il est clair que le parcours d'arbre implanté par la méthode **propage** de la classe **noeud** ne doit pas être remis en cause, le même parcours devant être utilisé qu'il s'agisse d'un nœud *sucre* ou non. Cependant, l'action de décalage, lorsqu'elle est appliquée sur un nœud *sucre*, doit faire l'objet d'un traitement particulier. Comme cette action est déclenchée par l'envoi du message **decale** ou **decaleDy** à chaque nœud du parcours, il suffit de redéfinir ces seules méthodes pour réutiliser sans aucune modification l'algorithme implanté par la méthode **propage** de la classe **noeud**.

Les nouvelles méthodes, **decale** et **decaleDy** sont ainsi redéfinies dans la classe **noeudSucre**. Elles sont écrites sur le modèle des méthodes de même nom définies dans la classe **noeud** en prenant en compte la mise à jour de la zone interne du nœud *sucre* :

```
( demethod (noeudSucre decale) (self p1Ou2 ligneDeDepart dx dy)
    ( send 'decale ( send 'zone self) p1Ou2 ligneDeDepart dx dy)
    ( send 'decale ( send 'zoneInterne self) p1Ou2 ligneDeDepart dx dy))

( demethod (noeudSucre decaleDy) (self p1Ou2 dy)
    ( send 'decaleDy ( send 'zone self) p1Ou2 dy)
    ( send 'decaleDy ( send 'zoneInterne self) p1Ou2 dy))
```

Grâce au mécanisme de liaison dynamique et d'héritage, les problèmes algorithmiques liés au parcours de l'arbre sont localisés à l'intérieur d'une unique classe, la classe **noeud**. Le traitement spécifique des instances de la classe **noeudSucre** est, dans ce cas, traité par un masquage des deux méthodes qui modifient le contenu des nœuds *sucre*, **decale** et **decaleDy**.

6.6 Bilan

La réalisation du couplage entre la représentation arborescente et le texte du document est au cœur de l'implantation du manipulateur de documents. La description de ce couplage nous a donné l'occasion de présenter les principales classes de GÉPI, ainsi que l'utilisation de quelques mécanismes courants en programmation objet : réutilisation par composition (classe **zone**), réutilisation par héritage (classe **noeud**) et enfin, masquage de méthodes (traitement des nœuds *sucre*).

Rappelons que l'arbre des classes actuel (cf. Fig. 6.10, page 149) n'a pas été conçu directement sous cette forme, toutes ces classes n'existant pas dans les premières versions de GÉPI. Ce n'est qu'au fur et à mesure des besoins que la création de certaines classes s'est imposée. Par exemple, la première version de GÉPI ne disposait que d'une seule classe de nœud. La complexité d'écriture d'une telle classe devant prendre en compte les propriétés relatives à différentes catégories de nœuds nous a poussé progressivement à la définition des 6 sous-classe actuelles.

Une correspondance directe peut souvent être faite entre les objets du monde réel et les classes associées. Pour les classes représentant les différentes constructions du langage, ce fait est particulièrement évident. Chaque sous-classe de **construction** est une représentation quasi directe de l'aspect externe d'une catégorie de constructions. Ainsi, la représentation interne des objets manipulés par les méthodes est directement calquée sur leur représentation dans le monde réel. Pour écrire et tester les algorithmes, ce point est très important et est souvent une caractéristique de la programmation objets : le programmeur travaille directement sur des objets du monde réel. Chaque constatation qui est faite sur les objets du monde réel peut être directement transcrite dans les programmes.

L'absence de typage permet d'implanter un algorithme en ne faisant qu'un minimum de suppositions sur la nature des objets manipulés. Par exemple, l'unique supposition de l'algorithme de propagation est que chaque nœud rencontré lors du parcours soit capable de répondre aux messages **decale** et **decaleDy**. Ce faisant, la partie essentiellement algorithmique, le parcours en tant que tel, peut être isolé du traitement à effectuer sur chacun des nœuds rencontrés. La partie traitement peut alors prendre des visages différents selon la nature du nœud qui reçoit le message. Cette forme de polymorphisme permet d'écrire des algorithmes moins figés, plus réutilisables. Historiquement, la catégorie construction *sucre* au niveau de la description de langage n'est apparue que dans les dernières versions du projet. L'algorithme de propagation déjà en place pour les autres catégories de nœuds fut réutilisé tel quel.

Les deux principaux composants du manipulateur de documents, l'éditeur de textes pleine page, **edit** et l'éditeur d'arbres **treeeditor** sont également réutilisés sans modifications.

L'absence de typage statique alliée à un typage dynamique fort constitue selon nous toute la puissance des langages à objets. Cette souplesse a cependant un inconvénient indéniable lié au fait qu'aucune vérification statique n'est effectuable. Se convaincre de la correction d'un programme passe obligatoirement par l'essai de ce programme.

7
Décompilation et analyse syntaxique

" Le moine Keiho raconte que le Seigneur Aki avait dit un jour que la vertu martiale était le fanatisme. J'ai constaté que cela s'accordait avec ma propre résolution et dès lors je suis devenu de plus en plus extrême dans mon fanatisme. [page 60] "

Jocho Yamamoto [Yamamoto 84]

Le décompilateur et l'analyseur syntaxique sont deux composants essentiels du manipulateur de documents. Ce chapitre passe au peigne fin l'implantation du décompilateur et, encore une fois, les mécanismes d'héritage et de liaison dynamique sont mis en avant.

La technique utilisée pour l'implantation du décompilateur, qui permet de répartir la difficulté algorithmique dans plusieurs classes, est aussi utilisée pour l'implantation de l'analyseur syntaxique. Le principe utilisé pour ce dernier, nous permet de remettre à la mode une antique technique d'analyse syntaxique.

Plan

La description du décompilateur occupe les trois premières sections de ce chapitre en commençant par la méthode principale du décompilateur et de ses outils (§7.1), puis, différents cas de décompilation sont traités séparément (§7.2) et enfin, l'implantation du déclenchement des actions de paragraphage (§7.3). Les deux sections suivantes sont consacrées respectivement, à l'analyseur syntaxique (§7.4) et à l'habituel bilan de fin de chapitre (§7.5).

7.1 Principes et outils de décompilation

Pour un éditeur syntaxique, le décompilateur est un composant essentiel car il permet de calculer la représentation textuelle d'un arbre syntaxique.

Le décompilateur de GÉPI est dit *incrémental* : il est capable de ne décompiler qu'une partie de l'arbre en ne modifiant que la partie de texte correspondante. Ainsi, lorsque l'on développe une construction `<instruction>` par exemple, seul le développement choisi pour cette construction est décompilé (page 12). La représentation textuelle est localement modifiée en remplaçant le non-terminal `<instruction>` par le résultat de la décompilation du sous-arbre choisit.

D'un point de vue global, l'algorithme de décompilation d'un arbre est un parcours d'arbre en profondeur d'abord. Tout au long de ce parcours, les nœuds et les feuilles de l'arbre qui sont visités produisent un flux de caractères qui, petit à petit, construit la représentation textuelle de l'arbre décompilé.

Le cas de la décompilation incrémentale ne diffère pas fondamentalement du cas de la décompilation complète d'un arbre. Quelques problèmes supplémentaires doivent néanmoins être résolus :

1. la décompilation peut démarrer à partir de n'importe quel nœud,
2. il faut être capable de remplacer la zone de texte associée au sous-arbre modifié,
3. et, les actions de paragraphage doivent être effectuées en fonction d'une nouvelle marge gauche de départ qui n'est pas nécessairement la colonne 0 du texte.

Dans le cas d'une décompilation complète de l'arbre, le nœud de départ est la racine de l'arbre à décompiler. Le problème (1) est simplement résolu en paramétrant le décompilateur avec le nœud de départ du sous-arbre à décompiler.

Les deux autres problèmes, (2) et (3) sont facilement résolus grâce au couplage permanent qui existe entre l'arbre et le texte du document. En effet, avant de commencer la décompilation, on connaît exactement la zone de texte associée au nœud que l'on veut remplacer. La colonne de départ de la décompilation est la colonne de la première position associée à cette zone. Une fois la décompilation effectuée, cette zone est remplacée par le flux de caractères issu de la décompilation du sous-arbre.

Avant de présenter la méthode principale de décompilation, la définition de quelques outils primitifs est nécessaire.

7.1.1 Le collecteur

Le premier outil que nous présentons est le collecteur qui est chargé de collecter au fur et à mesure un flux de caractères. Ce dernier est également chargé de calculer progressivement la position atteinte dans le texte et de permettre la mise

en œuvre du paragraphage. Cet outil est décrit par la classe `ccllecteur` dont nous ne présenterons que l'interface, c'est-à-dire la façon d'utiliser les instances de cette classe. La description de l'implantation de cette classe, par ailleurs fort simple, serait fastidieuse et n'apporterait rien à notre propos.

L'instanciation de la classe collecteur, nécessite l'initialisation de deux variables `position` et `langage`. Pour que le mystère reste entier en ce qui concerne l'implantation de cette classe, la méthode d'impression de la classe `objet`, `prin`, est masquée par une méthode de même nom dans la classe `collecteur`. Cette méthode rend invariablement la chaîne `unCollecteur` :

```
? (setq unCollecteur
?       ( creer 'collecteur
?               'position ( creer 'position)
?               'langage miniPascal))
= unCollecteur
```

La variable `position` est une instance de la classe `position` (cf. § 6.2.1). Elle initialise le collecteur avec la position de départ dans le texte du document où l'on veut capter le flux de caractères. Dans le cas d'une décompilation, cette position correspond à la position de début de la zone du nœud qui est décompilé. La variable `langage` est initialisée avec l'instance du langage utilisé pour réaliser la décompilation. Dans l'exemple précédent, nous supposons que le symbole `miniPascal` référence une instance de la classe `langage`.

L'instance de la classe collecteur que nous venons de créer est dès maintenant capable de collecter ses premiers caractères. Pour l'instant, la position associée à ce collecteur est la position de départ, fixée à l'instanciation :

```
? ( send 'ligne ( send 'position unCollecteur))
= 0
? ( send 'colonne ( send 'position unCollecteur))
= 0
```

Le message `ajoute` envoyé à une instance de la classe `collecteur` permet d'ajouter un ou plusieurs caractères dans le collecteur. La position associée au collecteur est mise à jour par l'instance elle-même :

```
? ( send 'ajoute unCollecteur "A")
= unCollecteur
? ( send 'ligne ( send 'position unCollecteur))
= 0
? ( send 'colonne ( send 'position unCollecteur))
= 1
? ( send 'ajoute unCollecteur "BC")
= unCollecteur
? ( send 'ligne ( send 'position unCollecteur))
= 0
? ( send 'colonne ( send 'position unCollecteur))
= 3
```

Lorsqu'un caractère retour chariot, `#\cr`, apparait dans le flux de caractères collecté, l'instance de la classe `collecteur` déplace elle-même la position courante qui lui est associée au début de la ligne suivante :

```
;; Attention, on ajoute un retour chariot
? ( send 'ajoute unCollecteur #\cr)
= unCollecteur
? ( send 'ligne ( send 'position unCollecteur))
= 1
? ( send 'colonne ( send 'position unCollecteur))
= 0
? ( send 'ajoute unCollecteur "DEF")
= unCollecteur
? ( send 'ligne ( send 'position unCollecteur))
= 1
? ( send 'colonne ( send 'position unCollecteur))
= 3
```

Le collecteur est bien entendu capable de restituer l'ensemble du texte collecté depuis sa création. Le message `toutLeTexte`, envoyé au collecteur permet de retrouver l'intégralité des caractères collectés sous la forme d'une liste de chaînes de caractères :

```
? ( send 'toutLeTexte unCollecteur)
= ("ABC" "DEF")
```

Le collecteur sait aussi effectuer les actions de paragraphage. Pour cela, le message `action` doit comporter l'action à effectuer ainsi que la position d'origine associée à l'action (cf. § 2.5). Par exemple, l'action `(b 10)` ajoute 10 caractères blancs :

```
? ( send 'action unCollecteur '(b 10) unePosition)
= unCollecteur
? ( send 'toutLeTexte unCollecteur)
= ("ABC" "DEF          ")
```

L'action `(cr)` ajoute un caractère retour chariot :

```
? ( send 'action unCollecteur '(cr) unePosition)
= unCollecteur
? ( send 'toutLeTexte unCollect>eur)
= ("ABC" "DEF          " "")
```

Les actions de paragraphage par défaut peuvent également être réalisées par le collecteur qui connaît le langage associé à la décompilation. Pour effectuer l'action de paragraphage par défaut, il suffit d'envoyer le message `actionParDefaut` à l'instance de la classe collecteur. L'action par défaut pour le langage miniPascal (cf. § 2.4) est `(b1)`, action qui consiste à ajouter un caractère espace :

```
? ( send 'actionParDefaut unCollecteur unePosition)
= unCollecteur
? ( send 'toutLeTexte unCollecteur)
= ("ABC" "DEF          " " ")
```

Ce premier outil, le collecteur, permet de prendre en charge une partie du travail de décompilation : capter progressivement le flux de caractères et effectuer les actions de paragraphage. Dans la suite nous ne l'utilisons qu'au travers de son interface, c'est-à-dire à l'aide des méthodes qui lui sont associées.

7.1.2 Affichage des feuilles de l'arbre

Pendant la décompilation, lorsqu'une feuille de l'arbre est visitée, il faut être capable d'afficher sa représentation textuelle. Pour ce faire, toutes les instances des différentes classes de nœuds sont capables de répondre au message `affichage`[1].

Par exemple, la chaîne d'affichage d'un nœud contenant la construction non développée `instruction` est construite en mettant le nom de la construction entre *chevrons* pour visualiser le fait que la construction n'est pas encore développée :

```
?    ( send 'affichage
?          ( creer 'noeudChoix
?                'construction ( send 'construction
?                                     miniPascal
?                                     'instruction)))
= <instruction>
```

Lorsqu'un nœud est optionnel, la chaîne d'affichage du nœud est mise entre crochets :

```
?    ( send 'affichage
?          ( creer 'noeudChoix
?                'construction ( send 'construction
?                                     miniPascal
?                                     'instruction)
?                'optionnel? 'visible))
= [<instruction>]
```

Si d'aventure le nœud qui reçoit le message `affichage` est un nœud invisible, la chaîne retournée est la chaîne vide, qui est invisible :

```
?    ( send 'affichage
?          ( creer 'noeudChoix
?                'construction ( send 'construction
?                                     miniPascal
?                                     'instruction)
?                'optionnel? 'invisible))
=
```

7.1.3 La méthode principale de décompilation

La classe GÉPI est la classe qui représente les environnements de programmation. Une instance de cette classe est un environnement de programmation avec son langage, son éditeur de texte et son éditeur d'arbre (cf. § 1.2). C'est dans cette classe que la méthode principale de décompilation est définie (cf. Fig. 7.1).

1. Il n'y a pas de classe particulière pour représenter les feuilles de l'arbre : une feuille est tout simplement un nœud sans fils. Une méthode `affichage`, définie dans la classe nœud elle-même, est capable de traiter l'affichage de la majeure partie des différentes classes de nœuds. Le cas particulier des nœuds *atomique*, par exemple, est traité en masquant la méthode `affichage` dans la classe `noeudAtomique`.

```
(demethod (gepi decompile) (self noeudAremplacer nouveauNoeud)
    ;; (1) Initialisations et instanciation du collecteur :
        ...
    ;; (2) Lancement de la décompilation :
        (send 'decompile nouveauNoeud collecteur position self)
    ;; (3) Modification du texte de l'éditeur pleine page :
        ...
    ;; (4) Modification de l'arbre :
        ...
    ;; (5) Propagation des modifications de zone
    ;;     dans le reste de l'arbre :
        ...)
```

Figure 7.1. L'algorithme général de décompilation de GÉPI, la méthode **decompile** de la classe **gepi**.

Cette méthode est chargée de lancer la décompilation. Elle permet de remplacer un nœud de l'arbre, **noeudAremplacer**, par un autre nœud, **nouveauNoeud**. L'arbre ainsi que le texte associé sont mis à jour. Dans l'arbre, **nouveauNoeud** prend la place de **noeudAremplacer**. Dans le texte, la zone associée à **noeudAremplacer** est supprimée pour laisser la place à la partie de texte issue du résultat de la décompilation de **nouveauNoeud**.

La méthode **decompile** de la classe **gepi** comporte cinq étapes distinctes, exécutées en séquence (cf. Fig. 7.1). La première étape, (1), consiste à effectuer les préparatifs de la décompilation, en particulier, l'instanciation du collecteur qui servira pour la décompilation du nouveau nœud. L'étape suivante, (2), est l'étape qui consiste à lancer la décompilation sur le nouveau nœud. Cette opération est réalisée par l'envoi du message **decompile** à l'instance référencée par la variable **nouveauNoeud**. La troisième étape, (3), met à jour le texte de l'éditeur pleine page : les caractères mémorisés dans le collecteur lors de la deuxième étape prennent leur place dans le texte. Ensuite (4) l'arbre associé au document est lui aussi mis à jour, **nouveauNoeud** remplace **noeudAremplacer**. Enfin, la dernière étape (5), consiste à propager les modifications de zones dans le reste de l'arbre. Cette dernière opération est réalisée à l'aide des méthodes de propagation, **propage** (cf. § 6.5).

La partie la plus importante, détaillée dans la suite, est celle qui consiste à décompiler un nœud (2). Cette opération est réalisée grâce à un envoi de message de sélecteur **decompile**, le même sélecteur que la méthode principale. Il reste maintenant à définir cette méthode dans les différentes classes de nœuds, chaque classe répondant à ce message de façon spécifique.


```
(demethod (noeudChoix decompile) (self collecteur positionOrigine gepi)
    ;; (1) Alimentation du collecteur avec la chaîne d'affichage :
    (send 'ajoute collecteur (send 'affichage self))
    ;; (2) Mise à jour de la zone d'influence dans le texte :
    (send 'calculeZone self positionOrigine collecteur))
```

Figure 7.2. La décompilation des nœuds *choix*. La méthode **decompile** de la classe **noeudChoix**.

7.2 La décompilation au cas par cas

Toutes les catégories de nœuds doivent être en mesure de répondre au message **decompile** envoyé par la méthode principale de décompilation (cf. Fig. 7.1). L'écriture d'une unique méthode dans la classe nœud est envisageable. Cependant, une telle méthode prenant en charge tous les problèmes associés aux différentes catégories de nœuds s'avère complexe à écrire. En particulier, il est nécessaire de tester explicitement la catégorie d'appartenance d'un nœud : s'agit-il d'un nœud *atomique* ?, d'un nœud *groupe*? ou encore d'un nœud *liste* ? ...

La technique qui consiste à définir une méthode spécifique dans chaque sous-classe de la classe nœud est bien plus en rapport avec la philosophie objets. Chaque méthode est automatiquement invoquée en respect de la classe d'appartenance de l'objet receveur. Il n'est donc pas nécessaire de tester explicitement le type du nœud visité. Chaque méthode traite un cas particulier sur lequel on peut se concentrer le moment venu en faisant abstraction complète des autres cas.

Les méthodes de sélecteur **decompile** associées aux différentes classes de nœuds ont toutes le même profil. Elles comportent, en plus du receveur, trois arguments. Une méthode de décompilation définie dans une de ces classes doit avoir l'entête suivante :

```
( demethod (noeud XYZ decompile) (self collecteur positionOrigine gepi)
    ...)
```

L'argument **collecteur** référence une instance de la classe de même nom et est chargée de recevoir le flux de caractères correspondant à la décompilation du nœud receveur. La position dans le texte où débute la décompilation est donnée par l'argument **positionOrigine**. L'instance référencée, de la classe position, permet d'effectuer les éventuelles actions de paragraphage relativement à cette position de départ. Le dernier argument, **gepi**, référence une instance de la classe **gepi**. Cette instance permet en particulier d'accéder au langage associé à l'arbre à décompiler.

```
(demethod
   (noeudAtomique decompile) (self collecteur positionOrigine gepi)
   ;; (1) Alimentation du collecteur avec la chaîne d'affichage :
   (let ((listeDeCaracteres (send 'listeDeCaracteres self)))
      (if listeDeCaracteres
            ;; Le nœud est expansé :
            (send 'ajoute collecteur listeDeCaracteres)
         ;; le nœud n'est pas expansé :
         (send 'ajoute collecteur (send 'affichage self))))
   ;; (2) Mise à jour de la zone d'influence dans le texte :
   (send 'calculeZone self positionOrigine collecteur))
```

Figure 7.3. La décompilation des nœuds *atomique*. La méthode `decompile` de la classe `noeudAtomique`.

7.2.1 Décompilation des nœuds *choix*

Les nœuds instances de la classe `noeudChoix` ne possèdent pas de fils (cf. § 2.3.2) et, sont donc très faciles à décompiler (cf. Fig. 7.2). La décompilation d'un tel nœud nécessite deux opérations, dans un premier temps, la chaîne d'affichage du nœud est ajoutée dans le collecteur (1), puis, dans un deuxième temps, la zone d'influence associée au nœud décompilé est mise à jour (2).

La première phase de la décompilation d'un nœud choix consiste donc simplement à envoyer le message , ajoute au collecteur. La chaîne ajoutée est obtenue en envoyant le message `affichage` au nœud *choix* qu'il faut décompiler.

