THÈSE

présentée à

L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir

LE GRADE DE DOCTEUR-INGÉNIEUR
Spécialité : **Informatique**

par

Philippe DESCHAMP

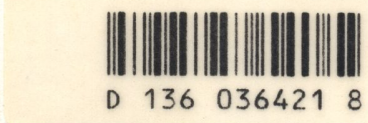
Service Commun de la Documentation
INPL
Nancy-Brabois

Sujet de la thèse :

PRODUCTION DE COMPILATEURS À PARTIR D'UNE DESCRIPTION SÉMANTIQUE DES LANGAGES DE PROGRAMMATION : LE SYSTÈME PERLUETTE

Soutenue le 24 octobre 1980 devant la Commission composée de :

M.	**C.**	**PAIR**	Président
Mme	**M.C.**	**GAUDEL**	
MM.	**J.P.**	**HATON**	Examinateurs
	B.	**LORHO**	


D 136 036421 8

136036421 8

CM 1980 DESCHAMP, P.

THÈSE

présentée à

L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir

LE GRADE DE DOCTEUR-INGÉNIEUR
Spécialité : Informatique

par

Philippe DESCHAMP

Service Commun de la Documentation
INPL
Nancy-Brabois

Sujet de la thèse :

PRODUCTION DE COMPILATEURS À PARTIR D'UNE DESCRIPTION SÉMANTIQUE DES LANGAGES DE PROGRAMMATION : LE SYSTÈME PERLUETTE

Soutenue le 24 octobre 1980 devant la Commission composée de :

M.	**C. PAIR**	Président
Mme	**M.C. GAUDEL**	
MM.	**J.P. HATON**	Examinateurs
	B. LORHO	

Je remercie vivement C. PAIR, Président de l'INPL, de me faire l'honneur de présider le jury de cette thèse ; je n'oublie pas l'enseignement qu'il m'a prodigué à l'Ecole des Mines et lui suis reconnaissant d'avoir suivi le développement de mes travaux à l'INRIA depuis lors.

Je garde un excellent souvenir de l'enseignement de DEA de J.P. HATON et je suis spécialement honoré de sa présence dans ce jury.

B. LORHO m'a accueilli dans son équipe où j'ai pu développer mes recherches et leurs applications, grâce en particulier à ses travaux antérieurs ; ses encouragements et ses critiques amicales m'ont été précieux.

Que M.C. GAUDEL trouve ici l'expression très sincère de ma profonde gratitude et de mon amitié ; elle m'a révélé les affres et les joies du travail de recherche, et ses directives et conseils ont été déterminants dans la réalisation de ce travail.

Je tiens à remercier tout particulièrement M. MAZAUD pour sa collaboration efficace, et pour m'avoir supporté malgré mes tabacs à pipe malodorants.

Je remercie aussi l'ensemble de mes collègues de l'équipe Langages et Traducteurs pour l'ambiance de saine et franche camaraderie qui y règne.

Je remercie Madame C. AUBIN qui a effectué la frappe de cette thèse.

Qu'on me permette enfin de remercier Ch. MARINO, ma femme, pour ce qu'elle est, ainsi que mes parents pour leurs encouragements.

RESUME

Cette thèse est consacrée à la présentation du système Perluette de production de traducteurs ; ce système est fondé sur une sémantique formelle des langages de programmation, basée sur les types abstraits algébriques, et permet l'utilisation d'attributs pour le traitement des contraintes textuelles.

Le chapitre I est une présentation générale de cette sémantique, ainsi que des méthodes de spécification et de preuve que nous utilisons ; nous y traitons un exemple simple, mais complet. Le chapitre II décrit la structure du système ainsi que celle des traducteurs qu'il produit. Le chapitre III est consacré à la description des modules du système et de leur implantation - par attributs -. Le chapitre IV enfin, présente la description détaillée d'un langage de programmation réaliste, qui comporte des procédures et des tableaux d'entiers.

SOMMAIRE

	pages
CHAPITRE 0 : INTRODUCTION	
0.1. Historique de la traduction dans le domaine des calculateurs	0.1
0.2. La méta-compilation	0.3
0.3. Rappels sur les grammaires à attributs - le système DELTA	0.4
0.4. Plan de cette thèse	0.11
CHAPITRE I : METHODES : PRESENTATION GENERALE	
I.1. Principes	I.12
I.2. Un exemple	I.16
I.2.1. Description du langage à traduire	
I.2.2. Le langage objet	
I.2.3. Spécification de la traduction	
I.3. Preuve de la spécification	I.36
I.3.1. La méthode de preuve	
I.3.2. Réalisation de la preuve	
I.3.2.1. Types et opérations non modifiables	
I.3.2.2. Opérations modifiables	
I.3.2.3. Modifications	
CHAPITRE II : UTILISATION DU SYSTEME	
II.1. Structure et fonctionnement du traducteur produit	II.43
II.1.1. Structure générale	
II.1.2. La première phase	
II.1.3. La deuxième phase	
II.1.4. La troisième phase	
II.1.5. Vue d'ensemble	

II.2. Méthode d'utilisation de Perluette II.49
II.2.1. Le langage source
II.2.1.1. Le type de données source
II.2.1.2. La grammaire attribuée du langage source
II.2.2. Le langage objet
II.2.2.1. Le type abstrait objet
II.2.2.2. La description du code
II.2.3. La représentation des termes sources par des termes objets
II.2.3.1. Représentation des types
II.2.3.2. Représentation des opérations
II.3. Récapitulation II.69

CHAPITRE III : REALISATION DU SYSTEME

III.1. Implantation des traducteurs produits par Perluette III.72
III.2. Implantation de Perluette III.77
III.2.1. Implantation de Perluette 1
III.2.2. Implantation de Perluette 3
III.2.3. Implantation de Perluette 2
III.3. Exemples simples de fonctionnement III.88
III.3.1. Traitement de l'addition indépendamment des autres opérations
III.3.2. Traitements couplés de l'addition et de la soustraction
III.4. Conclusion III.94

CHAPITRE IV : EXEMPLE : DESCRIPTION DE SIMPROC

IV.1. Le langage SIMPROC IV.95
IV.1.1. Les expressions de SIMPROC
IV.1.2. Axiomatisation d'identificateurs non tous différents
IV.1.3. Les instructions de SIMPROC
IV.1.4. Les fonctions sémantiques
IV.2. Structure du langage objet IV.114
IV.3. Implantation de SIMPROC IV.120
IV.4. Conclusion IV.126

CHAPITRE V : CONCLUSION V.127

ANNEXES A.129

BIBLIOGRAPHIE B.160

CHAPITRE 0

INTRODUCTION

0.1 Historique de la traduction dans le domaine des calculateurs

Les premiers calculateurs étaient programmés directement dans "leur" langage, dit "langage machine" ; très rapidement fut introduite la notion de "langage autocode", assurant une programmation plus souple et "lisible" ; cette notion permit le développement de langages d'assemblage, traités par des assembleurs, autorisant l'utilisation de noms symboliques et de "macro-instructions". Puis furent définis des langages plus "évolués", dont les phrases étaient les mêmes quel que soit le calculateur auquel elles étaient destinées, d'où la possibilité d'écrire des programmes "généraux", valables pour tous les ordinateurs possédant un traducteur du langage utilisé.