La deuxième phase de la décompilation d'un nœud choix consiste à mettre à jour la zone d'influence associée au nœud dans le texte. Comme l'opération d'affichage, cette opération est définie pour l'ensemble des nœuds. Elle est réalisée en envoyant le message `calculeZone` au nœud qui doit mettre à jour sa zone d'influence. Les deux arguments de cette méthode, `positionOrigine` et `collecteur` permettent de mettre à jour simplement la zone d'influence du nœud. L'argument `positionOrigine` est la position qui correspond au début de la zone d'influence du nœud dans le texte. La position courante associée au collecteur, la dernière position atteinte, est la position de fin de la zone d'influence associée au nœud.

Cette première méthode, définie dans la classe `noeudChoix` traite le cas de décompilation d'un nœud *choix*. Définissons maintenant une nouvelle méthode `decompile`, dans la classe `noeudAtomique` pour traiter le cas des nœuds *atomiques*.

7.2.2 Décompilation des nœuds *atomique*

Comme les nœuds *choix* les nœuds *atomique* n'ont jamais de fils. La méthode de décompilation correspondante est, comme dans le cas des nœuds *choix*, réalisée en deux étapes (cf. Fig. 7.3).

Le collecteur est alimenté avec la chaîne d'affichage du nœud (1). Lorsque le nœud *atomique* est expansé, on utilise la liste de caractères portée par le nœud lui même pour alimenter le collecteur. Lorsque le nœud n'est pas expansé, la chaine d'affichage du nœud, obtenue à l'aide de la méthode `affichage`, est utilisée pour alimenter le collecteur.

La deuxième étape, comme pour les nœuds *choix* consiste à mettre à jour la zone d'influence associée au nœud receveur (2). Comme précédemment, la méthode `calculeZone` est utilisée.

7.2.3 Décompilation des nœuds *liste*

Le cas des nœuds *atomique* étant maintenant résolu, occupons nous des nœuds *listes*. Un tel nœud peut avoir des fils et, selon la construction qui lui est associée, il peut s'agir d'une liste avec ou sans tête, avec ou sans séparateur, avec ou sans queue.

La méthode de décompilation associée à la classe des nœuds *liste*, comme les précédentes, comporte deux étapes (cf. Fig. 7.4), l'alimentation du collecteur (1) et la mise à jour de la zone d'influence associée au nœud (2).

La première étape, l'alimentation du collecteur (1), est plus compliquée que dans les cas précédents. Lorsque le nœud possède des fils, l'alimentation du collecteur est effectuée en envoyant le message `decompile` à la construction portée par le nœud *liste* (1.1). Lorsque le nœud *liste* ne possède pas de fils, la chaîne d'affichage du nœud lui-même est utilisée pour alimenter le collecteur (1.2).

Dans la classe `constructionListe`, la méthode de sélecteur `decompile` (cf. Fig. 7.5) est chargée de continuer le travail commencé par la méthode de même nom dans la classe `noeudListe`. Cette méthode comporte un argument supplémentaire, la liste des fils associés au nœud qui porte la construction. Les trois étapes de cette méthode permettent d'alimenter le collecteur avec l'affichage de la tête (1), l'affichage et la décompilation des fils (2), l'affichage des éventuels séparateurs (2.2) et enfin l'affichage de la queue (3).

N'étant plus sur une construction terminale, il ne faut pas oublier, avant et après chaque affichage, de déclencher les actions de paragraphage associées à la construction *liste*. Ces actions, définies avec le langage (cf. § 2) sont référencées par la variable `indent` du receveur (cf. § 6.3.4).

Ainsi, avant et après chaque affichage, le message `action` est envoyé au collecteur pour qu'il effectue l'action correspondante. Cette action est soit contenue dans la liste `indent`, soit, si aucune action n'a été prévue, l'action (b0), qui ne fait rien, est exécutée. L'écriture de ce test est succinte en LISP grâce à l'utilisation conjuguée de la fonction `nth` et de la fonction `or`. La fonction `nth` retourne un élément dans une liste, les éléments étant indicés à partir de zéro :

```
(demethod (noeudListe decompile) (self collecteur positionOrigine gepi)
   ;; (1) Alimentation du collecteur :
   (let ((fils (send 'fils self)))
      (if fils
            ;; Alors : (1.1) Traitement des fils :
            (send 'decompile
                  (send 'construction self)
                  collecteur
                  fils
                  positionOrigine
                  gepi)
          ;; Sinon : (1.2) Affichage du nom de construction :
          (send 'ajoute collecteur (send 'affichage self))))
   ;; (2) Mise à jour de la zone d'influence dans le texte :
   (send 'calculeZone self positionOrigine collecteur))
```

Figure 7.4. La décompilation des nœuds *liste*. La méthode `decompile` de la classe `noeudListe`.

```
? (nth 0
?      '((b 0) (b 1) (b 2) (b 3) () (b 5) (b 6) (b 7)))
= (b 0)
? (nth 4
?      '((b 0) (b 1) (b 2) (b 3) () (b 5) (b 6) (b 7)))
= ()
```

Comme en C [Kernighan 78], l'exécution de la fonction associée au prédicat ou, `or`, s'arrête dès qu'un argument à la valeur logique vraie :

```
? (or () '(b0))
= (b0)
```

Le traitement d'un item de la liste est réalisé en envoyant le message `decompile` au nœud correspondant, pris dans la liste des `fils` (2.1). La descente de l'arbre continue donc, *récursivement*, en profondeur d'abord. Le message étant envoyé à un nœud, la liste des arguments doit respecter l'entête commune à toutes les méthodes définies pour les différentes sous-classes de `noeud` (page 167). Bien entendu, il ne s'agit pas dans ce cas d'une récursivité directe : la méthode invoquée dans la méthode `decompile` de la classe `constructionListe` est bien de sélecteur `decompile`, mais le message est envoyé à une instance d'une sous-classe de `noeud`. Selon la structure de l'arbre décompilé, il peut donc se produire ou non une exécution récursive. Par exemple, lorsque qu'un fils d'un nœud *liste* est lui-même un nœud *liste*.


```
(demethod
  (constructionListe decompile) (self collecteur fils position gepi)
  (let ((indent (send 'indent self)))
    ;; (1) Actions de paragraphage et affichage de la tête :
    (send 'action collecteur (or (nth 0 indent) '(b0)) position)
    (send 'ajoute collecteur (send 'tete self))
    (send 'action collecteur (or (nth 1 indent) '(b0)) position)
    ;; (2) Traitement des fils :
    (while fils
       ;; (2.1) Traitement d'un item :
       (send 'action collecteur (or (nth 2 indent) '(b0)) position)
       (send 'decompile
          (nextl fils)
          collecteur
          (send 'position collecteur)
          gepi)
       (send 'action collecteur (or (nth 3 indent) '(b0)) position)
       ;; (2.2) Actions de paragraphage et affichage du séparateur :
       (when fils
          (send 'action collecteur (or (nth 4 indent) '(b0)) position)
          (send 'ajoute collecteur (send 'separateur self))
          (send 'action collecteur (or (nth 5 indent) '(b0)) position)))
    ;; (3) Actions de paragraphage et affichage de la queue :
    (send 'action collecteur (or (nth 6 indent) '(b0)) position)
    (send 'ajoute collecteur (send 'queue self))
    (send 'action collecteur (or (nth 7 indent) '(b0)) position)))
```

Figure 7.5. La décompilation des fils d'un nœud *liste*. La méthode `decompile` de la classe `constructionListe`.

7.2.4 Décompilation des autres nœuds

Comme nous venons de le voir sur trois catégories de nœuds, l'algorithme de décompilation est *réparti* dans les différentes classes de nœuds à l'aide des méthodes de sélecteur `decompile`. Bien entendu, on ne peut pas dire que les différentes méthodes de sélecteur `decompile` sont indépendantes les unes des autres car ces méthodes doivent respecter certaines conventions communes. Par exemple, toutes les méthodes définies dans les sous-classes de `noeud` doivent avoir le même nombre d'arguments. En outre, par convention, chaque méthode est chargée d'ajouter dans le collecteur le fruit de la décompilation du nœud correspondant à sa classe. Cependant, même si cette répartition doit être accompagné d'un respect rigoureux de conventions tacites, elle permet de traiter chaque cas séparément en faisant abstraction des autres cas. La décompilation des nœuds *atomique* est traitée sépa-

rément de la décompilation des nœuds *listes* par exemple. En plus de ce pouvoir d'abstraction, cette répartition permet de tester séparément les différentes méthodes. Par exemple, on peut tester dès maintenant les méthodes que nous venons d'écrire sur un arbre ne comportant que des constructions *choix*, *atomiques* ou *liste*.

Encore une fois, le mécanisme de liaison dynamique s'avère précieux en nous évitant l'écriture d'un bon nombre de tests. En effet, en aucun cas il n'est nécessaire de tester la catégorie d'appartenance d'un nœud ou d'une construction. Si d'aventure une nouvelle classe de nœuds ou de constructions venait à être rajoutée, les méthodes existantes n'ont pas besoin d'être modifiées, le mécanisme de liaison dynamique se chargeant de rediriger les envois de message `decompile` sur cette nouvelle classe.

La décompilation des autres nœuds, nœuds *groupe* et nœuds *sucre* fait appel, comme dans les cas précédents, à des méthodes de sélecteur `decompile` respectivement dans les classes `noeudGroupe`, `constructionGroupe`, `noeudSucre` et `constructionSucre`. Ces méthodes ne sont pas plus compliquées à écrire que celles définies pour les nœuds *liste*, le principe restant le même. Le lecteur intéressé trouvera en annexe 9 le programme source de ces méthodes.

Les nœuds *textes*, aussi bizarre que cela puisse paraître peuvent aussi faire l'objet d'une décompilation. Par exemple, lorsque l'on remplace un nœud *groupe* par un nœud *texte* en cas d'effacement d'un caractère dans la zone correspondante. Inutile de préciser que la décompilation des nœuds *textes* est fort simple puisqu'elle ne consiste qu'à ajouter le texte correspondant dans le collecteur. Bien entendu, comme dans les autres cas de nœuds, ce travail est réalisé à l'aide d'une méthode spécifique de sélecteur `decompile`, définie cette fois dans la classe `noeudTexte`.

7.3 Déclenchement des actions de paragraphage

Le déclenchement des actions de paragraphage fait appel au mécanisme d'envoi de messages ainsi qu'à la possibilité de calculer dynamiquement le sélecteur d'un message[2]. L'utilisation de ce mécanisme permet d'une part d'implanter simplement le déclenchement des actions de paragraphage et d'autre part de ne pas figer le jeu d'actions de paragraphage utilisé par le décompilateur.

Comme nous l'avons vu précédemment, les actions de paragraphage sont déclenchées en envoyant soit le message `action`, soit le message `actionParDefaut` au collecteur. Si l'on ne regarde pas l'implantation de ces méthodes dans la classe `collecteur`, on peut penser par exemple, que les différentes actions de paragraphage sont rangées dans une table, et que les méthodes `action` et `actionParDefaut` se chargent de la consultation de cette table et déclenchent dans un deuxième temps l'exécution correspondante.

2. Le calcul dynamique du sélecteur est réalisé en smalltalk à l'aide de la primitive `perform:`. Cette possibilité n'est en général offerte que par les langages à objets suffisamment interprétés, comme ceux construits au dessus de LISP par exemple (cf. § 5.1.2).


```
(demethod (collecteur action) (self action positionOrigine)
    (send (car action)
          self
          (cdr action)
          positionOrigine))
```

Figure 7.6. Un exemple d'envoi de message avec calcul dynamique du sélecteur. Le corps de la méthode `action` consiste à envoyer le message de sélecteur "`(car action)`" au collecteur.

En fait, chaque action de paragraphage est implantée à l'aide d'une méthode définie dans la classe `collecteur`. Par convention, le sélecteur associé à une action de paragraphage est le nom de l'action elle même. Par exemple, à l'action `(cr)` correspond la méthode de sélecteur `cr` de la classe collecteur. De même, à l'action `(b N)` correspond la méthode de sélecteur `b` de la classe collecteur. D'une façon générale, le premier élément de la liste constituant l'action de paragraphage correspond au sélecteur de la méthode associée à l'action et, le reste de la liste correspond aux arguments éventuels de l'action elle-même :

```
? ;; Le sélecteur associé à une action :
? (car '(b 6))
= b
? ;; La liste d'arguments de l'action :
? (cdr '(b 6))
= (6)
```

La liste LISP correspondant à l'action de paragraphage est passée en argument aux méthodes `action` et `actionParDefaut`. Dans le corps de ces méthodes, le sélecteur du message à envoyer au collecteur est calculé à l'aide de la fonction LISP `car` (cf. Fig. 7.6). Le reste de la liste qui constitue les éventuels arguments de l'action de paragraphage, obtenu avec la fonction LISP `cdr` sont passés en argument à la méthode de paragraphage.

Cette technique ne laisse pas le choix quant à l'entête des différentes méthodes de paragraphage. Qu'il s'agisse d'une action avec ou sans arguments, la méthode qui lui est associée possède deux arguments en plus du receveur, la liste d'arguments de l'action de paragraphage elle-même et la position d'origine associée à l'action :

```
( demethod (collecteur actionX) (self arguments positionOrigine)
  ...)
```

Voici par exemple la méthode qui est associée à l'action `(b1)`. Cette méthode consiste simplement à ajouter un caractère espace dans le collecteur. Dans ce cas, les deux arguments de la méthode ne sont pas utilisés :

```
( demethod (collecteur b1) (self arguments positionOrigine)
    ( send 'ajoute self " "))
```

Si l'action de paragraphage comporte des arguments, ceux ci sont utilisables à l'intérieur de la méthode de paragraphage :

```
( demethod (collecteur b) (self arguments positionOrigine)
    (let ((nombreDeBlancs (car arguments)))
      (repeat nombreDeBlancs
          ( send 'ajoute self " "))))
```

Si cette technique impose une entête identique pour toutes les méthodes associées aux actions de paragraphage, elle permet en revanche de réaliser simplement la recherche et le déclenchement d'une action. Dans ce cas, le mécanisme de transmission de messages effectue automatiquement la sélection de l'action à déclencher. En outre, il est possible de rajouter de nouvelles actions de paragraphage sans rien changer aux méthodes de décompilation. La définition d'une nouvelle action de paragraphage se limite à la définition d'une nouvelle méthode dans la classe `collecteur`.

Pour terminer cette section sur une note reposante, voici le texte source de la méthode associée à l'action (`b0`) (page 52), chargée de ne rien faire :

```
( demethod (collecteur b0) (self arguments positionOrigine))
```

7.4 Analyse syntaxique

La souplesse d'utilisation du manipulateur de documents est directement liée à la possibilité de manipuler un arbre comportant des feuilles *textes*, c'est-à-dire des parties non syntaxiquement validées. Naturellement, sur de telles parties, les commandes structurelles ne sont applicables qu'après avoir effectué une analyse chargée de reconstruire l'arbre syntaxique correspondant. La mise en œuvre d'un analyseur syntaxique pour le manipulateur de documents est donc essentielle.

7.4.1 Problèmes spécifiques

Si les travaux sur l'analyse syntaxique sont nombreux[3], il est cependant difficile de les réutiliser directement car l'analyseur syntaxique est soumis à certaines contraintes inhérentes aux choix effectués pour le manipulateur de documents. A l'usage, il s'avère que l'analyseur idéal du manipulateur de document doit avoir les caractéristiques suivantes :

1. Changement d'axiome : l'axiome de l'analyseur syntaxique doit pouvoir être interchangeable afin de ne pas recommencer inutilement l'analyse syntaxique sur l'ensemble du document. En effet, dans de nombreux cas, la *réparation* syntaxique peut être effectuée localement. Par exemple, lorsque le corps d'une procédure est modifié, il est plus économique de ne recommencer l'analyse syntaxique que pour le corps de cette procédure.
2. Analyse des constructions non-terminales : une portion de texte à analyser peut contenir des noms de constructions non-terminales. Dans ce cas, les noms de constructions doivent être considérés comme des terminaux éventuels de la grammaire concrète.
3. Catégories de grammaires acceptées : la classe des grammaires acceptées par l'analyseur doit être la plus large possible afin de laisser un maximum de liberté quant à l'écriture des descriptions de langages.
4. Interactivité : l'utilisation de l'analyseur est interactive. Cette caractéristique permet, dans un premier temps, de laisser de coté les problèmes liés à la reprise de l'analyse en cas d'erreur. En effet, on peut imaginer qu'un dialogue s'établit dès qu'une erreur est détectée et que la correction de cette erreur est immédiate.

Ces différentes contraintes posent le problème du choix de l'analyseur syntaxique. De toute évidence, les produits existants comme Yacc [Aho 74] [Berry 84] ne peuvent pas être réutilisés, en particulier, à cause de la contrainte No 2. Pas moyen d'y échapper, il faut reconstruire un analyseur en partant de zéro[4]...

7.4.2 Choix de la technique d'analyse

Parmi les méthodes d'analyse syntaxique, deux grandes familles se distinguent, les méthodes *descendantes* et les méthodes *ascendantes*. L'analyse descendante part de l'axiome de la grammaire et tente de créer une chaîne identique à la chaîne d'entrée par *dérivations* successives. Ainsi, l'opération de base est le remplacement d'un symbole non-terminal par une de ses parties droites. On dit que l'axiome est la première *cible*, dont on dérive d'autres cibles par la répétition de l'opération de remplacement. Un analyseur ascendant travaille de façon inverse. Il trouve une chaîne de symboles identiques à ceux formant une partie droite d'une règle de grammaire, et leur substitue le symbole non-terminal correspondant. Cette opération porte le nom de *réduction* . En travaillant par réductions sucessives, l'analyseur essaie de retrouver l'axiome de la grammaire.

Qu'il s'agisse d'une méthode descendante ou ascendante, le problème de base de l'analyse syntaxique reste identique : comment prendre à chaque étape d'une

3. Pour débuter en compilation, [Cunin 80], puis [Gries 71] et bien sûr le célèbre *Dragon* [Aho 77]. Pour un développement formel des grammaires et des langages, le lecteur peut consulter [Lentin 67], [Gross 70] ou [Aho 73].

4. C'est d'une main ferme et décidée que j'ai repris le polycopié de Pierre Marchand [Marchand 78], professeur à l'université de Nancy I. Je tiens à remercier Pierre pour les réponses qu'il a donné à mes questions souvent naïves sur la théorie des langages. J'ai d'ailleurs accepté, suite à ces questions, de suivre une seconde fois les cours et les travaux dirigés qu'il donne avec maestria aux étudiants de maîtrise. Si certains détails m'échappent encore, ce deuxième passage m'a fait le plus grand bien. Merci Pierrot.

analyse la bonne direction, qu'il s'agisse d'une dérivation (méthode descendante) ou d'une réduction (méthode ascendante). Pour augmenter l'efficacité des analyseurs, de nombreux travaux sont à l'origine d'une classification des grammaires et des algorithmes d'analyse. Les catégories de grammaires LL(k) et LR(k) permettent respectivement l'implantation d'algorithmes descendants déterministes [Foster 68] et acendants déterministes [Knuth 65].

S'il est vrai que les grammaires LR(k) prennent en compte une classe plus large de langage que les grammaires LL(k)[5], les difficultés de mise en oeuvre d'un analyseur ascendant nous ont poussés à écarter cette méthode. En outre, les spécificités de l'analyseur du manipulateur de documents (page 174) ne facilite pas le calcul des tables d'analyse habituellement associées aux analyseurs LR(k)[6].

Après une première implantation de l'algorithme de Jay Earley [Earley 70] qui se révéla coûteuse en temps d'exécution[7], nous sommes revenus à une méthode descendante[8], connue sous le nom de méthode parallèle, ou encore analyse prédictive,très ancienne, venant de [Greibach 64] [Greibach 65]. Cette méthode est très simple à mettre en œuvre et consiste à essayer toutes les dérivations possibles, en *parallèle*, jusqu'à ce qu'il soit possible d'en éliminer. Bien entendu, cette méthode n'est pas aussi générale que celle d'Earley. En particulier, certaines récursivités à gauche ne peuvent pas être traitées par un tel analyseur. En fait, il est généralement nécessaire d'éliminer la récursivité à gauche dans une grammaire, chaque fois que l'on veut utiliser une méthode d'analyse descendante [Cunin 80, page 48]. Signalons que la théorie assure que cette élimination est toujours possible et qu'il existe des algorithmes satisfaisants pour le faire automatiquement [Bordier 71].

5. Si un langage permet une analyse déterministe, alors, il est possible de trouver une grammaire LR(1) pour ce langage. La méthode LL(1) ne s'applique qu'à un sous-ensemble des langages déterministes [Cunin 80, page 75].

6. Différents algorithmes permettant la génération de tables pour analyseurs LR(1) répondant à la contrainte No 2 (page 174) sont donnés dans [Ghezzi 80], [Wegman 80] et [Jalili 82]. Pour les grammaires LR(k), un algorithme Pascal est donné dans [Degano 88]

7. Cet algorithme est ascendant et descendant à la fois : "It is similar to both Knuth's LR(k) algorithm and the familiar top-down algorithm". Il est capable de prendre en compte de façon *efficace* tout type de grammaire à contexte libre. Le facteur d'efficacité annoncé concerne le nombre d'étapes de bases de l'algorithme et dépend de la longueur de la chaîne à analyser ainsi que de la classe de grammaire utilisée. Toujours selon son auteur, ce facteur est linéaire pour une large classe de grammaire avec une limite en n2 pour les grammaires non ambiguë et pouvant aller jusqu'en n3 pour ces dernières. Après avoir adapté cet algorithme aux spécificités du manipulateur de documents, l'efficacité de cet algorithme s'est considérablement dégradée. Si effectivement le nombre d'étapes d'opérations de bases de l'algorithme n'a pas augmenté de façon désatreuse sur les essais que nous avons effectués, en revanche, le temps de calcul nécessaire à la réalisation d'une étape a augmenté de telle façon que nous avons jugés les performances de l'algorithme insufisantes pour une utilisation interactive.

8. Un analyseur descendant incrémental pour grammaires LL(1) est implanté dans le système PSG [Bahlke 86]

7.4.3 Adaptation de la méthode prédictive

La technique d'analyse prédictive que l'on peut mettre en œuvre de façon immédiate en utilisant un algorithme récursif présente l'avantage d'être facilement adaptable aux contraintes imposées par le manipulateur de documents (page 174). Par exemple, la contrainte No 2 qui consiste à considérer tout non-terminal comme un terminal potentiel du texte source consiste à ajouter une nouvelle possibilité de reconnaissance pour chaque non-terminal du langage.

Notre première expérimentation de la méthode prédictive consistait à dérouler l'algorithme sur une grammaire classique (forme BNF) calculée à partir des différentes constructions constituant le langage. Ce calcul est évident et consiste principalement à transformer les constructions *listes* à l'aide de plusieurs productions dont une possède deux alternatives et est récursive à droite (page 45). Les constructions *groupe* ne possédant pas de partie optionnelle peuvent pour leur part être traduites en productions à alternative unique. Il en est de même pour les constructions *sucres*. Les constructions *choix* sont quant à elles traduites par des productions dont le nombre d'alternatives correspond au nombre de choix offerts.

Dans la version actuelle, la phase de traduction qui consiste à calculer une grammaire BNF à partir de la description du langage a été abandonnée : les constructions du langage sont maintenant directement interprétées par l'analyseur syntaxique. La stratégie d'analyse reste cependant la même : elle est descendante et consiste à explorer toutes les possibilités d'analyse. L'interprétation directe des constructions pour l'analyse syntaxique du texte source présente plusieurs avantages. La mise en œuvre est simple, s'est avérée plus efficace et est aussi plus conviviale. En outre, l'implantation de l'algorithme d'analyse à fait l'objet d'un découpage similaire à l'implantation de l'algorithme de décompilation.

Efficacité

Du point de vue de l'efficacité, le gain est lié à la suppression d'un bon nombre de productions à alternatives multiples. Ces productions sont en effet celles qui nécessitent une analyse parallèle et qui sont donc les plus coûteuses. En particulier, les constructions *listes* sont maintenant analysées à l'aide d'un simple automate. L'automate correspondant est par ailleurs fort simple à écrire puisqu'il consiste à reconnaître le marqueur de tête, puis la suite d'items répétés éventuellement séparés par une chaîne particulière et enfin, à reconnaître le marqueur de fin de répétition. Une technique identique est adoptée pour traiter le cas des constructions *groupe* qui possèdent une partie optionnelle. Finalement, seule l'analyse des constructions *choix* nécessite une analyse en parallèle de plusieurs alternatives.

Convivialité

La nouvelle méthode évite la création de non-terminaux intermédiaires, nécessaires lors du passage en forme BNF. Tous les messages d'erreurs peuvent ainsi être exprimés en terme de nom de constructions, plus familiers à l'utilisateur. Ces

noms de constructions, déjà utilisés dans la description du langage sont aussi utilisés lors des développements par exemple. En outre, l'analyse syntaxique d'une construction *liste* à l'aide d'un automate permet cerner plus facilement la cause d'une erreur de syntaxe. En effet, si l'automate chargé de la reconnaissance d'une construction *liste* a déjà rencontré le marqueur de début de liste ainsi que quelques items correctement séparés, il peut à juste titre émettre une hypothèse quant à la nature de l'erreur de syntaxe détectée. Dans ce cas, une éventuelle correction automatique est envisageable. L'analyseur étant utilisé de façon interactive, une confirmation ou une indication permettant la correction effective peut être envisagée.

Implantation

Du point de vue du codage proprement dit, l'algorithme est réalisé en utilisant une technique similaire à celle du décompilateur (cf. § 7.2), c'est-à-dire au cas par cas. Les différentes classes de constructions sont munies de méthodes de même sélecteur, **analyse**, qui s'appellent de façon récursive. Comme les méthodes de décompilation, ces méthodes sont liées entre-elles par un ensemble de conventions communes. Ces conventions sont du même ordre que les conventions qui lient les méthodes de décompilation et permettent de traiter les différents cas d'analyse de façon séparée. L'analyse des constructions *choix* est effectuée indépendamment des constructions *groupe* par exemple. De même les deux catégories de listes, *liste0-n* et *liste1-n* sont traitées de façon séparée dans les classes correspondantes, `constructionListe0-n` et `constructionListe-n` (cf. Fig. 6.10 page 149). Comme dans le cas de la décompilation, de nombreux tests sont évités (s'agit-il d'une construction *liste0-n* ou *liste1-n* par exemple) et les différentes méthodes peuvent être mises au point séparément (analyse d'une construction *atomique* isolée par exemple).

7.4.4 Le problème de l'ambiguïté

Dans certains cas, lorsque la grammaire décrivant un langage est mal choisie, il existe pour un même texte source plusieurs arbres syntaxiques corrects en regard de cette grammaire. L'ambiguïté est un problème réel, en particulier parce qu'il n'existe pas d'algorithme général permettant de décider si une grammaire est ambiguë ou non.

En outre, les constructions non-terminales pouvant faire partie du texte source à analyser, le degré d'ambiguïté de la grammaire[9] peut augmenter. Par exemple, avec les trois constructions suivantes :

9. Le degré d'ambiguïté d'une phrase à reconnaître est le nombre d'arbres de dérivation distincts qui peuvent être construits en analysant cette phrase. Une grammaire a un degré d'ambiguïté fini si, pour chaque phrase reconnue par cette grammaire le degré d'ambiguïté est fini.


```
(A
    ( choix B C))
(B
    ( atomique "C"))

(C
    ( atomique "le vrai C"))
```

La chaîne d'entrée `"C"` peut donner lieu à la construction de deux arbres différents selon que l'on considère qu'il s'agit de la construction atomique `B` expansée ou de la construction `C`, non expansée.

L'analyse prédictive explorant toutes les possibilités de dérivation permet la détection des ambiguïtés et, l'analyseur étant interactif, il est possible d'établir un dialogue avec l'utilisateur pour choisir la meilleure représentation arborescente.

7.5 Bilan

A l'usage, la méthode d'analyse prédictive donne des résulats tout à fait acceptables pour les descriptions des langages que nous avons testés (miniPascal, ldl, C ou Ada). Ces résultats *acceptables* sont certainement liés à l'interprétation directe des constructions *listes* à l'aide d'automates. En effet, dans les grammaires BNF décrivant ces langages, un bon nombre de productions à alternatives multiples sont nécessaires pour décrire les répétitions (liste d'arguments, liste de déclarations, liste d'instructions ...). L'interprétation directe des constructions *listes* diminue le nombre de possibilités qui doivent être essayées en parallèle par l'analyseur. Par exemple, la description du langage miniPascal (page 48) compte uniquement 5 constructions *choix*. Les 4 constructions *listes* utilisées suppriment autant de production à alternatives multiples. De même l'interprétation directe des constructions *groupe* possédant une partie optionnelle permet également d'éviter l'analyse parallèle de plusieurs alternatives.

Il est difficile de comparer de façon académique différentes méthodes d'analyse car la catégorie de grammaire acceptée est souvent un facteur d'efficacité primordial. Dire qu'une grammaire de la forme LL(k) est plus intuitive à écrire qu'une grammaire LR(k) ou l'inverse reste quoi qu'on dise un critère de comparaison purement subjectif.

Du point de vue du codage, l'algorithme de décompilation et l'algorithme d'analyse syntaxique ont plusieurs points communs. Tous les deux travaillent en interprétant directement la description du langage donnée sous la forme de constructions. Le point commun le plus intéressant est cependant le découpage de ces algorithmes dans différentes classes. Même si les méthodes correspondantes sont intimement liées par un ensemble de conventions communes, le découpage de la difficulté algorithmique est évident. D'un point de vue pratique, ce découpage permet une mise au point séparée et progressive des différents cas d'analyse ou de décompilation. Le contexte interprété de l'environnement utilisé étant lui aussi un facteur important pour la phase de mise au point.

Les algorithmes de décompilation et d'analyse syntaxique sont par ailleurs très agréables à écrire, l'utilisation combinée de la récursivité et de la liaison dynamique permettant une puissance d'expression intéressante. La concision de ces programmes est bien entendu liée au domaine traité qui se prête bien à une écriture récursive. Pour sa part, le mécanisme de liaison dynamique permet d'éviter un bon nombre de tests (cf. § 7.4.3 et § 7.2.4).

S'il est difficile d'évaluer le temps d'exécution de l'analyseur syntaxique car une analyse dépend de la classe de la grammaire utilisée, la complexité de l'algorithme de décompilation est exactement celui d'un parcours d'arbre en profondeur d'abord. La terminaison de ces deux algorithmes *récursif* peut être prouvée simplement par le fait que les appels récursifs sont toujours effectués pour résoudre un sous-problème du problème traité.

Le mécanisme de calcul dynamique du sélecteur, utilisé pour le déclenchement des actions de paragraphage (cf. § 7.3), peut être comparé au mécanisme Pascal qui consiste à définir une procédure prenant un argument qui est lui même une procédure. Le mécanisme de Pascal est cependant moins puissant puisque, pour un appel particulier, la procédure donnée en argument est désignée statiquement. Un sélecteur passé en argument peut quant à lui correspondre à une famille de procédures de même nom, qui seront déclenchées correctement en regard de la classe d'appartenance du receveur[10].

Un point important de la programmation objet est le haut niveau d'abstraction qu'il est possible d'atteindre, les instances d'une classe pouvant être manipulées sans connaître leur implantation, mais seulement à l'aide de sélecteurs auxquelles ces instances savent répondre. Bien entendu, l'utilisation d'un langage à objets ne force pas le programmeur à utiliser cette possibilité. On peut écrire en leloup comme en Smalltalk des programmes allant à l'encontre de ce principe. De ce point de vue, l'écriture de *jolis* programmes reste un art difficile pour lequel le meilleur apprentissage passe souvent par l'analyse de ses échecs.

10. En général, les langages à objets compilés comme Eiffel par exemple, ne permettent pas définir une méthode comportant un argument du type *sélecteur*. Cette particularité est par exemple une des clefs de l'implantation du systèmes MVC de Smalltalk (cf. § 4.3.2).

8
Bilan

" Object Oriented Programming is a phrase that is beginning to catch on, just like the phrase structured programming did in the '70s. "

E. Horowitz [Horowitz 83]

Le bilan de cette thèse est effectué selon les points de vue suivants : manipulation de documents (§ 8.1), aspects du langage de description de langages (§ 8.2), résultats d'analyse syntaxique (§ 8.3), programmation objet (§ 8.4) et enfin leloup (§ 8.5).

8.1 Manipulation de documents

L'édition syntaxique est, rappelons le, le point de départ de cette thèse. Le manipulateur de documents conçu dans le cadre du projet GÉPI se distingue des produits existants par sa souplesse d'emploi. Les deux modes d'éditions, édition textuelle et édition syntaxique sont intimement mélés : on garde les avantages de l'édition syntaxique sans rien perdre de la souplesse d'utilisation des éditeurs de textes classiques. Les éditeurs comme Cépage ou CPS, actuellement les meilleurs produits sur le marché, n'ont pas cette souplesse d'utilisation (cf. § 1.1.2 et § 1.2) et, en un mot, leur utilisation est trop contraignante. A la manière du manipulateur de documents de GÉPI, il est primordial de conserver tous les avantages de l'édition textuelle classique si l'on veut banaliser l'accès à l'édition syntaxique. De ce point de vue, le bilan est donc positif.

8.1.1 Couplage

La plupart des éditeurs syntaxiques ne travaillent que sur une seule représentation du document qu'ils manipulent : sa représentation arborescente. En outre, ils imposent que cette représentation soit toujours correcte en regard de la syntaxe

associée au document. Ce choix est de toute évidence à l'origine de leur manque de souplesse. Pour intégrer les deux modes d'éditions, la solution que nous avons choisie consiste à coupler deux représentations d'un même document, sa représentation textuelle et sa représentation arborescente. De plus, afin de conserver toutes les possibilités de l'édition textuelle, la représentation arborescente peut comporter des feuilles correspondant à du texte syntaxiquement incorrect (cf. § 1.2 et § 6.1). L'inconvénient de cette approche réside dans la difficulté d'implantation du couplage entre les deux représentations qui doivent rester cohérentes à tous moments. Le problème est d'autant plus complexe que les deux représentations d'un même document sont complémentaires. Il n'est pas possible de calculer la représentation arborescente d'un document comportant des erreurs de syntaxe. Inversement, il n'est pas possible de calculer la représentation textuelle à partir de la représentation arborescente, cette dernière ne contenant pas le détail des bruits comme les commentaires par exemple.

La technique de couplage par contraintes (page 99) est attrayante pour deux raisons. D'une part elle évite l'apparition d'incohérences entre les objets couplés, le système assurant automatiquement le respect des contraintes. D'autre part, elle permet d'isoler la partie couplage du reste de l'application. Si cette technique peut s'appliquer à de nombreux cas d'interfaçages [Carter 84] [Borning 86], il nous semble difficile de l'appliquer au le manipulateur de documents pour plusieurs raisons. En premier lieu, l'application de cette technique suppose que le langage d'écriture de contraintes permet d'exprimer une relation de correction syntaxique entre un texte et une représentation arborescente. Les contraintes doivent réaliser l'équivalent de la décompilation pour répercuter les modifications effectuées sur l'arbre et l'équivalent de l'analyse syntaxique pour répercuter les modifications faites sur le texte. Une écriture complètement déclarative d'une telle relation est difficilement envisageable. En deuxième lieu, le déclenchement automatique du mécanisme de vérification des contraintes devrait être modulable étant entendu qu'il n'est pas réaliste de lancer une analyse syntaxique après chaque modification du texte du document.

La solution que nous avons adopté pour implanter le couplage est complètement procédurale. Contrairement à une approche par contraintes, si l'on désire ajouter une nouvelle commande au manipulateur de documents, aucun mécanisme automatique ne garantit que cette commande conserve la cohérence entre les deux représentations du document. Néanmoins, pour la programmation des nouvelles commandes, trois opérations de base ont été isolées pour prendre en compte le respect du couplage : l'algorithme de propagation d'une modification (cf. § 6.5), l'algorithme de décompilation (cf. § 7.2) et enfin l'analyseur syntaxique (cf. § 7.4). La première de ces opérations, l'algorithme de propagation, est la plus utilisée (page 152). Comme pour la décompilation, la complexité de cet algorithme est linéairement dépendante du nombre de nœuds de l'arbre.

8.2 Le langage de description de langages

La définition du langage de description de langages (cf. chap. 2), a été effectuée en parallèle avec la mise au point du manipulateur de documents, de prototypes en prototypes. La version actuelle de ce langage a été testée sur miniPascal (cf. § 2.4) et sur des langages *réels* comme C et Ada. Dans ce dernier cas, l'écriture de la description de langage est presque aussi simple que l'écriture de diagrammes de syntaxes (cf. annexe 11). La description du langage de description de langages lui-même a également été effectuée (cf. annexe 10). Cette dernière ne présente d'ailleurs qu'un intérêt limité car la syntaxe de ce langage est très pauvre et que les catégories de constructions sont peu nombreuses. En effet, pour tous les langages dont la syntaxe est issue du langage *lisp*, comme leloup par exemple, l'utilisation du manipulateur de documents ne présente guère d'intérêt, une simple mise en correspondance visuelle des parenthèses étant suffisante pour la vérification syntaxique.

Rappelons que la définition du langage de description de langages est fortement inspirée de LDL, le langage de description de langage de l'éditeur Cépage [Meyer 87c]. La principale idée de LDL consiste à regrouper dans une même description la syntaxe concrète et la syntaxe abstraite d'un langage. Deux nouvelles catégories de constructions ont été ajoutées, les constructions *sucre* (cf. § 2.3.5) et les constructions *liste1-n* (cf. § 2.3.3). Certaines redondances du langage de Cépage ont été évitées. En outre, la description du paragraphage, de la lexicographie et des commentaires est prise en compte de façon plus systématique. Si les améliorations apportées au langage LDL permettent de réduire la taille des descriptions, les principes de base de ce langage sont conservés.

8.3 Analyse syntaxique

Dans le contexte du manipulateur de documents, l'analyse syntaxique présente certaine spécificités comme par exemple la présence éventuelle de noms de constructions[1] dans le texte source ou encore par le fait qu'il est nécessaire de changer la construction de départ[2] de l'analyse (cf. § 7.4.1). L'originalité de la technique utilisée consiste à interpréter directement la description du langage en termes de constructions pour réaliser l'analyse (cf. § 7.4.3). L'utilisation d'un formalisme de plus haut niveau que les grammaires BNF pour effectuer l'analyse syntaxique permet de contourner quelques problèmes liés à ce domaine. Si les résultats sont encourageants (cf. § 7.4.3), il est à l'heure actuelle prématuré de crier victoire, l'analyseur n'ayant pas encore fait l'objet de tests suffisamment conséquents pour être considéré comme *optimal*.

En outre, parmi les évolutions prévues pour le manipulateur de documents, l'une d'entre elles consiste à utiliser dans la fenêtre texte plusieurs fontes de ca-

1. L'équivalent des non-terminaux.
2. L'équivalent de l'axiome.

ractères (page 12). De cette manière, on peut visualiser différemment les noms de constructions, les terminaux, les zones incorrectes syntaxiquement etc. Si l'information visuelle est très utile pour l'utilisateur de l'éditeur, elle l'est également pour l'analyseur syntaxique qui peut utiliser cette information pour distinguer, au même titre que l'utilisateur, les noms de constructions des terminaux du langage par exemple. Ce dernier point doit permettre de régler plus facilement les problèmes d'ambiguïtés provenant de la présence de noms de constructions dans le texte source (cf. § 7.4.4).

8.4 Programmation objet

Du point de vue de la technique d'implantation utilisée, l'approche objet s'est révélée être un excellent outil de prototypage (cf. § 4.3). Dès que le problème à résoudre fait appel à de nombreux composants, l'intérêt de la liaison dynamique et de l'héritage est évident pour la réutilisation de composants existants (cf. § 4.1.3). Selon nous, seuls les langages disposants de l'équivalent de ces deux mécanismes peuvent porter le qualificatif de langage à objets (cf. § 3.4). En particulier, même si les avis sont partagés en ce qui concerne le langage Ada [Buzzard 85] [Seidewitz 87] [Touati 87], nous considérons que le langage Ada n'est pas un langage à objets. Si l'absence d'héritage peut, dans une certaine mesure, être contourné (cf. § 4.1.2), la simulation de la liaison dynamique en Ada (cf. Fig. 4.11, page 85) pose des problèmes de maintenance.

Pour développer l'interface du manipulateur de documents, aucun des principaux composants n'a été réécrit. L'éditeur pleine page, l'éditeur d'arbres, les menus, les boutons et les barres de défilement ont été réutilisés tels quels. Le code qu'il a fallu écrire[3] prend en charge l'intégration de ces composants et résout les problèmes spécifiques au couplage, à la décompilation et à l'analyse syntaxique.

Un aspect important de la liaison dynamique est de permettre l'utilisation aisée de structures de données hétérogènes (cf. § 3.3.2) en réduisant le nombre de tests à écrire. De telles structures facilitent l'écriture d'un bon nombre d'applications et ne sont pas utiles seulement dans le domaine de l'édition syntaxique ou de la gestion de stock. Pour ne donner qu'un exemple de plus, un logiciel tel que MacDraw du Macintosh [Lu 84] est de toute évidence plus simple à écrire et à maintenir avec un tel mécanisme[4] [Doyle 86] [Schmucker 86a].

3. La taille du texte source constituant l'implantation actuelle du manipulateur de documents avec saisie des descriptions de langages, analyse syntaxique, décompilation, couplage etc. fait état de 122351 caractères, soit 4136 lignes. 685 de ces lignes sont consacrées à l'implantation de leloup au dessus de la couche objet de Le-Lisp et 474 lignes sont des lignes de commentaires, finalement indispensables. Bien entendu, il s'agit du nombre actuel de lignes de code. Le nombre de lignes effectivement produites puis jetées tout au long du projet est très difficile à estimer et avoisine certainement les 50000 lignes. Un rapide calcul me fait actuellement penser que j'ai passé environ 6 heures pour obtenir chacune des lignes de l'implantation actuelle !