L'écriture des traducteurs de ces langages (souvent dénommés compilateurs, en raison du fait qu'ils composent un programme particulier en compilant* l'ensemble des instructions possibles du langage machine) a suivi un cheminement semblable : alors qu'à l'origine un compilateur était écrit dans le langage autocode du calculateur auquel il était destiné, sont apparus ensuite des "compilateurs croisés", permettant de traduire sur une machine des programmes destinés à une autre, ce qui a mené à la notion

---

* : du latin *compilare* (piller) ; une autre étymologie du mot "compiler" dans son sens informatique est quelquefois proposée : les compilateurs ont été parmi les premiers logiciels à utiliser des piles comme structure de données, d'où leur nom, dérivé du latin *cum* (avec) *pila* (pile).

d'auto-compilation : disposant d'un compilateur d'un langage sur une machine, on peut écrire dans ce langage un compilateur croisé de ce langage, produisant des programmes pour une autre machine, et ce compilateur croisé peut se traduire lui-même, ce qui permet d'obtenir pour l'autre machine un compilateur "direct" du langage. Des manipulations plus complexes ont été aussitôt imaginées, qui ne seront pas rappelées ici (voir notamment Earley et Sturgis [ES 70] et leurs diagrammes en "T").

Parallèlement à ces techniques d'écriture de traducteurs de nombreuses recherches sur la description formelle des langages de programmation ont été poursuivies ; en particulier, la formalisation de la syntaxe de ces langages a été menée à bien, et il en a résulté la possibilité de définition d'outils qui soulagent les implanteurs de traducteurs de besognes fastidieuses : les générateurs d'analyseurs lexicographiques et syntaxiques sont désormais chose courante [AU 77, Bou 77, Bou 80]. La définition d'un langage est maintenant considérée en général comme formée de trois parties : sa définition lexicale et syntaxique, la description des contraintes contextuelles que doivent respecter les programmes écrits dans ce langage (par exemple les règles de déclaration des symboles utilisés) et la spécification d'outils permettant d'obtenir la signification - ou sémantique - de tout programme du langage souscrivant à ces contraintes. Un consensus existe pour la définition syntaxique et lexicale, exprimée soit en "Backus-Naur-Form"(BNF) ou une de ses nombreuses variantes, soit sous forme d'expressions régulières ; pour la description des contraintes contextuelles, les méthodes les plus connues sont les "doubles grammaires" définies par van Wijngaarden, les "grammaires affixes" décrites par Koster et les attributs introduits par Knuth ; les notations permettant d'attribuer une valeur sémantique à un programme -sémantiques axiomatique [Hoa 69], dénotationelle [Ten 76] , algèbrique [CN 76],... - n'ont pu jusqu'ici faire l'unanimité, et sont pour la plupart mal adaptées à la compilation telle qu'on la conçoit en général, c'est à dire la traduction en un "langage machine" existant.

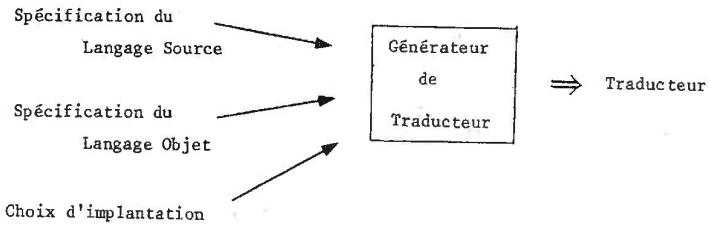
0.2 La méta-compilation

L'approche conventionnelle de l'écriture des compilateurs est de construire chaque compilateur d'un langage pour une machine donnée de manière ad hoc : chaque partie du compilateur est placée à l'endroit qui semble le plus commode ; une conséquence de cette approche est que les parties dépendant du langage source ou du langage objet sont éparpillées dans le corps du compilateur. Il est par conséquent extrêmement difficile de réutiliser des parties d'un compilateur existant pour écrire un compilateur du même langage pour une autre machine, ou d'un autre langage pour la même machine.

D'autre part, une caractéristique vitale pour un compilateur est sa fiabilité, car c'est un produit utilisé en permanence dans la vie informatique quotidienne et sur lequel s'appuient tous les usagers. Le problème de l'équivalence du programme source et du programme objet engendré est donc fondamental. Or la façon dont sont distribuées les dépendances vis à vis du langage source et du langage objet rend très incommode toute preuve de correction d'un compilateur conventionnel.

Il est donc indispensable, pour réduire le coût de production des traducteurs ainsi que pour permettre la démonstration de leur validité, de disposer de systèmes acceptant en entrée, de manière indépendante, les spécifications du langage à traduire - langage source - et du langage objet - résultat de la traduction -. Pour spécifier totalement un traducteur, il est nécessaire de décrire, outre ces deux langages, les choix d'implantation que l'on désire pour les objets manipulés par le langage source : cette description comprendra notamment, suivant le langage en cause, le choix pris pour l'implantation des tableaux - par ligne ou par colonne -, comment on réalise le passage de paramètre par nom ou la façon dont une procédure accède aux variables qui lui sont externes - variables dites "globales" -. Dans la plupart des traducteurs actuels, ces choix d'implantation sont implicites - seul le texte du compilateur permet de les connaître - et sont, par ailleurs, quelquefois explicités dans des articles écrits par les auteurs du traducteur. La description de ces choix d'implantation est en fait une partie primordiale de l'écriture d'un traducteur, qui doit , par conséquent, être spécifiée rigoureusement.

Un système de méta-compilation doit donc être schématiquement du type suivant :



Le système Perluette (*) présenté ici fonctionne selon ce schéma ; les spécifications des langages source et objet sont faites en utilisant le même formalisme de base, les types de données abstraits algébriques ; les contraintes contextuelles associées au langage source, ainsi qu'une partie de la traduction, sont écrites sous la forme d'attributs sémantiques.

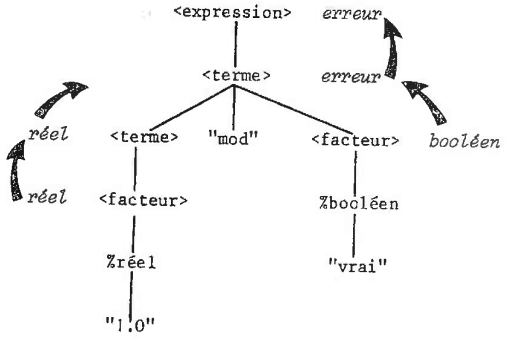
0.3 Rappels sur les grammaires à attributs - le système DELTA

Les attributs sont utilisées, dans le système Perluette, comme une méthode de vérification des contraintes contextuelles et d'évaluation de la sémantique des programmes : les aspects non contextuels du langage sont décrits par une grammaire libre du contexte, sous une forme dérivée de la BNF mentionnée précédemment ; pour chaque production de cette grammaire peuvent être données des définitions d'attributs, qui sont en fait des modes de calcul des valeurs qui seront attachées au noeuds des arbres syntaxiques construits à partir de la grammaire. Lors de la construction d'un tel arbre syntaxique sur un programme, les attributs sont évalués, pour tout noeud, en fonction des attributs des noeuds père et fils ; ils permettent donc d'hériter et de synthétiser l'information de noeud en noeud.

Un exemple d'arbre "décoré" par des attributs est donné dans la figure ci-dessous ; cette figure ne sert qu'à montrer l'intérêt de l'utilisation des attributs pour le traitement des aspects contextuels

(*) Acronyme pour "Production Elégante et RéeLlement aUtomatiquE de TraducTEurs".

d'un langage : l'exemple de grammaire à attributs que nous donnons plus loin est d'une autre nature. Sur cette figure, les non-terminaux noeuds de l'arbre syntaxique sont notés entre les caractères < et >, que nous dénommerons "chevrons", les terminaux - feuilles de l'arbre et éléments du programme - sont notés entre guillemets, et les attributs en *italique* (Certains terminaux, dits génériques, introduisent un niveau supplémentaire dans l'arbre ; leur nom est par convention précédé du caractère pourcent (%).). La figure représente une possibilité de circulation d'un attribut "type" dans l'expression (erronée) "1.0 mod vrai"



Nous ne préciserons pas plus avant cet exemple, qui est suffisamment clair : le type des sous-phrases d'une expression (termes et facteurs) est un attribut synthétisé (qui "remonte" dans l'arbre des feuilles vers la racine - ces termes sont particulièrement mal choisis mais l'usage en veut ainsi) ; cet attribut peut prendre notamment les valeurs *réel* (pour une expression à valeur dans $\mathbb{R}$) et *booléen* (pour les expressions à valeurs dans {faux,vrai}); il peut prendre également la valeur *erreur* (et un message à cet effet sera certainement issu lors de son calcul), comme ici où l'opérateur "modulo" n'a pas été prévu pour un premier opérande du type réel et un second du type booléen.