4. Le principe de base de Macdraw consiste à manipuler simultanément des objets graphiques de différentes natures. Les réactions de ces objets aux différentes interventions de l'utilisateur peuvent être traitées séparément dans les classes associées à ces catégories d'objets graphiques.

8.4.1 Le typage

En prenant du recul, le problème de la liaison dynamique pose celui du typage : vaut-il mieux typer statiquement comme Ada (cf. page 83), dynamiquement comme Smalltalk (cf. § 3.3.1) ou encore choisir une solution intermédiaire comme Eiffel (cf. page 70) ? Un telle question ne peut pas appeller une réponse catégorique. Cependant, à notre avis, le typage statique d'un langage comme Ada restreint considérablement la puissance d'expression du langage en rendant difficile, voire dangereuse, la manipulation de structures de données hétérogènes (cf. § 4.1.3). En effet, dans le cas ou l'on désire manipuler de telles structures en Ada, il faut assurer à la main le respect de l'adéquation d'une opération avec la catégorie de l'objet manipulé. Par exemple, lorsqu'on décompile un nœud, il faut tester s'il s'agit d'un nœud portant une construction *groupe*, une construction *sucre*, une construction *liste*...
En outre, cette adéquation entre l'action à réaliser et la nature de l'objet doit souvent être mise en place pour plusieurs opérations. Par exemple, pour l'analyse syntaxique, la séquence de tests précédente doit être répétée. Il en est de même pour les opérations couper copier et coller du manipulateur de documents. Si d'aventure une nouvelle catégorie d'objets vient à être rajoutée, toutes ces séquences de tests doivent être mises à jour. Une erreur intervenant dans cette mise à jour ne rend pas pour autant le programme incorrect vis à vis du typage statique auquel il est soumis et, les problèmes à l'exécution qui s'en suivent sont imprévisibles.

Le mécanisme de liaison dynamique commun à Smalltalk et à Eiffel permet d'éviter l'écriture des tests et donc de supprimer les problèmes de mise à jour induits. En Eiffel la sécurité est accrue car le compilateur vérifie statiquement l'existance des méthodes correspondants à chacune des opérations homonymes définies pour des sous-classes différentes. Par exemple, le compilateur est capable de vérifier qu'il existe effectivement une méthode `decompile` ou `analyse` dans chacune des sous-classes de `noeud` (cf. § 7.2.4). Ce gain en sécurité est atteint en réduisant la souplesse d'utilisation du langage. En particulier, pour passer l'étape de compilation, le programme doit comporter la définition de toutes les méthodes susceptibles d'être invoquées. Par exemple, il ne serait pas possible de compiler l'algorithme de décompilation tant que toutes les méthodes `decompile` ne sont pas définies dans toutes les sous-classes de `noeud`. En phase de tests unitaires ou pour le prototypage, une vérification aussi stricte n'est pas nécessaire et se révèle quelque peu contraignante. La liberté complète offerte par Smalltalk permet de tester directement les différentes méthodes de décompilation. En outre, le mécanisme de calcul dynamique du sélecteur (cf. § 7.3) permet une forte paramétrisation des algorithmes (cf. § 6.5.4 et § 4.1.3). Cette souplesse ne va pas sans inconvénients et, en particulier, la *vérification* des programmes Smalltalk ne peut se faire que par l'expérimentation.

8.4.2 Production de logiciels

En ce qui concerne le prototypage ou la conception de logiciels ne nécessitant pas un haut niveau de sécurité, un langage comme Smalltalk est bien adapté. L'utilisation de cet outil peut à notre avis, modifier ou même réduire le cycle de vie habituellement associé à la conception de logiciel. La puissance de l'outil permet souvent, parallèlement à l'établissement du cahier des charges, de réaliser une maquette du logiciel. La maquette peut progressivement être étendue et servir de base de discussion avec l'utilisateur. Dans la phase suivante, un prototype du logiciel peut à son tour être réalisé. Ce prototype peut lui aussi être soumis à l'utilisateur afin qu'il confirme les fonctionnalités du produit. Enfin, ce prototype peut servir à l'implantation du produit final. Pour cette dernière phase, deux voies sont envisageables, selon que le niveau de sécurité impose l'utilisation d'un langage plus *rigoureux* ou non. Dans le cas où la sécurité d'exploitation est un critère essentiel, une traduction à l'aide d'un langage comme Eiffel est souhaitable, le respect des interfaces des différents modules utilisés pouvant être contrôlé par le compilateur[5]. Dans les autres cas, le passage au produit final peut simplement consister à compléter et à affiner les différentes fonctionnalités du prototype.

8.5 leloup

Les méandres du projet GÉPI nous ont amené progressivement à la définition du langage leloup qui s'ajoute à la kyrielle des langages à objets au dessus de LISP[6]. Tout comme le manipulateur de documents (cf. § 8.1) et le langage de description de langages (cf. § 8.2), la définition de leloup s'est modifiée de prototypes en prototypes. L'implantation de leloup nous a permis de mieux comprendre les mécanisme objets, en les implantant nous-même, à la demande.

8.5.1 Héritage multiple

5. En outre, Eiffel étant un langage compilé, on peut espérer quelques gains en efficacité. Lorsque ce gain est primordial, on peut aussi envisager la traduction du programme Smalltalk en C ou en C++ [Stroustrup 86] [Koenig 88] ou encore Objective-C [Cox 83a]. Pour ce dernier langage, il existe d'ailleurs un outil, Producer [Cox 87], de traduction automatique. Quoiqu'il en soit, traduction ou pas, la complexité des algorithmes mis en œuvre reste la même qu'on utilise ou non un langage à objets.

6. Sans se cantonner aux seuls langages de classes, on peut citer : FRL [Roberts 77], Flavors [Moon 80], Act 1 [Lieberman 81], Ceyx [Hullot 83b], LOOPS [Bobrow 83a], MERING [Ferber 83], Formes [Serpette 84], KEE [Kehler 84], ABCL1 [Yonezawa 86], KRS [Marcke 87], YAFOOL [Ducournau 87b], Shirka [Rechenmann 88], etc.
Parmi les tendances actuelles, signalons la norme CLOS (Common Lisp Object System) qui généralise la notion de receveur à tous les arguments d'une méthode [Bobrow 87] [DeMichiel 87]. Il est difficile de dire si cette généralisation apporte un bénéfice certain par rapport à une approche Smalltalk. L'essai des multi-méthodes CLOS sur un problème grandeur nature est aussi un de nos prochains axes de travail.

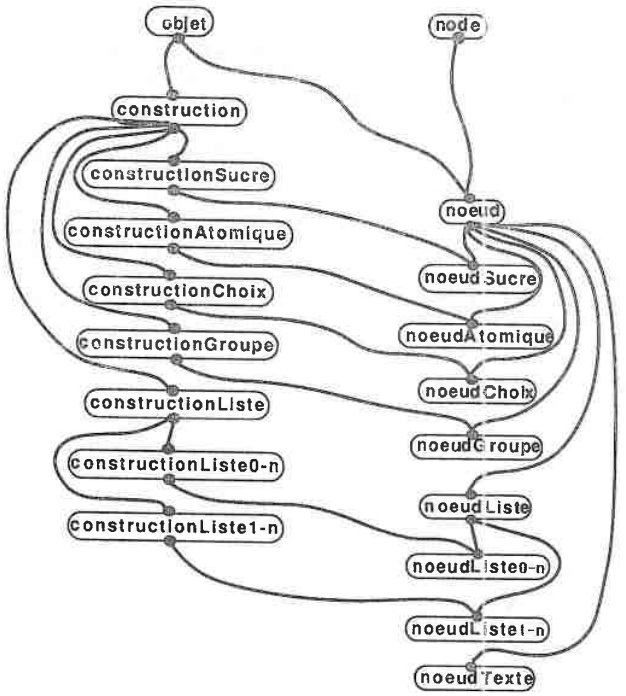


Figure 8.1. Réutilisation par héritage des propriétés des classes associées aux constructions par les classes associées aux nœuds (solution abandonnée).

L'héritage multiple (cf. § 5.2) a été ajouté pour réutiliser dans les classes associées aux nœuds les propriétés des classes associées aux constructions (cf. Fig. 8.1). Cette réutilisation par héritage a été abandonnée dans la suite pour une réutilisation par composition, la classe nœud possédant une variable `construction` (cf. § 6.4.2). Dans ce cas particulier la solution utilisant l'héritage multiple a été abandonnée pour plusieurs raisons. D'une part elle provoque une duplication d'informations, chaque instance d'une classe de nœuds possédant également toutes les variables d'une instance associée à une classe de constructions. Cet héritage massif de variables au niveau des classes de nœuds[7] rend pénible le codage des ins-

7. De 5 variables pour la classe `noeudAtomique` j'usqu'à 11 variables pour la classe `noeudListe1-n` !

tanciations en allongant la liste des variables à initialiser. D'autre part, d'un point de vue conceptuel, il n'y a pas lieu de dupliquer plusieurs fois dans l'ensemble des objets manipulés le représentant d'une construction particulière. Pour un langage donné, il est normal de ne disposer que d'une seule construction pour représenter la boucle "tantQue" par exemple.

Finalement, dans le graphe d'héritage du projet GÉPI, un seul lien d'héritage multiple subsiste et sert à réutiliser la classe **node**, prédéfinie (cf. Fig. 6.10 page 149). Ainsi, pour un langage donné, chaque construction est représentée par une unique instance. Si nous n'avons pas, pour l'implantation du manipulateur de documents, éprouvé un véritable besoin d'héritage multiple, il ne faut pas en conclure que cet aspect d'un langage à objets est inutile [Borning 82b] [Cardelli 84] [Dugerdil 86] [Hendler 86] [Stroustrup 87a] [Ducournau 87a] [Touretzky 87] [Chouraqui 88].

8.5.2 Le mécanisme d'exceptions

Dans certains cas, il y a lieu d'utiliser le mécanisme d'exceptions de leloup (cf. § 5.2.6) :

1. utilisation d'une méthode *mal placée* en regard de la liste de priorité,
2. utilisation d'une méthode absente de la hiérarchie d'héritage d'une classe,
3. ou, simplement pour renommer une méthode héritée[8] ou non.

Selon nous, il y a lieu de s'inquiéter lorsque le mécanisme d'exceptions n'est plus utilisé de façon exceptionnelle ! A l'usage, il s'avère souvent que le découpage ou l'agencement des classes est inapproprié. Bien qu'il soit difficile de justifier cette affirmation, un argument plaide en notre faveur : le mécanisme d'exceptions est, dans une certaine mesure, un retour à l'appel de procédure (surtout dans le cas numéro 2). Fort heureusement, il ne reste qu'une seule exception dans l'implantation actuelle du manipulateur de documents.

8.5.3 Les réflexes

Pour éviter la définition manuelle de méthodes d'accès, le droit en lecture écriture aux variables est implicite en leloup (cf. § 5.1.3). Les réflexes (cf. § 5.3) permettent de contrebalancer cette permissivité qui peut être gênante dans certains cas (cf. page 141 et page 145 et page 148). Un réflexe permet en outre de localiser l'ensemble des actions à entreprendre en cas d'accès à une variable. On peut par exemple effectuer des mesures de fréquence d'accès à une variable par une simple modification d'un réflexe associé à cette variable (cf. page 127). Il est difficile d'avoir la même souplesse en Smalltalk, l'accès aux variables à l'intérieur d'une

8. Lorsque la méthode est héritée, on peut aussi la renomer par masquage.

classe n'étant pas effectué par envoi de message mais directement pris en compte par le compilateur. Ainsi, la mesure de l'accès à une variable ne peut s'effectuer simplement que si l'on a pris la précaution de n'accéder à cette variable que par l'intermédiaire d'une méthode spécifiquement écrite.

8.5.4 Lisp et leloup

Si le mélange LISP et programmation objet font *bon ménage*, il nous est arrivé d'avoir à faire face à des problèmes liés à la coexistence de deux styles de programmation[9]. Par exemple, dans les débuts du projet, par peur d'inefficacité, il nous semblait évident que telle ou telle partie devait être programmée en lisp *pur*. Ce faisant, la partie en question, était alors *hackée* de belle façon. Le problème de ces morceaux de code, écrits en lisp *pur*, est qu'ils sont généralement dépendants de l'implantation de la structure de données qu'ils manipulent, au même titre qu'une procédure Pascal est dépendante de la structure de ses arguments. Par suite, la modification de l'implantation d'une classe dont certaines parties sont *câblées* passe généralement par la remise en cause de ces parties. Fatigués de recommencer sans cesse certaines parties de code, nous avons abandonné cette technique pour le *tout objet*.

Si les caractéristiques de leloup sont quelques peu différentes de Smalltalk en ce qui concerne l'héritage multiple par exemple, la programmation en leloup ne diffère pas fondamentalement de la programmation en Smalltalk. Le code du manipulateur de documents pourrait d'ailleurs être traduit sans efforts particulier en Smalltalk.

9. Les remarques qui suivent valent aussi pour les langages ou tous les concepts ne sont pas unifiés comme C++ par exemple.

9
Décompilation des nœuds "sucre"

```
(demethod
   (noeudSucre decompile) (self collecteur positionOrigine gepi)
   (let ((fils (send 'fils self))
         (constructionsInvisibles (send 'constructionsInvisibles self))
         (zoneInterne (send 'zoneInterne self)))
     ;; (1) Alimentation du collecteur :
     (send 'decompile
           (send 'construction self)
           collecteur
           fils
           constructionsInvisibles
           zoneInterne
           positionOrigine
           gepi))
   ;; (2) Mise à jour de la zone d'influence dans le texte :
   (send 'calculeZone self positionOrigine collecteur))
```

Figure 9.1. La méthode `decompile` de la classe `noeudSucre`.

La décompilation des nœuds *sucre*, comme dans le cas des nœuds *liste* est réalisée à l'aide de deux méthodes de sélecteur `decompile`. La méthode `decompile` de la classe `noeudSucre` (cf. Fig. 9.1) supervise les opérations. La méthode `decompile` définie dans la classe `constructionSucre` (cf Fig. 9.2) est chargée de la décompilation des éventuels fils de la construction *sucre* décompilée.

Les méthodes qui traitent les nœuds *groupe* sont construites sur le même principe. L'unique différence consistant à prendre en compte d'éventuelles parties optionnelles.

```
(demethod (constructionSucre decompile)
  (self collecteur fils cstrInvisibles zoneInterne positionOrigine gepi)
  (let ((sucre (send 'sucre self))
        (prettyFlag 'paragraphage))
    (while sucre
      (cond
        ((stringp (car sucre)) ;; Un terminal :
           (unless (eq prettyFlag 'paragraphage)
               (send 'actionParDefaut collecteur positionOrigine))
           (setq prettyFlag 'terminal)
           (send 'ajoute collecteur (nextl sucre)))
        ((symbolp (car sucre)) ;; Une construction :
           (unless (eq prettyFlag 'paragraphage)
               (send 'actionParDefaut collecteur positionOrigine))
           (setq prettyFlag 'construction)
           (nextl sucre)
           (if (send 'constructionSucre? (car cstrInvisibles))
               (send 'decompile
                     (car cstrInvisibles)
                     collecteur
                     fils
                     (cdr cstrInvisibles)
                     zoneInterne
                     (send 'position collecteur)
                     gepi)
             (let ((positionInterne (send 'position collecteur)))
               (if fils
                   (send 'decompile
                         (car cstrInvisibles)
                         collecteur
                         fils
                         positionInterne
                         gepi)
                 (send 'ajoute
                       collecteur
                       (send 'affichage (car cstrInvisibles)))
                 (send 'calculeZone zoneInterne positionInterne
                       collecteur)))))
        (t ;; Une action de paragraphage :
           (setq prettyFlag 'paragraphage)
           (send 'action collecteur (nextl sucre) positionOrigine))))))
```

Figure 9.2. La méthode decompile de la classe noeudSucre.

10
Autodescription

Cette partie contient l'autodescription du langage de description de langages. Cette dernière n'est pas compliquée mais, d'une certaine manière, est fatigante à regarder tout comme un dessin récursif d'Escher [Hofstadter 85].

```
( langage ldl
  ( terminaux "abreviations" "actionParDefaut" "bruits" "constructions"
              "choix" "extern" "groupe" "indentationParDefaut" "langage"
              "paragraphage" "sucre" "terminaux" "terminauxSeparateurs")
( terminauxSeparateurs "(" ")")
( abreviations
    (chaine """ ( 0-n ( ou "\\" "\"" ( sauf """))) """)
    (chiffre ( tranche "0" "9"))
    (nom ( 0-n ( sauf caractereSpecial)))
    (nombre ( 1-n chiffre))
    (caractere ( ou chaine nombre ("#\" nom)))
    (commentaire ";" ( sauf #\cr))
    (special ( ou #\cr #\lf #\sp #\tab)))
( bruits
    ( atomique ( 0-n ( ou special commentaire))))
( paragraphage
    ( actionParDefaut (b0))
    ( indentationParDefaut 3))
( constructions
  (langage
      ( groupe "(" "langage" (b1) nom (i+)
               "(" "terminaux" (i+ 6) terminaux ")" (i+)
               "(" "terminauxSeparateurs" (i+ 6) terminaux ")" (i+)
               "(" "abreviations" (i+ 6) abreviations ")" (i+)
               "(" "paragraphage" (i+ 6)
               "(" "indentationParDefaut" (b1) nombre ")" (i+ 6)
               "(" "actionParDefaut" (b1) action ")" ")" (i+)
               "(" "bruits" (b1) corpsDeConstruction ")" (i+)
               "(" "constructions" (i+ 6) constructions ")" ")"))
  (abreviation
      ( groupe "(" nom (b1) expr ")"))
```

```
(abreviations
    ( liste0-n abreviation)
    ( indent ()() ()(i0) ()()))
(action
    ( atomique "(" nom ( 0-n ( ou chaine " " nombre)) ")"))
(atome
    ( atomique ( ou chaine nom)))
(chaine
    ( atomique chaine))
(clauseExtern
    ( sucre "(" "extern" (b1) chaine ")"))
(clauseIndent
    ( sucre "(" "indent" (b1) expr ")"))
(clauseTete
    ( sucre "(" "tete" (b1) chaine ")"))
(clauseSeparateur
    ( sucre "(" "separateur" (b1) chaine ")"))
(clauseQueue
    ( sucre "(" "queue" (b1) chaine ")"))
(construction
    ( groupe "(" nom (b1) corpsDeConstruction ")"))
(constructionAtomique
    ( groupe ( 0-1 clauseExtern) (i0) "(" "atomique" (b1) expr ")"))
(constructionChoix
    ( groupe ( 0-1 clauseExtern) (i0) "(" "choix" (b1) expr ")"))
(constructionGroupe
    ( groupe ( 0-1 clauseExtern) (i0) "(" "groupe" (b1) expr ")"))
(constructionListe
    ( groupe ( 0-1 clauseExtern) (i+) "(" liste?-n nom ")" (i+)
             ( 0-1 clauseTete) (i+) ( 0-1 clauseSeparateur) (i+)
             ( 0-1 clauseSeparateur) (i+) ( 0-1 clauseIndent)))
(constructions
    ( liste1-n construction)
    ( indent ()() ()(i0) ()() ()()))
(constructionSucre
    ( groupe ( 0-1 clauseExtern) (i0) "(" "sucre" (b1) expr ")"))
(corpsDeConstruction
    ( choix constructionSucre constructionGroupe constructionChoix
          constructionAtomique constructionListe))
(expr
    ( liste1-n s-expression))
(liste
    ( liste1-n s-expression)
    ( tete "("
    ( queue ")"))
(listeDactions
    ( liste0-n action))
(nom
```



```
    ( atomique nom))
(nombre
    ( atomique nombre))
(s-expression
    ( choix liste atome))
(terminaux
    ( liste0-n chaine)
    ( indent ()() ()(i0) ()()))
(liste?-n
    ( atomique ( ou "liste0-n" "liste1-n")))))
```

11
Description du langage Ada

La description suivante du langage Ada [Ada 83] a été établie à l'aide des diagrammes de syntaxe donnés dans [Booch 87, page 371–407]. Tous les noms de diagrammes ont leur équivalent dans la description, généralement sous la forme d'un nom de construction, certains diagrammes étant traduits à l'aide d'une abréviation comme `exponent` ou `extended_digit` par exemple.

Il n'est pas toujours possible de traduire un diagramme à l'aide d'une seule construction. Les noms de constructions rajoutés sont repérés par le commentaire "; ***". Dans cette description, les constructions *sucre* se révèlent très utiles pour conserver l'information sémantique contenue dans les diagrammes. Par exemple les constructions `VARIABLE_NAME`, `FUNCTION_NAME`, `EXCEPTION_NAME` ou encore `COMPONENT_NAME` sont toutes des constructions *sucre* dérivant vers la construction `NAME`.

```
( langage Ada
  ( terminaux "abort" "access" "accept" "and" "array" "at" "begin" "body"
              "case" "constant" "delay" "digits" "do" "declare" "else" "elsif" "end"
              "entry" "exception" "exit" "for" "function" "generic" "goto" "if" "is"
              "mod" "new" "new" "of" "or" "package" "pragma" "procedure" "range"
              "renames" "separate" "select" "then" "type" "task" "use" "with" "when"
              "xor")
  ( terminauxSeparateurs "," "." ";" "(" ")" "|" ":=" "=>")
  ( abreviations
    (base integer)
    (base_integer (extended_digit ( 0-n ( ou ("_" extended_digit) extended_digit))))
    (digit ( tranche "0" "9"))
    (exponent ("E" ( 0-1 ( ou "+" "-")) integer))
    (extended_digit ( ou letter digit))
    (graphic_character ( ou ( sauf "'") ( sauf """")))
    (integer (digit ( 0-n ( 0-1 "_") digit)))
    (identifier (letter ( 0-n ( ou "_" extended_digit))))
    (label_name identifier)
    (letter ( ou ( tranche "a" "z") ( tranche "A" "Z")))
    (simple_name identifier)
    (comment "--" ( sauf #\cr))
    (special_character ( ou #\cr #\lf #\sp #\tab))
    (bruits ( 0-n ( ou special_character comment))))
( bruits
    ( atomique bruits))
```

```
( paragraphage
    ( actionParDefaut (b1))
    ( indentationParDefaut 3))
( constructions ;; ------------------------------
(COMPILATION
    ( liste0-n COMPILATION_UNIT)
    ( indent ()() ()(1 3) ()() ()()))
(ABORT_STATEMENT
    ( liste1-n TASK_NAME)
    ( tete "abort")
    ( separateur ",")
    ( queue ";"))
(ACCEPT_ALTERNATIVE
    ( groupe ACCEPT_STATEMENT ( 0-1 STATEMENTS)))
(ACCEPT_STATEMENT
    ( groupe "accept" ENTRY_SIMPLE_NAME
            ( 0-1 PARENT_ENTRY_INDEX)
            ( 0-1 FORMAL_PART)
            ( 0-1 ACCEPT_STATEMENT_DO_PART) ";"))
(ACCEPT_STATEMENT_DO_PART ; ***
       ( groupe "do" STATEMENTS "end" ( 0-1 ENTRY_SIMPLE_NAME)))
(ACCESS_TYPE_DEFINITION
    ( sucre "access" SUBTYPE_INDICATION))
(ACTUAL_PARAMETER
    ( choix EXPRESSION EXPLICIT_CONVERSION VARIABLE_NAME))
(ACTUAL_PARAMETER_PART
    ( liste1-n PARAMETER_ASSOCIATION)
    ( tete "(")
    ( separateur ",")
    ( queue ")"))
(ADDRESS_CLAUSE
    ( groupe "for" SIMPLE_NAME "use" "at" SIMPLE_EXPRESION ";"))
(AGGREGATE
    ( liste1-n COMPONANT_ASSOCIATION)
    ( tete "(")
    ( separateur ",")
    ( queue ")"))
(ALIGNMENT_CLAUSE
    ( groupe "at" "mod" STATIC_SIMPLE_EXPRESSION ";"))
(all
    ( atomique "all"))
(ALLOCATOR
    ( sucre "new" QUALIFIED_EXPRESSION_OR_SUBTYPE_INDICATION))
(AND_RELATION_LIST ; ***
    ( groupe "and" RELATION))
(AND_THEN_RELATION_LIST ; ***
    ( groupe "and" "then" RELATION))
(ARGUMENT_ASSOCIATION
    ( groupe ( 0-1 ARGUMENT_IDENTIFIER_PART) EXPRESSION_OR_NAME))
(ARGUMENT_ASSOCIATION_LIST ; ***
    ( liste1-n ARGUMENT_ASSOCIATION)
    ( tete "(")
    ( separateur ",")
    ( queue ")"))
```



```
(ARGUMENT_IDENTIFIER
    ( sucre IDENTIFIER))
(ARGUMENT_IDENTIFIER_PART ; ***
    ( sucre ARGUMENT_IDENTIFIER "=>"))
(ARRAY_OR_SUBTYPE
    ( choix CONSTRAINED_ARRAY_DEFINITION SUBTYPE_INDICATION))
(ARRAY_TYPE_DEFINITION
    ( choix CONSTRAINED_ARRAY_DEFINITION UNCONSTRAINED_ARRAY_DEFINITION))
(ASSIGNMENT_STATEMENT
    ( groupe VARIABLE_NAME ":=" EXPRESSION ";"))
(ATTRIBUTE
    ( groupe PREFIX "'" ATTRIBUTE_DESIGNATOR))
(ATTRIBUTE_DESIGNATOR
    ( groupe SIMPLE_NAME ( 0-1 PARENT_UNIVERSAL_STATIC_EXPRESSION)))
(BASE
    ( atomique base))
(BASED_INTEGER
    ( atomique based_integer))
(BASED_LITERAL
    ( atomique (base "#" based_integer ( 0-1 "." based_integer) "#" ( 0-1 exponent))))
(BASIC_DECLARATION
    ( choix DEFERED_CONSTANT_DECLARATION EXEPTION_DECLARATION
            GENERIC_DECLARATION GENERIC_INSTANCIATION NUMBER_DECLARATION
            OBJECT_DECLARATION PACKAGE_DECLARATION RENAMING_DECLARATION
            SUBPROGRAM_DECLARATION SUBTYPE_DECLARATION TASK_DECLARATION
            TYPE_DECLARATION))
(BASIC_DECLARATIVE_ITEM
    ( choix BASIC_DECLARATION REPRESENTATION_CLAUSE USE_CLAUSE))
(BASIC_DECLARATIVE_ITEM_LIST ; ***
    ( liste0-n BASIC_DECLARATIVE_ITEM))
(BINARY_ADDING_OPERATOR
    ( atomique ( ou "+" "-" "&")))
(BLOCK_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(BLOCK_SIMPLE_NAME_PART ; ***
    ( sucre BLOCK_SIMPLE_NAME ":"))
(BLOCK_STATEMENT
    ( groupe ( 0-1 BLOCK_SIMPLE_NAME_PART)
             ( 0-1 DECLARE_PART)
             "begin" STATEMENTS
             ( 0-1 EXCEPTION_LIST)
             "end" ( 0-1 BLOCK_SIMPLE_NAME) ";"))
(BODY
    ( choix BODY_STUB PROPER_BODY))
(BODY_STUB
    ( groupe SUBPROGRAM_OR_PACKAGE_OR_TASK "is" "separate" ";"))
(BOOLEAN_EXPRESSION
    ( sucre EXPRESSION))
(CASE_STATEMENT
    ( groupe "case" EXPRESSION "is" (i+) CASE_STATEMENT_LIST "end" "case" ";"))
(CASE_STATEMENT_ALTERNATIVE
    ( groupe WHEN_PART STATEMENTS))
(CASE_STATEMENT_LIST ; ***
    ( liste1-n CASE_STATEMENT_ALTERNATIVE))
```

```
(CHARACTER_LITERAL
    ( atomique ("'" graphic_character "'")))
(CHOICE
    ( choix COMPONENT_SIMPLE_NAME DISCRETE_RANGE SIMPLE_EXPRESSION others))
(CHOICE_LIST ; ***
    ( liste1-n CHOICE)
    ( separateur "|")
    ( queue "=>"))
(CODE_STATEMENT
    ( groupe TYPE_MARK "'" RECORD_AGGREGATE ";"))
(COMPILATION_UNIT
    ( groupe CONTEXT_CLAUSE (i0) UNIT))
(COMPONENT_ASSOCIATION
    ( groupe ( 0-1 CHOICE_LIST) EXPRESSION))
(COMPONANT_CLAUSE
    ( groupe COMPONENT_NAME "at" STATIC_SIMPLE_EXPRESSION range STATIC_RANGE ";"))
(COMPONENT_CLAUSE_LIST ; ***
    ( liste0-n COMPONENT_CLAUSE))
(COMPONENT_DECLARATION
    ( groupe IDENTIFIER_LIST ":" COMPONENT_SUBTYPE_INDICATION
            ( 0-1 DEFAULT_VALUE) ";"))
(COMPONENT_DECLARATION_LIST ; ***
    ( groupe COMPONENT_DECLARATION_LIST1-N ( 0-1 VARIANT_PART)))
(COMPONENT_DECLARATION_LIST1-N ; ***
    ( liste1-n COMPONENT_DECLARATION))
(COMPONENT_LIST
    ( choix null COMPONENT_DECLARATION_LIST VARIANT_PART))
(COMPONENT_NAME
    ( sucre NAME))
(COMPONENT_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(COMPONENT_SUBTYPE_DEFINITION
    ( sucre SUBTYPE_INDICATION))
(COMPONENT_SUBTYPE_INDICATION
    ( sucre SUBTYPE_INDICATION))
(COMPOUND_STATEMENT
    ( choix ACCEPT_STATEMENT BLOCK_STATEMENT CASE_STATEMENT IF_STATEMENT
          LOOP_STATEMENT SELECT_STATEMENT))
(CONDITION
    ( sucre BOOLEAN_EXPRESSION))
(CONDITIONAL_ENTRY_CALL
    ( groupe "select" (i+) ENTRY_CALL_STATEMENT ( 0-1 STATEMENTS)
            (i0) "else" (i+) STATEMENTS (i-) "end" "select" ";"))
(constant
    ( atomique "constant"))
(CONSTRAINED_ARRAY_DEFINITION
    ( groupe "array" INDEX_CONSTRAINT "of" COMPONENT_SUBTYPE_INDICATION))
(CONSTRAINT
    ( choix DISCRIMINANT_CONSTRAINT FIXED_POINT_CONSTRAINT
          FLOATING_POINT_CONSTRAINT INDEX_CONSTRAINT RANGE_CONSTRAINT))
(CONTEXT_CLAUSE
    ( liste0-n WITH_USE)
    ( indent ()() ()(i0) ()() ()()))
(DECIMAL_LITERAL
```



```
    ( atomique integer ( 0-1 "." integer) ( 0-1 exponent)))
(DECLARATIVE_PART
    ( groupe BASIC_DECLARATIVE_ITEM_LIST LATER_DECLARATIVE_ITEM_LIST))
(DECLARE_PART ; ***
    ( sucre "declare" DECLARATIVE_PART))
(DEFAULT_VALUE ; ***
    ( sucre ":=" EXPRESSION))
(DEFERED_CONSTANT_DECLARATION
    ( groupe IDENTIFIER_LIST ":" "constant" TYPE_MARK ";"))
(DELAY_ALTERNATIVE
    ( groupe DELAY_STATEMENT ( 0-1 STATEMENTS)))
(DELAY_STATEMENT
    ( sucre "delay" SIMPLE_EXPRESSION ";"))
(DELTA
    ( extern "delta <>")
    ( atomique ("delta" bruits "<>")))
(DERIVED_TYPE_DEFINITION
    ( sucre "new" SUBTYPE_INDICATION))
(DESIGNATOR
    ( choix IDENTIFIER OPERATOR_SYMBOL))
(DIGITS ; ***
    ( extern "digits <>")
    ( atomique ("digits" bruits "<>")))
(DISCRETE_RANGE
    ( choix DISCRETE_SUBTYPE_INDICATION RANGE))
(DISCRETE_SUBTYPE_INDICATION
    ( sucre SUBTYPE_INDICATION))
(DISCRIMINANT_ASSOCIATION
    ( groupe ( 0-1 DISCRIMINANT_SIMPLE_NAME_LIST) EXPRESSION))
(DISCRIMINANT_CONSTRAINT
    ( liste1-n DISCRIMINANT_ASSOCIATION)
    ( tete "(")
    ( separateur ",")
    ( queue ")"))
(DISCRIMINANT_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(DISCRIMINANT_SIMPLE_NAME_LIST ; ***
    ( liste1-n DISCRIMINANT_SIMPLE_NAME)
    ( separateur "|")
    ( queue "=>"))
(DISCRIMINANT_SPECIFICATION
    ( groupe IDENTIFIER_LIST ":" TYPE_MARK ( 0-1 DEFAULT_VALUE)))
(DISCRIMINANT_PART
    ( liste1-n DISCRIMINANT_SPECIFICATION)
    ( tete "(")
    ( separateur ";")
    ( queue ")"))
(ELSE_PART ; ***
    ( sucre "else" (i+) STATEMENTS (i-)))
(ELSIF ; ***
    ( groupe "elsif" CONDITION "then" (i+) STATEMENTS (i-)))
(ELSIF_LIST ; ***
    ( liste0-n ELSIF))
(ENTRY_CALL_STATEMENT
```

```
    ( groupe ENTRY_NAME ( 0-1 ACTUAL_PARAMETER_PART) ";"))
(ENTRY_DECLARATION
    ( groupe "entry" IDENTIFIER ( 0-1 PARENT_DISCRETE_RANGE)
             ( 0-1 FORMAL_PART) ";"))
(ENTRY_DECLARATION_LIST ; ***
    ( liste0-n ENTRY_DECLARATION))
(ENTRY_INDEX
    ( sucre EXPRESSION))
(ENTRY_NAME
    ( sucre NAME))
(ENTRY_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(ENUMERATION_LITERAL
    ( choix CHARACTER_LITERAL IDENTIFIER))
(ENUMERATION_LITERAL_SPECIFICATION
    ( sucre ENUMERATION_LITERAL))
(ENUMERATION_REPRESENTATION_CLAUSE
    ( groupe "for" TYPE_SIMPLE_NAME "use" AGGREGATE ";"))
(ENUMERATION_TYPE_DEFINITION
    ( liste1-n ENUMERATION_LITERAL_SPECIFICATION)
    ( tete "(")
    ( separateur ",")
    ( queue ")"))
(EXEPTION_CHOICE
    ( choix EXCEPTION_NAME others))
(EXCEPTION_CHOICE_LIST ; ***
    ( liste1-n EXCEPTION_CHOICE)
    ( tete "when")
    ( separateur "|")
    ( queue "=>"))
(EXCEPTION_DECLARATION
    ( groupe IDENTIFIER_LIST ":" "exception" ";"))
(EXCEPTION_HANDLER
    ( groupe EXCEPTION_CHOICE_LIST STATEMENTS))
(EXCEPTION_NAME
    ( sucre NAME))
(EXCEPTION_PART ; ***
    ( liste1-n EXCEPTION_HANDLER)
    ( tete "exception"))
(EXCEPTION_LIST ; ***
    ( liste1-n EXCEPTION_HANDLER)
    ( tete "exception"))
(EXIT_STATEMENT
    ( groupe "exit" ( 0-1 LOOP_NAME) ( 0-1 WHEN_CONDITION) ";"))
(EXPLICIT_CONVERSION ; ***
    ( groupe TYPE_MARK "(" VARIABLE_NAME ")"))
(EXPRESSION
    ( groupe RELATION ( 0-1 EXPRESSION_AUX)))
(EXPRESSION_AUX ; ***
    ( choix AND_RELATION_LIST OR_RELATION_LIST XOR_RELATION_LIST
            AND_THEN_RELATION_LIST OR_ELSE_RELATION_LIST))
(EXPRESSION_LIST ; ***
    ( liste1-n EXPRESSION)
    ( tete "(")
```



```
    ( separateur ",")
    ( queue ")"))
(EXPRESSION_OR_NAME ; ***
    ( choix EXPRESSION NAME))
(FACTOR
    ( groupe ( 0-1 PRIMARY) ( 0-1 HIGHEST_PRECEDENCE_OPERATOR_PRIMARY)))
(FIXED_ACCURACY_DEFINITION
    ( sucre "delta" STATIC_SIMPLE_EXPRESSION))
(FIXED_POINT_CONSTRAINT
    ( groupe FIXED_ACCURACY_DEFINITION ( 0-1 RANGE_CONSTRAINT)))
(FLOATING_ACCURACY_DEFINITION
    ( sucre "digits" STATIC_SIMPLE_EXPRESSION))
(FLOATING_POINT_CONSTRAINT
    ( groupe FLOATING_ACCURACY_DEFINITION ( 0-1 RANGE_CONSTRAINT)))
(FOR_SCHEME ; ***
    ( sucre "for" LOOP_PARAMETER_SPECIFICATION))
(FORMAL_PARAMETER
    ( sucre PARAMETER_SIMPLE_NAME))
(FORMAL_PARAMETER_NAME ; ***
    ( sucre FORMAL_PARAMETER "=>"))
(FORMAL_PART
    ( liste1-n PARAMETER_SPECIFICATION)
    ( tete "(")
    ( separateur ";")
    ( queue ")"))
(FULL_TYPE_DECLARATION
    ( groupe "type" IDENTIFIER ( 0-1 DISCRIMINANT_PART) "is"
             TYPE_DEFINITION ";"))
(FUNCTION_CALL
    ( groupe FUNCTION_NAME ( 0-1 ACTUAL_PARAMETER_PART)))
(FUNCTION_INSTANCIATION ; ***
    ( groupe "function" DESIGNATOR "is" "new" GENERIC_FUNCTION_NAME))
(FUNCTION_NAME
    ( sucre NAME))
(FUNCTION_SPECIFICATION
    ( groupe "function" DESIGNATOR ( 0-1 FORMAL_PART) "return" TYPE_MARK))
(GENERIC_ACTUAL_PARAMETER
    ( choix ENTRY_NAME EXPRESSION SUBPROGRAM_NAME TYPE_MARK VARIABLE_NAME))
(GENERIC_ACTUAL_PART
    ( liste1-n GENERIC_ASSOCIATION)
    ( tete "(")
    ( separateur ",")
    ( queue ")"))
(GENERIC_ASSOCIATION
    ( groupe ( 0-1 GENERIC_FORMAL_PARAMETER_PART) GENERIC_ACTUAL_PARAMETER))
(GENERIC_FORMAL_PARAMETER_PART ; ***
    ( sucre GENERIC_FORMAL_PARAMETER "=>"))
(GENERIC_DECLARATION
    ( sucre GENERIC_SPECIFICATION ";"))
(GENERIC_FORMAL_PARAMETER
    ( choix OPERATOR_SYMBOL PARAMETER_SIMPLE_NAME))
(GENERIC_FORMAL_PART
    ( liste0-n GENERIC_PARAMETER_DECLARATION)
    ( tete "generic"))
```

```
(GENERIC_INSTANTIATION
    ( groupe GENERIC_INSTANCIATION_AUX ( 0-1 GENERIC_ACTUAL_PART) ";"))
(GENERIC_INSTANCIATION_AUX ; ***
    ( choix FUNCTION_INSTANCIATION PACKAGE_INSTANCIATION
          PROCEDURE_INSTANCIATION))
(GENERIC_PARAMETER_DECLARATION
    ( choix GENERIC_IDENTIFIER_LIST GENERIC_PRIVATE_TYPE_DECLARATION
          GENERIC_TYPE GENERIC_WITH))
(GENERIC_IDENTIFIER_LIST ; ***
    ( groupe IDENTIFIER_LIST ":" ( 0-1 MODE) TYPE_MARK ( 0-1 DEFAULT_VALUE) ";"))
(GENERIC_PRIVATE_TYPE_DECLARATION ; ***
    ( groupe PRIVATE_TYPE_DECLARATION ";"))
(GENERIC_TYPE ; ***
    ( groupe "type" IDENTIFIER "is" GENERIC_TYPE_DEFINITION ";"))
(GENERIC_WITH ; ***
    ( groupe "with" SUBPROGRAM_SPECIFICATION ( 0-1 IS_PART) ";"))
(GENERIC_SPECIFICATION
    ( groupe GENERIC_FORMAL_PART PACKAGE_OR_SUBPROGRAM_SPECIFICATION))
(GENERIC_FUNCTION_NAME
    ( sucre NAME))
(GENERIC_PACKAGE_NAME
    ( sucre NAME))
(GENERIC_PROCEDURE_NAME
    ( sucre NAME))
(GENERIC_TYPE_DEFINITION
    ( choix PARENT_<> DELTA DIGITS RANGE<> ACCESS_TYPE_DEFINITION
          ARRAY_TYPE_DEFINITION))
(GOTO_STATEMENT
    ( sucre "goto" LABEL_NAME ";"))
(HIGHEST_PRECEDENCE_OPERATOR
    ( atomique ( ou "**" "abs" "not")))
(HIGHEST_PRECEDENCE_OPERATOR_PRIMARY ; ***
    ( groupe HIGHEST_PRECEDENCE_OPERATOR PRIMARY))
(IDENTIFIER
    ( atomique identifier))
(IDENTIFIER_LIST
    ( liste1-n IDENTIFIER)
    ( separateur ","))
(IF_STATEMENT
    ( groupe "if" CONDITION "then" (i+)
          STATEMENTS (i-)
          ELSIF_LIST
          ( 0-1 ELSE_PART)
          "end" "if" ";"))
(INCOMPLETE_TYPE_DECLARATION
    ( groupe "type" IDENTIFIER ( 0-1 DISCRIMINANT_PART) ";"))
(INDEX_CONSTRAINT
    ( liste1-n DISCRETE_RANGE)
    ( tete "(")
    ( separateur ",")
    ( queue ")"))
(INDEX_SUBTYPE_DEFINITION
    ( sucre TYPE_MARK "range" "<>"))
(INDEX_SUBTYPE_DEFINITION_LIST ; ***
```



```
    ( liste1-n INDEX_SUBTYPE_DEFINITION)
    ( tete "(")
    ( separateur ",")
    ( queue ")"))
(INDEXED_COMPONENT
    ( groupe PREFIX EXPRESSION_LIST))
(INTEGER_TYPE_DEFINITION
    ( sucre RANGE_CONSTRAINT))
(INSTRUCTION_PART ; ***
    ( groupe "begin" (i+) STATEMENTS (i-) ( 0-1 EXCEPTION_PART)))
(ITERATION_SCHEME
    ( choix FOR_SCHEME WHILE_SCHEME))
(IS_PART ; ***
    ( atomique ("is" bruits ( ou name "<>"))))
(LABEL
    ( atomique "<<" label_name ">>"))
(LABEL_LIST ; ***
    ( liste1-n LABEL))
(LABEL_NAME ; ***
    ( atomique label_name))
(LATER_DECLARATIVE_ITEM
    ( choix BODY GENERIC_DECLARATION GENERIC_INSTANTIATION
          PACKAGE_DECLARATION SUBPROGRAM_DECLARATION
          TASK_DECLARATION USE_CLAUSE))
(LATER_DECLARATIVE_ITEM_LIST ; ***
    ( liste0-n LATER_DECLARATIVE_ITEM))
(LENGTH_CLAUSE
    ( groupe "for" ATTRIBUTE "use" SIMPLE_EXPRESSION ";"))
(LIBRARY_UNIT
    ( choix GENERIC_DECLARATION GENERIC_INSTANTIATION PACKAGE_DECLARATION
          SUBPROGRAM_DECLARATION SUBPROGRAM_BODY))
(LIBRARY_UNIT_BODY
    ( choix PACKAGE_BODY SUBPROGRAM_BODY))
(limited ; ***
    ( atomique "limited"))
(LOGICAL_OPERATOR
    ( atomique ( ou "and" "or" "xor")))
(LOOP_PARAMETER_SPECIFICATION
    ( groupe IDENTIFIER "in" ( 0-1 reverse) DISCRETE_RANGE))
(LOOP_STATEMENT
    ( groupe ( 0-1 LOOP_SIMPLE_NAME) ( 0-1 ITERATION_SCHEME) (i0)
          "loop" (i+) STATEMENTS (i-) "end" "loop"
          ( 0-1 SIMPLE_NAME) ";"))
(LOOP_SIMPLE_NAME ; ***
    ( atomique simple_name bruits ":"))
(MODE
    ( atomique ( 0-1 ( ou "in" "out" ("in" bruits "out")))))
(MULTIPLYING_OPERATOR
    ( atomique ( ou "*" "/" "mod" "rem")))
(NAME
    ( choix ATTRIBUTE CHARACTER_LITERAL INDEXED_COMPONENT OPERATOR_SYMBOL
          SELECTED_COMPONENT SIMPLE_NAME SLICE))
(not
    ( atomique "not"))
```

```
(NULL_STATEMENT
    ( extern "null ;")
    ( atomique "null" bruits ";"))
(NUMBER_DECLARATION
    ( groupe IDENTIFIER_LIST ":" "constant" ":=" UNIVERSAL_STATIC_EXPRESSION ";"))
(NUMERIC_LITERAL
    ( choix BASED_LITERAL DECIMAL_LITERAL))
(null
    ( atomique "null"))
(OBJECT_DECLARATION
    ( groupe IDENTIFIER_LIST ":" ( 0-1 constant) ARRAY_OR_SUBTYPE
            ( 0-1 DEFAULT_VALUE) ";"))
(OBJECT_NAME
    ( sucre NAME))
(OPERATOR_SYMBOL
    ( sucre STRING_LITERAL))
(OR_ELSE_RELATION_LIST ; ***
    ( groupe "or" "else" RELATION))
(OR_PART ; ***
    ( liste0-n SELECT_ALTERNATIVE)
    ( tete "or"))
(OR_RELATION_LIST ; ***
    ( groupe "or" RELATION))
(others
    ( atomique "others"))
(PACKAGE_BODY
    ( groupe PACKAGE_BODY... "is" (i0)
            ( 0-1 DECLARATIVE_PART) (i0) ( 0-1 INSTRUCTION_PART)
            (i0) "end" ( 0-1 PACKAGE_SIMPLE_NAME) ";"))
(PACKAGE_BODY... ; ***
    ( sucre "package" "body" PACKAGE_SIMPLE_NAME))
(PACKAGE_DECLARATION
    ( sucre PACKAGE_SPECIFICATION ";"))
(PACKAGE_INSTANCIATION ; ***
    ( groupe "package" DESIGNATOR "is" "new" GENERIC_PACKAGE_NAME))
(PACKAGE_NAME
    ( sucre NAME))
(PACKAGE_OR_SUBPROGRAM_SPECIFICATION ; ***
    ( choix PACKAGE_SPECIFICATION SUBPROGRAM_SPECIFICATION))
(PACKAGE_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(PACKAGE_SPECIFICATION
    ( groupe "package" IDENTIFIER "is" (i+) BASIC_DECLARATIVE_ITEM_LIST (i-)
            ( 0-1 PRIVATE_PART) (i0) "end" ( 0-1 PACKAGE_SIMPLE_NAME)))
(PARAMETER_ASSOCIATION
    ( groupe ( 0-1 FORMAL_PARAMETER_NAME) ACTUAL_PARAMETER))
(PARAMETER_SPECIFICATION
    ( groupe IDENTIFIER_LIST ":" MODE TYPE_MARK ( 0-1 DEFAULT_VALUE)))
(PARAMETER_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(PRAGMA
    ( groupe "pragma" IDENTIFIER ( 0-1 ARGUMENT_ASSOCIATION_LIST) ";"))
(PARENT_DISCRETE_RANGE ; ***
    ( sucre "(" DISCRETE_RANGE ")"))
```



```
(PARENT_ENTRY_INDEX ; ***
    ( sucre "(" ENTRY_INDEX ")"))
(PARENT_EXPRESSION ; ***
    ( sucre "(" EXPRESSION ")"))
(PARENT_EXPRESSION_OR_AGGREGATE ; ***
    ( choix PARENT_EXPRESSION AGGREGATE))
(PARENT_UNIVERSAL_STATIC_EXPRESSION ; ***
    ( sucre "(" UNIVERSAL_STATIC_EXPRESSION ")"))
(PARENT_<> ; ***
    ( extern "( <> )")
    ( atomique ("(" bruits "<>" bruits ")")))
(PREFIX
    ( choix FUNCTION_CALL NAME))
(PRIMARY
    ( choix AGGREGATE ALLOCATOR FUNCTION_CALL NAME null NUMERIC_LITERAL
            QUALIFIED_EXPRESSION STRING_LITERAL TYPE_CONVERSION
            PARENT_EXPRESSION))
(PRIVATE_PART ; ***
    ( liste0-n BASIC_DECLARATIVE_ITEM)
    ( tete "private"))
(PRIVATE_TYPE_DECLARATION
    ( groupe "type" IDENTIFIER ( 0-1 DISCRIMINANT_PART) "is"
            ( 0-1 limited) "private" ";"))
(PROCEDURE_CALL_STATEMENT
    ( groupe PROCEDURE_NAME ( 0-1 ACTUAL_PARAMETER_PART) ";"))
(PROCEDURE_NAME
    ( sucre NAME))
(PROCEDURE_INSTANCIATION ; ***
    ( groupe "procedure" DESIGNATOR "is" "new" GENERIC_PROCEDURE_NAME))
(PROCEDURE_SPECIFICATION ; ***
    ( groupe "procedure" IDENTIFIER ( 0-1 FORMAL_PART)))
(PROPER_BODY
    ( choix PACKAGE_BODY SUBPROGRAM_BODY TASK_BODY))
(QUALIFIED_EXPRESSION
    ( groupe TYPE_MARK "'" PARENT_EXPRESSION_OR_AGGREGATE))
(QUALIFIED_EXPRESSION_OR_SUBTYPE_INDICATION ; ***
    ( choix QUALIFIED_EXPRESSION SUBTYPE_INDICATION))
(RAISE_STATEMENT
    ( groupe "raise" ( 0-1 EXCEPTION_NAME) ";"))
(RANGE
    ( choix SIMPLE_RANGE RANGE_ATTRIBUTE))
(RANGE_CONSTRAINT
    ( sucre "range" RANGE))
(RANGE_OR_TYPE_MARK ; ***
    ( choix RANGE TYPE_MARK))
(RANGE<> ; ***
    ( extern "range <>")
    ( atomique ("range" bruits "<>")))
(REAL_TYPE_DEFINITION
    ( choix FIXED_POINT_CONSTRAINT FLOATING_POINT_CONSTRAINT))
(RECORD_AGGREGATE
    ( sucre AGGREGATE))
(RECORD_TYPE_DEFINITION
    ( sucre "record" (i0) COMPONENT_LIST (i0) "end" "record"))
```

```
(RECORD_REPRESENTATION_CLAUSE
    ( groupe "for" TYPE_SIMPLE_NAME "use" (i0) "record"
            ( 0-1 ALIGNMENT_CLAUSE)(i0) COMPONENT_CLAUSE_LIST (i0)
            "end" "record" ";"))
(RELATION
    ( groupe SIMPLE_EXRESSION ( 0-1 RELATION_AUX)))
(RELATION_AUX ; ***
    ( choix RELATIONAL_EXPRESSION IN_EXPRESSION))
(RELATIONAL_EXPRESSION
    ( groupe RELATIONAL_OPERATOR SIMPLE_EXPRESSION))
(IN_EXPRESSION
    ( groupe ( 0-1 not) "in" RANGE_OR_TYPE_MARK))
(RELATIONAL_OPERATOR
    ( atomique ( ou "=" "/=" "<" "<=" ">" ">=")))
(RENAMING_DECLARATION
    ( choix RENAMING_IDENTIFIER RENAMING_EXCEPTION RENAMING_PACKAGE
          RENAMING_SUBPROGRAM))
(RENAMING_EXCEPTION
    ( groupe IDENTIFIER ":" "exception" "renames" EXCEPTION_NAME ";"))
(RENAMING_IDENTIFIER ; ***
    ( groupe IDENTIFIER ":" TYPE_MARK "renames" OBJECT_NAME ";"))
(RENAMING_PACKAGE
    ( groupe "package" IDENTIFIER "renames" PACKAGE_NAME ";"))
(RENAMING_SUBPROGRAM
    ( groupe SUBPROGRAM_SPECIFICATION "renames" SUBPROGRAM_OR_ENTRY_NAME))
(REPRESENTATION_CLAUSE
    ( choix ADDRESS_CLAUSE TYPE_REPRESENTATION_CLAUSE))
(REPRESENTATION_LIST ; ***
    ( liste0-n REPRESENTATION_CLAUSE))
(RETURN_STATEMENT
    ( groupe "return" ( 0-1 EXPRESSION) ";"))
(reverse
    ( atomique "reverse"))
(SECONDARY_UNIT
    ( choix LIBRARY_UNIT_BODY SUBUNIT))
(SELECT_ALTERNATIVE
    ( groupe ( 0-1 WHEN_CLAUSE) SELECTIVE_WAIT_PART))
(SELECT_STATEMENT
    ( choix CONDITIONAL_ENTRY_CALL TIMED_ENTRY_CALL SELECTIVE_WAIT))
(SELECTED_COMPONENT
    ( groupe PREFIX "." SELECTOR))
(SELECTIVE_WAIT_ALTERNATIVE
    ( choix ACCEPT_ALTERNATIVE DELAY_ALTERNATIVE TERMINATE_ALTERNATIVE))
(SELECTIVE_WAIT
    ( groupe "select" SELECT_ALTERNATIVE (i+) OR_PART ( 0-1 ELSE_PART) (i+)
          "end" "select" ";"))
(SELECTOR
    ( choix all CHARACTER_LITERAL OPERATOR_SYMBOL SIMPLE_NAME))
(STATEMENTS
    ( liste1-n STATEMENT)
    ( indent ()() ()(i0) ()() ()()))
(SIMPLE_EXPRESSION
    ( groupe ( 0-1 UNARY_ADDING_OPERATOR) TERM ( 0-1 SIMPLE_EXPRESSION_AUX)))
(SIMPLE_EXPRESSION_AUX
```



```
    ( groupe BINARY_ADDING_OPERATOR TERM ( 0-1 SIMPLE_EXPRESSION_AUX)))
(SIMPLE_NAME
    ( atomique simple_name))
(SIMPLE_OR_COMPOUND_STATEMENT ; ***
    ( choix SIMPLE_STATEMENT COMPOUND_STATEMENT))
(SIMPLE_RANGE ; ***
    ( groupe SIMPLE_EXPRESSION ".." SIMPLE_EXPRESSION))
(SIMPLE_STATEMENT
    ( choix ABORT_STATEMENT ASSIGNMENT_STATEMENT CODE_STATEMENT
          DELAY_STATEMENT ENTRY_CALL_STATEMENT EXIT_STATEMENT
          GOTO_STATEMENT NULL_STATEMENT PROCEDURE_CALL_STATEMENT
          RAISE_STATEMENT RETURN_STATEMENT))
(SLICE
    ( groupe PREFIX "(" DISCRETE_RANGE ")"))
(STATEMENT
    ( groupe ( 0-1 LABEL_LIST) SIMPLE_OR_COMPOUND_STATEMENT))
(STATIC_RANGE
    ( sucre RANGE))
(STATIC_SIMPLE_EXPRESSION
    ( sucre SIMPLE_EXPRESSION))
(STRING_LITERAL
    ( atomique """ ( 0-n graphic_character) """))
(SUBPROGRAM_BODY
    ( groupe SUBPROGRAM_SPECIFICATION "is" (i0) ( 0-1 DECLARATIVE_PART)
          "begin" (i+) STATEMENTS (i-) ( 0-1 EXCEPTION_PART)
          (i0) "end" ( 0-1 DESIGNATOR) ";"))
(SUBPROGRAM_DECLARATION
    ( sucre SUBPROGRAM_SPECIFICATION ";"))
(SUBPROGRAM_NAME
    ( sucre NAME))
(SUBPROGRAM_OR_ENTRY_NAME
    ( sucre NAME))
(SUBPROGRAM_OR_PACKAGE_OR_TASK ; ***
    ( choix SUBPROGRAM_SPECIFICATION PACKAGE_BODY... TASK_BODY...))
(SUBPROGRAM_SPECIFICATION
    ( choix PROCEDURE_SPECIFICATION FUNCTION_SPECIFICATION))
(SUBTYPE_DECLARATION
    ( groupe "subtype" IDENTIFIER "is" SUBTYPE_INDICATION ";"))
(SUBTYPE_INDICATION
    ( groupe TYPE_MARK ( 0-1 CONSTRAINT)))
(SUBUNIT
    ( groupe "separate" "(" NAME ")" PROPER_BODY))
(TASK_BODY
    ( groupe TASK_BODY... "is" (i0) ( 0-1 DECLARATIVE_PART) (i0)
          "begin" (i0) STATEMENTS (i0) EXCEPTION_PART
          (i0) "end" ( 0-1 TASK_SIMPLE_NAME) ";"))
(TASK_BODY... ; ***
    ( sucre "task" "body" TASK_SIMPLE_NAME))
(TASK_DECLARATION
    ( sucre TASK_SPECIFICATION ";"))
(TASK_NAME
    ( sucre NAME))
(TASK_SPECIFICATION
    ( groupe "task" ( 0-1 type) IDENTIFIER ( 0-1 TASK_SPECIFICATION_AUX)))
```

```
(TASK_SPECIFICATION_AUX ; ***
    ( groupe "is" (i0) ENTRY_DECLARATION_LIST (i0)
            REPRESENTATIONCLAUSE_LIST (i0) "end" ( 0-1 TASK_SIMPLE_NAME)))
(TASK_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(TERM
    ( groupe FACTOR TERM_AUX))
(TERM_AUX ; ***
    ( groupe MULTIPLYING_OPERATOR ( 0-1 TERM)))
(TERMINATE_ALTERNATIVE
    ( extern "terminates ;")
    ( atomique "terminate" bruits ";"))
(TIMED_ENTRY_CALL
    ( groupe "select" (i0) ENTRY_CALL_STATEMENT ( 0-1 STATEMENTS)
            (i0) "or" DELAY_ALTERNATIVE (i0) "end" "select" ";"))
(TYPE_CONVERSION
    ( groupe TYPE_MARK "(" EXPRESSION ")"))
(TYPE_DECLARATION
    ( choix FULL_TYPE_DECLARATION INCOMPLETE_TYPE_DECLARATION
            PRIVATE_TYPE_DECLARATION))
(TYPE_DEFINITION
    ( choix ACCESS_TYPE_DEFINITION ARRAY_TYPE_DEFINITION
            DERIVED_TYPE_DEFINITION ENUMERATION_TYPE_DEFINITION
            INTEGER_TYPE_DEFINITION REAL_TYPE_DEFINITION
            RECORD_TYPE_DEFINITION))
(TYPE_MARK
    ( sucre TYPE_NAME_OR_SUBTYPE_NAME))
(TYPE_NAME_OR_SUBTYPE_NAME
    ( sucre NAME))
(TYPE_REPRESENTATION_CLAUSE
    ( choix ENUMERATION_REPRESENTATION_CLAUSE LENGTH_CLAUSE
            RECORD_REPRESENTATION_CLAUSE))
(TYPE_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(UNARY_ADDING_OPERATOR
    ( atomique ( ou "+" "-")))
(UNCONSTRAINED_ARRAY_DEFINITION
    ( groupe "array" INDEX_SUBTYPE_DEFINITION_LIST "of"
            COMPONENT_SUBTYPE_INDICATION))
(UNIVERSAL_STATIC_EXPRESSION
    ( sucre EXPRESSION))
(UNIT_SIMPLE_NAME
    ( sucre SIMPLE_NAME))
(UNIT ; ***
    ( choix LIBRARY_UNIT SECONDARY_UNIT))
(USE_CLAUSE
    ( liste1-n PACKAGE_NAME)
    ( tete "use")
    ( separateur ",")
    ( queue ";"))
(USE_CLAUSE_LIST ; ***
    ( liste0-n USE_CLAUSE))
(VARIABLE_NAME
    ( sucre NAME))
```



```
(VARIANT
    ( groupe CHOICE_LIST COMPONENT_LIST))
(CHOICE_LIST
    ( liste1-n CHOICE)
    ( tete "when")
    ( separateur "|")
    ( queue "=>"))
(VARIANT_LIST
    ( liste1-n VARIANT))
(VARIANT_PART
    ( groupe "case" DISCRIMINANT_SIMPLE_NAME "is" (i+)
            VARIANT_LIST (i-) "end" "case" ";"))
(WHEN_CONDITION ; ***
    ( sucre "when" CONDITION))
(WHEN_PART ; ***
    ( liste1-n CHOICE)
    ( tete "when")
    ( separateur "|")
    ( queue "=>"))
(WHEN_CLAUSE ; ***
    ( sucre "when" CONDITION "=>"))
(WHILE_SCHEME ; ***
    ( sucre "while" CONDITION))
(WITH_CLAUSE
    ( liste1-n UNIT_SIMPLE_NAME)
    ( tete "with")
    ( separateur ",")
    ( queue ";"))
(WITH_USE ; ***
    ( groupe WITH_CLAUSE USE_CLAUSE_LIST))
(XOR_RELATION_LIST ; ***
    ( groupe "xor" RELATION))))
```

12
Glossaire

Dans certains cas, les définitions que nous donnons ici sont celles que *nous* avons retenues. Elles peuvent différer peu ou prou de celles qui sont utilisées dans la littérature par d'autres auteurs.

Les entrées sont notées en gras. Une entrée française est suivie de sa traduction anglo-saxonne, notée entre parenthèses. Une entrée anglo-saxonne est soulignée. Elle est suivie d'un renvoi à l'entrée française correspondante. Les entrées de ce type n'existent que pour les expressions dont la traduction française n'est pas évidente.

arbre de syntaxe concrète (parse tree) : Se dit d'un arbre syntaxique construit en associant un nœud à chaque production de grammaire concrète utilisée pour faire l'analyse syntaxique du document correspondant. Le terme *arbre de dérivation* est synonyme. L'information contenue dans un tel arbre est souvent redondante car elle comprend les terminaux du langage.

arbre de syntaxe abstraite (syntax tree) : Se dit d'un arbre syntaxique construit en associant un nœud à chaque production de grammaire abstraite. L'utilisation d'un arbre abstrait permet d'éliminer les redondances d'information habituellement présentes dans un arbre de syntaxe concrète (cf. § 6.1).

arbre syntaxique : Désigne d'une manière générale les arbres de syntaxe abstraite ou les arbres de dérivation. Attention, en anglais, le terme *syntax tree* ne désigne que les arbres de syntaxe abstraite.

abstraction de données (data abstraction) : Principe selon lequel un *objet* est complètement défini par son *interface*, c'est-à-dire l'ensemble des opérations qui lui sont applicables. La réalisation de ces opérations et la représentation physique de l'état de l'objet restent cachées et inaccessibles au monde extérieur.

BNF : Backus-Naur Form : notation permettant de décrire la syntaxe d'un

langage à l'aide de productions possédant une ou plusieurs alternatives.

browser : Nom de l'outils interactif du système Smalltalk qui permet de parcourir l'arbre des classes, de chercher et de visualiser une méthode. En français, le nom habituellement donné à cet outil est *feuilleteur* ou encore *fouineur*.

classe abstraite (abstract class) : Se dit d'une classe dont l'instanciation ne présente guère d'intérêt, cette classe ne servant que de *réservoir à héritage* pour ses sous-classes (page 141).

éditeur syntaxique (syntax directed editor) : Editeur utilisant une représentation arborescente d'un document pour le manipuler.

décompilation : Défaire ce qui a été fait lors de la compilation. Dans le contexte de l'édition syntaxique, ce terme indique que l'on construit le texte source d'un document à l'aide de sa représentation arborescente (page 162).

encapsulation (encapsulation) : Mécanisme permettant de regrouper dans une même entité des données et les opérations qui s'appliquent sur ces données.

frame : Objet *prototype* inscrit dans une hiérarchie d'*héritage*, composé d'un ensemble d'attributs représentant ses propriétés. Il ne possède pas de comportement propre défini par un ensemble de procédures activables par envoi de *message*.

généricité (genericity) : Mécanisme qui permet de définir un modèle de programme dans lequel des types ou des opérations ne sont pas fixés *a priori*. Ce n'est qu'au moment de l'instanciation d'un module générique que l'on précise ces types ou ces opérations. La généricité ne présente un intérêt que lorsque le typage est statique [Meyer 86b].

grammaire LL(k) (LL(k) grammar) : Une grammaire qui permet d'écrire un analyseur descendant déterministe qui n'examine qu'un seul symbole du texte source à l'avance s'appelle une grammaire LL(1). L'idée originale de cette méthode vient de [Foster 68]. Une approche théorique, découverte indépendamment, se trouve dans [Knuth 71]. Par extension, une grammaire permettant d'écrire un analyseur descendant déterministe n'examinant que k symboles du texte source à l'avance est appelée LL(k).

hacker : Nom donné à un spécialiste d'un système ou d'un langage informatique. Le *hacker* connaît parfaitement tous les détails du système qu'il utilise, y compris son implantation. Cette appellation n'est en général pas du tout péjorative.

héritage (inheritance) : Mécanisme permettant le partage et la réutilisation de

propriétés entre les *objets*. La relation d'héritage est une relation de généralisation/spécialisation qui organise les objets en une structure hiérarchique. L'héritage peut être statique, avec recopie de structure, ou dynamique avec recherche des propriétés à l'exécution.

instance (instance) : Représentant physique d'une *classe* obtenu par moulage du dictionnaire des variables et détenant les valeurs de ces variables. Son comportement est défini par les *méthodes* de sa classe (cf § 3.2.2).

interface (interface) : Ensemble des opérations applicables à un *objet* et connues du monde extérieur.

langage LL(k) (LL(k) language) : Un langage est dit LL(k) s'il est possible d'écrire une grammaire LL(k) décrivant la syntaxe de ce langage. Ce problème est indécidable [Cunin 80, page 62].

liste de priorité (precedence list) : Liste de classes qui détermine l'ordre de recherche d'une propriété (méthode ou variable) pour une classe donnée (page 115).

liaison (binding) : Mécanisme permettant d'associer un sélecteur à la *méthode* à appliquer lors d'un envoi de *message*. La liaison peut être statique ou dynamique (cf. § 3.3.1).

masquage (shadowing, substitution, overriding) : Redéfinition d'une propriété héritée.

message (message) : Requête adressée à un *objet* demandant l'exécution d'une opération sur l'objet. Un message comprend classiquement un objet destinataire, un sélecteur de *méthode*, les arguments sur lesquels la méthode doit être appliquée et quelquefois une continuation. L'effet de ce mécanisme est aussi connu sous le nom de liaison dynamique (cf. § 3.2.4).

métaclasse (metaclass) : *Classe* dont les *instances* sont des classes.

méthode (method) : Procédure ou fonction appartenant à l'*interface* d'une *classe* et désignée par un sélecteur.

objet (object) : Entité regroupant des données et des procédures. Par abus de langage, terme générique pour désigner une *instance*.

objet composite (composite object) : *Objet* défini comme l'assemblage d'autres

objets, appelés ses parties ou ses composants.

overloading : surcharge

polymorphisme (polymorphism) : Tous les auteurs n'étant pas d'accord sur le sens de ce mot, nous nous en tenons à une définition simplifiée et intuitive : une fonction est polymorphe si elle s'applique uniformément à des arguments qui ne sont pas toujours du même type. Par exemple, la même notation peut être utilisée pour additionner deux entiers, concaténer deux chaînes de caractères ou encore faire l'union de deux ensembles. Dans les langages à objets, toute variable désignant un *objet* est potentiellement polymorphe, puisqu'elle peut désigner au cours d'une exécution plusieurs objets de types différents (cf. § 3.2.4). On parle dans ce cas de polymorphisme d'héritage [Cardelli 85], par opposition au polymorphisme paramétrique de ML [Milner 84] et au polymorphisme dit *ad hoc* que permettent la *surcharge* et le *masquage* [Wegner 87b][1].

precedence list : liste de priorité

prototype (prototype) : Etymologiquement, le mot prototype signifie *premier d'une série*, se rapporte à un premier exemplaire, construit industriellement , qui possède les mêmes fonctionnalités que l'objet final et qui sert à des tests en contexte réel.

réflexe (demon) : Procédure associée à un attribut grâce à une facette procédurale et activée automatiquement lors des accès à la valeur de l'attribut.

réutilisabilité (reusability) : Qualité d'un langage permettant de réutiliser l'existant pour le développment d'autres applications.

shadowing : masquage

slot : attribut

sorte-de (ako) : Désigne la relation d'héritage dans les réseaux sémantiques et dans les langages de *frames*.

sucre syntaxique : Terme utilisé pour désigner une construction syntaxique dont le but essentiel est d'améliorer l'esthétique du document.

superméthode (overridden method) : Désigne la *méthode* masquée par une

1. Dans cet article, à la page 508, Peter Wegner affirme : "*It appears that inheritance is more flexible and possibly more expressive than parametric polymorphism*".

méthode donnée.

surcharge (overloading) : Principe selon lequel des opérations de même nom s'appliquent à des arguments de types différents, par exemple + pour l'addition de deux nombres, la concaténation de chaînes de caractères, etc.

typage dynamique (dynamic typing) : Le type, ou la *classe* d'appartenance de l'*objet* désigné par une variable, n'est connu qu'à l'exécution. En conséquence, la validité des opérations ne peut être vérifiée qu'au moment de l'exécution (cf. § 3.2.4). Smalltalk, Objective-C et Lisp sont des langages à typage dynamique.

typage statique (static typing) : Le type d'une variable est contraint avant l'exécution. Au cours de celle-ci, la *classe* d'appartenance de l'*objet* désigné peut être une sous-classe de sa classe déclarée. La validité d'une opération est vérifiée lors de la compilation (cf. § 3.3.1). Simula, C++ et Eiffel sont des langages à typage statique.

variable de classe (class variable) : Au sens Smalltalk, variable partagée par les *instances* d'une *classe* et de ses sous-classes. Au sens ObjVlisp, variable définie dans une *métaclasse* dont la valeur est détenue par les classes qui sont instances de la métaclasse.

Bibliographie

[Abrial 85] J.R. Abrial. Spécification ou prototypage ? *Bigre + Globule (No 43-44), Prototypage, maquettage et génie logiciel*, pages 42–54, Lyon, 1985.

[Ada 83] *Reference Manual for the Ada Programming Language, ANSI/MIL-std 1815-a.* U.S. Department of Defense, 1983.

[Agha 86] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* Massachusetts Institute of Technology, Cambridge, Massasuchetts, 1986.

[Aho 73] A.V. Aho et J.D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[Aho 74] A. V. Aho et S. C. Johnson. LR Parsing (YACC - Yet Another Compiler Compiler). *ACM Computing Surveys*, 1974.

[Aho 77] A.V. Aho et J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Massachusetts, 1977.

[Aid 88] *Aida : environnement de développement d'applications. Manuel utilisateur, version 1.2.* ILOG, Gentilly, 1988.

[Albert 85] P. Albert. PROLOG et les objets. *Actes des 5èmes Journées Internationales sur les Systèmes Experts et leurs Applications.* pages 331–350, Avignon, 1985.

[Andre 86] E. André, B. Moreau, et B. Rougeot. *Vers un atelier flexible et intégré du logiciel : le projet CONCERTO.* Présentation Technique, 1986.

[Arsac 77] J. Arsac. *La construction de programmes structurés.* Dunod Informatique, Paris, 1977.

[Bahlke 86] R. Bahlke et G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[Bell 87] D. Bell, I. Morrey, et J. Pugh. *Software Engineering, A Programming Approach.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[Berry 84] G. Berry et B. Serlet. *CXYACC and LEX-KIT, version 2.1. Génération d'analyseurs syntaxiques et lexicaux pour CEYX/LELISP.* Rapport no. 4.84, GRECO Programmation, 1984.

[Bezivin 86a] J. Bézivin. *Langages objets et prototypage.* Rapport de Recherche no. 86-9, Laboratoire Informatique de Brest, 1986.

[Bezivin 86b] J. Bézivin. Smalltalk et le prototypage. *Génie Logiciel*, (3):16–19, 1986.

[Bidoit 84] M. Bidoit, C. Choppy, et S. Kaplan. ASSPEGIQUE : un environnement de spécification algébrique. *Actes 2ème Colloque de Génie Logiciel*, pages 357–371, Nice, 1984.

[Birtwistle 73] G. Birtwistle, O. Dahl, B. Myhrhaug, et K. Nygaard. *SIMULA begin.* Petrocelli Charter, New York, 1973.

[Bobrow 76] D.G. Bobrow et T. Winograd. *An Overview of KRL, a Knowledge Representation Language.* Rapport no. CSL 76-4, Xerox PARC, Palo Alto, California, 1976.

[Bobrow 77] D.G. Bobrow et T. Winograd. An overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977.

[Bobrow 83a] D.G. Bobrow et M. Stefik. *The LOOPS manual: a Data and Object Oriented Programming System for Interlisp.* Knowledge-Based VLSI Design Group Memo no. KB-VLSI-81-13, Xerox PARC, Palo Alto, California, 1983.

[Bobrow 83b] D.G. Bobrow et M.J. Stefik. *Data Oriented and Object Oriented Programming, New Metaphors for LISP.* Knowledge-Based VLSI Design Group Memo no. KB-VLSI-81-14, Xerox PARC, Palo Alto, California, 1983.

[Bobrow 87] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S. Keene, G. Kiczales, et D. A. Moon. *Common Lisp Object System Specification.* Rapport no. 87-002, ANSI Common Lisp committee X3J13, 1987.

[Boehm 82] B. W. Boehm. Les facteurs du coût du logiciel. *Techniques et Sciences Informatiques*, 1(1), 1982.

[Booch 86] G. Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, 12(2):211–221, 1986.

[Booch 87] G. Booch. *Software Engineering with Ada.* Addison-Wesley. Reading, Massachusetts, 1987.

[Booch 88] G. Booch. *Ingénierie du logiciel avec Ada, de la conception à la réalisation.* InterEditions, Paris, 1988.

[Bordier 71] J. Bordier. *Méthodes pour la mise au point de grammaires LL(1).* Thèse de 3ème cycle, Université de Grenoble, 1971.

[Borning 81] A.H. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[Borning 82a] A. Borning et D. Ingalls. A Type Declaration and Inference System for Smalltalk. *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 133–141, Albuquerque, New Mexico, 1982.

[Borning 82b] A.H. Borning et D.H.H. Ingalls. Multiple Inheritance in Smalltalk-80. *Proceedings of the 2nd Nationnal Conference on Artificial Intelligence, American Association for Artificial Intelligence*, pages 234–237, Pittsburgh, Pennsylvania, 1982.

[Borning 86] A. Borning et R. Duisberg. Constraint-Based Tools for Building User Interfaces. *ACM Transactions on Graphics*, 5(4):345-374, 1986.

[Borning 87] A. Borning et T. O'Shea. Deltatalk: an Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language. *Proceedings of the European Conference on Object Oriented Programming (ECOOP'87), special issue of Bigre No. 54*, pages 3–12, Paris, 1987.

[Bourne 85] S. Bourne. *Le système UNIX.* InterEditions, Paris, 1985.

[Brachman 83] R.J. Brachman. What IS-A is and isn't: an Analysis of Taxonomic Links in Semantic Networks. *COMPUTER*, 16(10):30–37, 1983.

[Briot 85] J.P. Briot. Les métaclasses dans les langages orientés objets. *Actes du 5ème Congrès AFCET Reconnaissance des Formes et Intelligence Artificielle*, pages 755–764, Grenoble, 1985.

[Briot 86] J.P. Briot et P. Cointe. The OBJVLISP Model: Definition of a Uniform, Reflexive and Extensible Object Oriented Language. *Proceedings of the 7th European Conference on Artificial Intelligence (ECAI), Vol. 1*, pages 270–277, Brighton, 1986.

[Buzzard 85] G.D. Buzzard et T.N. Mudge. Object-Based Computing and the Ada Programming Language. *COMPUTER*, 18(3):11–19, 1985.

[Byt 81] Byte. *Special Issue, The Smalltalk-80 System.* 6(8), McGraw-Hill, 1981.

[Canals 87] G. Canals. *GEPOL : L'éditeur syntaxique et l'environnement de conception.* Rapport de DEA, Université de Nancy 1, 1987.

[Canals 88] G. Canals, D. Colnet, Samuel Cruzlara, et J.C. Derniame. GEPI : un Générateur d'Environnements de Programmation Intégrés. *Le genie logiciel et ses applications*, pages 406–421, Toulouse, 1988.

[Cardelli 84] L. Cardelli. A Semantics of Multiple Inheritance. D.B. McQueen G. Kahn et G. Plotkin, éditeurs, *Semantics of Data Types*, pages 51–67, Springer-Verlag, Lecture Notes in Computer Science, Vol. 173, 1984.

[Cardelli 85] L. Cardelli et P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 85.

[Carter 84] C.A. Carter et W.R. LaLonde. *The design of a program editor based on constraints.* Technical Report 50, Computer Science Dept., Carleton University, Ottawa, Ontario, Canada, 1984.

[Chailloux 84] J. Chailloux, M. Devin, et J.M. Hullot. *Le_Lisp, a Portable and Efficient System.* Rapport de Recherche no. 319, INRIA, Rocquencourt, 1984.

[Chailloux 86] J. Chailloux, M. Devin, F. Dupont, J.M. Hullot, B. Serpette, et J. Vuillemin. *Le_Lisp de l'INRIA, version 15.2, le manuel de référence.* INRIA, Domaine de Voluceau, Rocquencourt, 1986.

[Choppy 86] C. Choppy. Techniques et aspect du prototypage. *Génie Logiciel*, (3):4–11, 1986.

[Chouraqui 88] E. Chouraqui et P. Dugerdil. Conflict Solving in a Frame-Like Multiple Inheritance System. *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI)*, pages 226–231, Munich, 1988.

[Clement 86] D. Clément, J. Heering, J. Incerpi, G. Kahn, P. Klint, B. Lang, et V. Pascual. *Preliminary Design of an Environment Generator.* Deliverable no. CEC348/A/T9/1, Communauté Economique Européenne, projet esprit GIPE, 1986.

[Clocksin 86] W.F. Clocksin et C.S. Mellish. *Programmer en Prolog.* Eyrolles, Paris, 1986.

[Clocksin 87] W.F. Clocksin et C.S. Mellish. *Programming in Prolog. Third. Revised and Extended Edition.* Springer-Verlag, Berlin, 1987.

[Cointe 82] P. Cointe. Une réalisation de SMALLTALK en VLISP. *Techniques et Sciences Informatiques*, 1(2):325–340, 1982.

[Cointe 83] P. Cointe. A VLISP Implementation of SMALLTALK-76. P. Degano et E. Sandewall, éditeurs. *Interactive Integrated Computing Systems*, pages 89–102, North-Holland, New York, 1983.

[Cointe 85] P. Cointe. *Implémentation et interprétation des langages orientés objets. Applications aux langages Smalltalk. Objvlisp et Formes.* Thèse d'État, Université de Paris 7, LITP 85.55, 1985.

[Cointe 87] P. Cointe. Metaclasses are First Class : the ObjVlisp Model. *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 156–167, Orlando, Florida, 1987.

[Colnet 86a] D. Colnet. *De CEYX.* Rapport CRIN no. 86-E-035. Centre de Recherche en Informatique de Nancy, 1986.

[Colnet 86b] D. Colnet, G. Masini, A. Napoli, Y. Noiret, et K. Tombre. *Les Langages Orientés Objets.* Rapport CRIN no. 86-R-077. Centre de Recherche en Informatique de Nancy, 1986.

[Cox 83a] B.J. Cox. Object-Oriented Programming in C. *UNIX Review*, 67–70, October/November 1983.

[Cox 83b] B.J. Cox. The Object Oriented Pre-Compiler — Programming Smalltalk-80 Methods in C Language. *ACM SIGPLAN Notices*, 18(1):15–22, 1983.

[Cox 84] B.J. Cox. Message/Object Programming: an Evolutionary Change in Programming Technology. *IEEE Software*, 1(1):50–61, 1984.

[Cox 86] B.J. Cox. *Object-Oriented Programming.* Addison-Wesley, Reading, Massachusetts, 1986.

[Cox 87] B.J. Cox et K.J. Schmucker. Producer: A Tool for Translating Smalltalk-80 to Objective-C. *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 423–429, Orlando, Florida, 1987.

[Cunin 80] P.Y. Cunin, M. Griffiths, et J. Voiron. *Comprendre la Compilation.* Springer-Verlag, Berlin, 1980.

[Dahl 66] O.J. Dahl, B. Myhrhaug, et K. Nygaard. SIMULA — An ALGOL-Based Simulation Language. *Communications of the ACM*, 9(9):671–678, 1966.

[Dahl 70] O.J. Dahl, B. Myhrhaug, et K. Nygaard. *SIMULA-67 Common Base Language.* Rapport no. S-22, Norwegian Computing Center, Oslo, Norway, 1970.

[Dahl 72] O.J. Dahl et C.A.R. Hoare. Hierarchical Program Structures. O.J. Dahl, E.W. Dijkstra, et C.A.R. Hoare, éditeurs, *Structured Programming.* Academic Press, New York, 1972.

[Degano 88] P. Degano, S. Mannucci, et B. Mojana. Efficient Incremental LR Parsing for Syntax-Directed Editors. *ACM Transactions on Programming Languages and Systems*, 10(3):345–373, 1988.

[DeMichiel 87] L.G. DeMichiel et R.P. Gabriel. The Common Lisp Object System: an Overview. *Proceedings of the European Conference on Object Oriented Programming (ECOOP'87), special issue of Bigre No. 54*, pages 201–220, Paris, 1987.

[Derniame 79] J.C. Derniame et J.P. Finance. *Types Abstraits de données : spécification, utilisation et réalisation.* Rapport CRIN no. 79-E-57, Centre de Recherche en Informatique de Nancy, 1979.

[Devin 88] M. Devin. AIDA : une interface portable pour Le-Lisp. *Enjeux*, (88):47–50, 1988.

[Dijkstra 76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DonzeauGouge 75] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, et J.J. Lévy. *A Structured Oriented Program Editor: A first step towards computer assisted programming.* Rapport de Recherche no. 114, INRIA, Domaine de Voluceau, Rocquencourt, 1975.

[DonzeauGouge 80] V. Donzeau-Gouge, G. Huet, G. Kahn, et B. Lang. *Programming Environments based on Structured Editors: The MENTOR Experience.* Rapport de Recherche no. 26, INRIA, Domaine de Voluceau, Rocquencourt, 1980.

[DonzeauGouge 84] V. Donzeau-Gouge, G. Kahn, B. Lang, et B. Melese. Documents Structure and Modularity in Mentor. *ACM SIGSOFT-SIGPLAN Software Engineering Symposium on Practical Software Developpement Environments*, pages 141–148, Pittsburgh, Pennsylvania, 1984.

[Doyle 86] K. Doyle, B. Haynes, M. Lentczner, et L. Rosenstein. An Object Oriented Approach to Macintosh Application Development. *Actes des 3èmes JLOO, Bigre+Globule No. 48*, pages 46–54, Paris, 1986.

[Ducournau 87a] R. Ducournau et M. Habib. On some Algorithms for Multiple Inheritance in Object Oriented Programming. *Proceedings of the European Conference on Object Oriented Programming (ECOOP'87), special issue of Bigre No. 54*, pages 291–300, Paris, 1987.

[Ducournau 87b] R. Ducournau et J. Quinqueton. *YAFOOL : encore un langage objet à base de frames, version 3.0. Le manuel de référence.* INRIA/SEMA.METRA, Rocquencourt, 1987.

[Ducournau 89] R. Ducournau et M. Habib. La multiplicité de l'héritage dans les langages à objets. *à paraître dans Techniques et Sciences Informatiques*, 1989.

[Dugerdil 86] P. Dugerdil. A propos des mécanismes d'héritage dans les langages orientés objets. *Actes du 2ème Colloque International d'Intelligence Artificielle*, pages 67–77, Marseille, 1986.

[Dutta 85] K. Dutta. Modular Programming in C: an Approach and an Example. *ACM SIGPLAN Notices*, 20(3):9–15, 1985.

[Earley 70] J. Earley. An efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[Ebel 83] N. Ebel. *EDIS : un éditeur de symboles avec vérificateur syntaxique.* Thèse No494, Ecole Polytechnique Fédérale de Lausanne, 1983.

[Ege 87] R.K. Ege, D. Maier, et A. Borning. The Filter Browser, Defining Interfaces Graphically. *Proceedings of the European Conference on Object Oriented Programming (ECOOP'87), special issue of Bigre No. 54*, pages 155–165, Paris, 1987.

[Epstein 88] D. Epstein et W.R. LaLonde. A Smalltalk Window System Based On Constraints. *Proceedings of the 3rd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88)*, pages 83–94, San Diego, California, 1988.

[Etherington 87] D.W. Etherington. More on Inheritance Hierarchies with Exceptions. *Proceedings of the 6th National Conference on Artificial Intelligence, American Association for Artificial Intelligence*, Seattle, Washington, 1987.

[Fairley 85] R. Fairley. *Software Engineering Concepts.* McGraw-Hill, New York, 1985.

[Ferber 83] J. Ferber. *MERING : un langage d'acteurs pour la représentation et la manipulation des connaissances.* Thèse de Docteur Ingénieur, Université de Paris 6, 1983.

[Ferber 88] J. Ferber et J.P. Briot. Design of a Concurrent Language for Distributed Artificial Intelligence. *Proceedings on the International Conference on Fifth Generation Computer. Vol. 2,* pages 755–762, Tokyo, 1988.

[Fikes 87] R. Fikes et R. Nado. Semantically Sound Inheritance for a Formally Defined Frame Language with Defaults. *Proceedings of the 6th Nationnal Conference on Artificial Intelligence. American Association for Artificial Intelligence,* pages 443–448, Seattle, Washington, 1987.

[Fla 86] Reference Guide to Symbolics Common Lisp: Language Concepts. Symbolics Release 7 Document Set, 1986.

[Foster 68] J.M. Foster. A Syntax Improving Device. *Computer Journal,* 1968.

[Ghezzi 80] C. Ghezzi et D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM,* 27(3):564–579, 1980.

[Girardot 85] J.J. Girardot. *Les langages et les sytèmes LISP.* Editest, Paris, 1985.

[Goldberg 83] A. Goldberg et D. Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Goldberg 84a] A. Goldberg. *SMALLTALK-80, the Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[Goldberg 84b] A. Goldberg. The Smalltalk-80 System Release Process. *Smalltalk-80, Bits of history, Words of Advice,* pages 3–8, Addison-Wesley, Reading, Massachusetts, 1984.

[Greenblatt 84] R.D. Greenblatt, T.F. Knight Jr., J. Holloway, D.A. Moon, et D.L. Weinreb. The LISP Machine. D.R. Barstow, H.E. Shrobe, et E. Sandewall, éditeurs, *Interactive Programming Environments,* pages 326–352, McGraw-Hill, New York, 1984.

[Greibach 64] S.A. Greibach. Formal Parsing System. *Communications of the ACM,* 7(8), 1964.

[Greibach 65] S.A. Greibach. A New Normal Form Theorem for Context-Free Phrase Structure Grammars. *Journal of the ACM,* 12, 1965.

[Gries 71] D. Gries. *Compiler Construction for Digital Computer.* J. Wiley ans Sons, New York, 1971.

[Gross 70] M. Gross et A. Lentin. *Introduction to Formal Grammars.* Springer-Verlag, Berlin, 1970.

[Guyard 84] J. Guyard et J.P. Jacquot. Maiday : an environment for guided programming with a definitional language. *Proceedings of the 7th international conference on software engineering,* 1984.

[Hayes 79] P. Hayes. The Logic of Frames. D. Metzing, éditeur, *Frame Conception and Text Understanding,* pages 46–61. de Gruyter, Brighton, 1979.

[Hendler 86] J. Hendler. Enhancement for multiple-inheritance. *ACM SIGPLAN Notices,* 21(10), 1986.

[Hewitt 73] C. Hewitt, P. Bishop, et R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73),* pages 235–245, Stanford, California, 1973.

[Hofstadter 85] D. Hofstadter. *Gödel, Escher, Bach, les Brins d'une Guirlande Eternelle.* InterEditions, Paris, 1985.

[Horowitz 83] E. Horowitz. *Fundamentals of Programming Languages.* Springer-Verlag, Berlin, 1983.

[Hullot 83a] J.M. Hullot. Ceyx, a Multiformalism Programming Environment. *Proceedings IFIP83,* Paris, 1983.

[Hullot 83b] J.M. Hullot. CEYX: un environnement de programmation multi-formalisme. *Actes des 1ères JLOO, Bigre+Globule No. 37,* le Cap d'Agde, 1983.

[Hullot 85] J.M. Hullot. *ALCYONE, la boîte à outils objets.* Rapport de Recherche no. 60, INRIA, Domaine de Voluceau, Rocquencourt, 1985.

[Hullot 86] J.M. Hullot. SOS Interface, un générateur d'interfaces homme-machine (pour Macintosh). *Actes des 3èmes JLOO, Bigre+Globule No. 48,* pages 69–78, Paris, 1986.

[Ingalls 78] D.H. Ingalls. The SMALLTALK-76 Programming System Design and Implementation. *Proceedings of the 5th SPPL,* pages 9–15, Tucson, Arizona, 1978.

[Ingalls 81] D.H.H. Ingalls. Design Principles Behind Smalltalk. *Byte*, 6(8):286–298, 1981.

[Jacquot 84] J.P. Jacquot. Etude d'un outil d'assistance à la conception méthodique de programmes. Thèse de Docteur-Ingénieur. Institut National Polytechnique de Lorraine, 1984.

[Jalili 82] F. Jalili et J.H. Gallier. Building friendly parsers. *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 196–206, New York, 1982.

[Johnson 88] R.E. Johnson et B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Kay 76] A. Kay et A. Goldberg. *SMALLTALK-72 Instruction Manual.* Rapport no. SSL 76-6, Xerox PARC, Palo Alto, California, 1976.

[Kehler 84] T.P. Kehler et G.D. Clemenson. An Application Development System for Expert Systems. *Systems and Software*, 212–224, January 1984.

[Kernighan 78] B.W. Kernighan et D. Ritchie. *The C programming language.* Prentice Hall, 1978.

[Kernighan 81] B.W. Kernighan et J.R. Mashey. The UNIX Programming Environment. *COMPUTER*, 14(4):25–34, 1981.

[Knuth 65] D.E. Knuth. On the Translation of Languages from left to Right. *Information and Control*, 8(6):607–639, 1965.

[Knuth 68] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[Knuth 71] D.E. Knuth. Top-Down Syntax Analysis. *Acta Informatica*, 1(2):79–110, 1971.

[Koenig 88] A. Koenig. Why I use C++. *Journal of Object-Oriented Programming*, 1(2):38–42, 1988.

[Krakowiak 85] S. Krakowiak. *Principes des systèmes d'exploitation des ordinateurs.* Dunod Informatique, Paris, 1985.

[Krasner 88] G.E. Krasner et S.A. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[Lamport 86] L. Lamport. LaTeX *: user's guide & reference manual.* Rapport, Reading, Massachusetts, 1986.

[Lentin 67] A. Lentin et M. Gross. *Notions sur les grammaires formelles.* Gauthier-Villar, 1967.

[Lieberman 81] H. Lieberman. *A Preview of Act 1.* AI Memo no. 625, Artificial Intelligence Laboratory, MIT, Cambridge, Massasuchetts, 1981.

[Lieberman 86] H. Lieberman. Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems. *Actes des 3èmes JLOO, Bigre+Globule No. 48*, pages 79–89, Paris, 1986.

[Liskov 74] B. Liskov et S. Zilles. Programming with Abstract Data Types. *ACM SIGPLAN Notices*, 9(4):50–59, 1974.

[Liskov 75] B. Liskov et S. Zilles. Specification Techniques for Data Abstraction. *IEEE Transactions on Software Engineering*, 1(1):7–19, 1975.

[Liskov 77] B. Liskov, A. Snyder, R. Atkinson, et C. Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[Livercy 78] C. Livercy. *Théorie des programmes.* Dunod Informatique, Paris, 1978.

[Lu 84] C. Lu. *Le livre du Macintosh.* Microsoft Press, Cedic Nathan, Paris, 1984.

[Mac 86] *Inside Macintosh.* Addison-Wesley, 1986.

[Marchand 78] P. Marchand. *Théorie de langages (régulier et algébriques) et des automates.* Rapport CRIN no. 78-E-010. Centre de Recherche en Informatique de Nancy, 1978.

[Marcke 87] K. Van Marcke. KRS: An Object-Oriented Representation Language. *Revue d'Intelligence Artificielle*, 1(4):43–68, 1987.

[Masini 89] G. Masini, A. Napoli, D. Colnet, D. Leonard, et K. Tombre. *Les Langages à Objets (à paraître).* InterEditions, Paris, 1989.

[MedinaMora 81] R. Medina-Mora et D. S. Notkin. *ALOE User's and Implementor's Guide.* Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1981.

[Meyer 79] B. Meyer. *Les concepts de SIMULA-67.* Atelier Logiciel EDF, 1979.

[Meyer 85] B. Meyer, J.M. Nerson, et S.H. Ko. Showing Programs on a Screen. *Science of Computer Programming*, 5(2):111–142, 1985.

[Meyer 86a] B. Meyer. Cépage: Towards Computer-Aided Design of Software. *Computer Language*. 3(9):43–53. 1986.

[Meyer 86b] B. Meyer. Genericity versus Inheritance. *Proceedings of the 1st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, pages 391–405, Portland, Oregon, 1986.

[Meyer 86c] B. Meyer. *Eiffel : un langage et une méthode pour le génie logiciel.* Interactive Software Engineering, Inc., 1986.

[Meyer 87a] B. Meyer. Cépage: Towards Computer-Aided Design of Software. Interactive Software Engineering, Inc., 1987.

[Meyer 87b] B. Meyer. Eiffel: Programming for Reusability and Extendibility. *ACM SIGPLAN Notices*, 22(2):85–94, 1987.

[Meyer 87c] B. Meyer. *LDL: A language Description Language.* Interactive Software Engineering, Inc., 1987.

[Meyer 87d] B. Meyer. Reusability: the Case for Object Oriented Design. *IEEE Software*, 4(3):50–64, 1987.

[Meyer 88a] B. Meyer. *Programming as Contracting.* Interactive Software Engineering, Inc., 1988.

[Meyer 88b] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall International Series in Computer Science, 1988.

[Meyer 89] B. Meyer. *Conception et programmation par objets, pour du logiciel de qualité.* InterEditions, Paris, 1989.

[Milner 84] R. Milner. A Proposal for Standard ML. *Proceedings SLFP84*, pages 184–197, Austin, Texas, 1984.

[Minsky 75] M. Minsky. A Framework for Representing Knowledge. P. Winston, éditeur, *The Psychology of Computer Vision*, pages 211–281, McGraw-Hill, New York, 1975.

[Moon 80] D. Moon et D. Weinreb. *FLAVORS: Message Passing in the LISP Machine.* AI Memo no. 602, Artificial Intelligence Laboratory, MIT, Cambridge, Massasuchetts, 1980.

[Moon 86] D. Moon. Object-Oriented Programming with Flavors. *Proceedings of the 1st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, pages 1–8, Portland, Oregon, 1986.

[Naur 63] P. Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[Nierstrasz 88] O.M. Nierstrasz. A Survey of Orient-Oriented Concepts. D. Tsichritzis, éditeur, *Active Object Environments*, pages 1–17, Centre Universitaire d'Informatique, Université de Genève, 1988.

[Ossanna 86] J.F. Ossanna. *Nroff/Troff user's manual.* Berkeley Software Diffusion, Berkeley, California USA, 1986.

[Pair 88] C. Pair, R. Mohr, et R. Schott. *Construire les algorithmes.* Dunod Informatique, Paris, 1988.

[Pro 86] *MAQUETTAGE ET PROTOTYPAGE : OUTILS ET TECHNIQUES.* Génie Logiciel (revue No 3), 1986.

[Quillian 68] M.R. Quillian. Semantic Memory. M. Minsky, éditeur, *Semantic Information Processing*, pages 227–270, MIT Press, Cambridge, Massasuchetts, 1968.

[Rechenmann 88] F. Rechenmann. *SHIRKA : système de gestion de bases de connaissances centrées-objet, manuel de référence.* INRIA/-ARTEMIS, Grenoble, 1988.

[Remy 81] J.L. Remy. *Arbres : algorithmes et structures de données.* Rapport CRIN no. 81-E-010, Centre de Recherche en Informatique de Nancy, 1981.

[Reps 81] T. Reps. *The Synthesizer Editor Generator: Reference Manual.* Departement of Computer Science, Cornell University, 1981.

[Reps 83a] T. Reps. *Generating Language-Based Environments.* Thèse de Doctorat, Cornell University, 1983.

[Reps 83b] T. Reps, T. Teitelbaum, et A. Demers. Incremental Context-Dependent Analysis for Languages-Based Editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[Reps 84] T. Reps et T. Teitelbaum. The Synthesizer Generator. *ACM SIGPLAN Notices*, 19(5):42–48, 1984.

[Reps 85] T. Reps et T. Teitelbaum. *The Synthesizer Editor Generator Reference Manual.* Departement of Computer Science, Cornell University, 1985.

[Roberts 77] R.B. Roberts et I.P. Goldstein. *The FRL Manual.* The AI Magazine no. 409, Artificial Intelligence Laboratory, MIT, Cambridge, Massasuchetts, 1977.

[SaintJames 87] E. Saint-James. *De la méta-récursivité comme outil d'implémentation.* Thèse de Doctorat d'Etat, Université de Paris 6, 1987.

[Sandewall 84] E. Sandewall. Programming in an Interactive Environment: The LISP Experience. D.R. Barstow, H.E. Shrobe, et E. Sandewall, éditeurs, *Interactive Programming Environments*, pages 31–80, McGraw-Hill, New York, 1984.

[Scheid 74] F. Scheid. *Introduction A L'informatique.* Série Schaum, Paris, 1974.

[Schmucker 86a] K.J. Schmucker. Macapp: an Application Framework. *Byte*, 11(8):189–193, 1986.

[Schmucker 86b] K.J. Schmucker. Object-Oriented Languages for the Macintosh. *Byte*, 11(8):177–185, 1986.

[Schmucker 86c] K.J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hayden Book Company, Hasbrouck Heights, New Jersey, 1986.

[Seidewitz 87] E. Seidewitz. Object-Oriented Programming in Smalltalk and Ada. *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 202–213, Orlando, Florida, 1987.

[Serpette 84] B.P. Serpette. *Contextes, Processus, Objets, Séquenceurs : FORMES.* Thèse de 3ème cycle, Université de Paris 6, 1984.

[Silva 87] S. Cruzlara Silva et J.C. Derniame. *Yet Another Programming Environment Generator Based on Attribute Grammars.* Rapport CRIN no. 87-R-079, Centre de Recherche en Informatique de Nancy, 1987.

[Silva 88] S. Cruzlara Silva. *GEODE : un système pour la génération d'environnements de programmation intégrés.* Thèse, Institut National Polytechnique de Lorraine, 1988.

[Snyder 86] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *Proceedings of the 1st ACM Conference on Object-Oriented Programming Systems. Languages and Applications (OOPSLA'86)*, pages 38–45, Portland, Oregon, 1986.

[Sobelman 85] G. E. Sobelman et D. E. Krekelburg. *Advanced C, Techniques & Applications.* Que Corporation, Indianapolis, 1985.

[Stallman 86] R. Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, Massasuchetts, 1986.

[Steele Jr 84] G.L. Steele Jr. *COMMON LISP: the Language.* Digital Press, Bedford, Massachusetts, 1984.

[Stein 88] L. Stein, H. Lieberman, et D. Ungar. The Treaty of Orlando: A Shared View of Sharing. W. Kim et F. Lochovsky, éditeurs, *Object-Oriented Concepts, Applications and Databases*, Addison-Wesley, Reading, Massachusetts, 1988.

[Stroustrup 86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Series in Computer Science, Reading, Massachusetts, 1986.

[Stroustrup 87a] B. Stroustrup. Multiple Inheritance for C++. *Proceedings of EUUG Spring'87 Conference*, Helsinki, 1987.

[Stroustrup 87b] B. Stroustrup. What is "Object-Oriented Programming"? *Proceedings of the European Conference on Object Oriented Programming (ECOOP'87), special issue of Bigre No. 54*, pages 57–76, Paris, 1987.

[Tichy 87] W.F. Tichy. What Can Software Engineers Learn from Artificial Intelligence? *COMPUTER*, 20(11):43–54, 1987.

[Touati 87] H. Touati. Is Ada an Object Oriented Programming Language? *ACM SIGPLAN Notices*, 22(5):23–26, 1987.

[Touretzky 84] D.S. Touretzky. *LISP: a Gentle Introduction to Symbolic Computation.* Harper & Row, New York, 1984.

[Touretzky 86] D.S. Touretzky. *The Mathematics of Inheritance.* Morgan Kaufmann Publishers, Inc., Los Altos, California, 1986.

[Touretzky 87] D.S. Touretzky, J.F. Horty, et R.H. Thomason. A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems. *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI'87)*, pages 476–482, Milano, Italy, 1987.

[Wegman 80] M.N. Wegman. Parsing for structural editors. *Proceeding of the 21th Annual Symposium on Foundation of Computer Science*, pages 320–327, New York, 1980.

[Wegner 87a] P. Wegner. Dimensions of Object-Based Language Design. *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 168–182, Orlando, Florida, 1987.

[Wegner 87b] P. Wegner. The Object-Oriented Classification Paradigm. B. Shriver et P. Wegner, éditeurs, *Research Directions in Object Oriented Programming*, pages 479–560, MIT Press, Cambridge, Massasuchetts, 1987.

[Weinreb 81] D. Weinreb et D. Moon. *Lisp Machine Manual.* Artificial Intelligence Laboratory, MIT, Cambridge, Massasuchetts, 1981.

[Wertz 85] H. Wertz. *LISP, une introduction à la programmation.* Masson, Paris, 1985.

[White 78] J.L. White. Program is Data. *ACM SIGPLAN Notices.* 13(8):217–223, 1978.

[Wilensky 86] R. Wilensky. *LISPcraft.* W.W. Norton & Company, New York, 1986.

[Winston 84a] P.H. Winston et B.K.P. Horn. *LISP, second edition.* Addison-Wesley, Reading, Massachusetts, 1984.

[Winston 84b] P.H. Winston et K.A. Prendergast, éditeurs. *The AI Business.* MIT Press, Cambridge, Massasuchetts, 1984.

[Winston 88] P.H. Winston et B.K.P. Horn. *LISP.* Addison-Wesley, Reading, Massachusetts, Third édition, 1988.

[Wirth 71a] N. Wirth. Program Development by Step-wise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[Wirth 71b] N. Wirth. The Programming language PASCAL. *Acta Informatica*, 1(1):35–63, 1971.

[Wirth 76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall series in Automatic Computation, Englewood Cliffs, New Jersey, 1976.

[Wirth 84] N. Wirth. History and Goals of Modula-2. *Byte*, 9(8):145–152, 1984.

[Wirth 87] N. Wirth. *Algorithmes et structures de données.* Eyrolles, Paris, 1987.

[Yamamoto 84] J. Yamamoto. *HAGAKURE, le livre secret des Samouraïs.* G. Trédaniel, Paris, 1984.

[Yonezawa 86] A. Yonezawa, J.P. Briot, et E. Shibayama. Object-Oriented Concurrent Programming in ABCL/1. *ACM SIGPLAN Notices*, 21(11):258–268, 1986.

[Yoshikawa 83] E. Yoshikawa. *La parfaite lumière.* Balland, 1983.

Index

NOM DE L'ETUDIANT : Monsieur COLNET Dominique

NATURE DE LA THESE : DOCTORAT DE L'UNIVERSITE DE NANCY I EN INFORMATIQUE

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 27 JAN. 1989 n° 191

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

R. MAINARD

Résumé

Cette thèse peut être lue sous deux angles différents selon que l'on s'intéresse à l'édition syntaxique ou à la programmation objet.

Le chapitre 1 effectue un survol des différentes caractéristiques de GÉPI, un Générateur d'Environnements de Programmation Intégrés. Le langage de description de langages défini pour ce générateur est présenté au chapitre 2.

Les trois chapitres qui suivent, 3 4 et 5, sont consacrés à la programmation objet et peuvent être lus indépendamment : vocabulaire, notions de base, intérêt de la programmation objet, comparaison avec une approche plus classique et enfin, la description de leloup, le langage à objets défini pour le projet.

Les chapitres 6 et 7 sont consacrés à l'implantation du manipulateur de documents de GÉPI. L'implantation est décrite en leloup et peut être vue comme un exercice en grandeur nature d'utilisation d'un langage à objets.

Le dernier chapitre fait le bilan du projet dans son ensemble, aussi bien en ce qui concerne le générateur lui même qu'en ce qui concerne l'approche utilisée pour son implantation : la programmation objet en leloup, un langage très proche de Smalltalk.