L'exemple que nous allons développer maintenant a moins de rapport que celui-ci avec l'écriture d'un compilateur ; nous l'avons cependant choisi afin d'introduire de façon simple les attributs hérités (qui "descendent", de la racine vers les feuilles- voir ci-dessus). Sa simplicité va nous permettre de l'exposer en totalité ; il s'agit en fait de la transformation (syntaxique) de formes arithmétiques à la LISP en expressions arithmétiques courantes.

La syntaxe des formes arithmétiques à la LISP acceptées est donnée ci-dessous ; tout opérateur accepte un nombre quelconque d'arguments, supérieur ou égal à deux ; les expressions sont totalement parenthèsées, et on note (opérateur arguments) ces expressions.
Dans la grammaire ci-dessous nous utilisons deux terminaux génériques : %Opérateur et %Atome, qui sont définis, de façon séparée, au niveau lexical, par des expressions régulières : %Opérateur peut prendre les valeurs "+", "-", "*" et "/" et %Atome peut désigner des identificateurs ou des nombres.

Syntaxe :

```
<S-EXPR>    = (%Opérateur  <S-EXPR>  <Opérandes>)
            | %Atome
<Opérandes> = <S-EXPR>
            | <Opérandes>  <S-EXPR>
```

Niveau lexical :

```
%Opérateur = "+" | "-" | "*" | "/"

%Atome     = LETTRE {LETTRE | CHIFFRE}
           | CHIFFRE {CHIFFRE}
```

Les barres verticales (|) permettent d'exprimer des alternatives. Au niveau lexical, les meta-parenthèses { et } encadrent des expressions régulières qui peuvent être rencontrées un nombre quelconque de fois (éventuellement nul), les crochets [et] - non utilisés ici - servent à exprimer la possibilité d'une expression régulière (donc zéro ou une fois). Au niveau syntaxique, nous utilisons des productions récursives - comme dans la définition des opérandes - à gauche car elles sont, dans notre système, destinées à un analyseur ascendant "gauche-droite" qui traite plus simplement ce type de réduction.

Les expressions arithmétiques que nous voulons obtenir se caractérisent par le fait que les opérateurs n'y ont pas de priorité respective et sont binaires notés de manière infixée. Ces expressions seront parenthèsées comme les expressions à la LISP, en raison de l'associativité des opérateurs. Par exemple,

(* (+ a b) (+ c d e))
sera traduit en ((a+b)* (c+d+e)).

Quels attributs allons-nous utiliser ? Un premier attribut semble nécessaire pour synthétiser l'expression engendrée : nous l'appellerons "résultat" ; sa définition ne pose pas de problème pour le non-terminal <S-EXPR> : dans le cas d'un atome, c'est cet atome lui-même, dans l'autre cas il s'agit de la composition par l'opérateur des résultats du premier argument et de la liste des autres opérandes. En revanche, pour le définir dans le cas des opérandes, nous avons besoin de connaître l'opérateur s'appliquant à ces opérandes : un second attribut va nous permettre de réaliser cela ; il s'agira d'un attribut hérité, que nous nommerons "opération", défini lors de la rencontre d'un opérateur ; nous choisissons simplement que sa valeur soit le texte de cet opérateur.

Nous pouvons maintenant donner une définition symbolique de l'attribut résultat pour chacune des règles de la grammaire précédente :

```
résultat (<S-EXPR> = (%Opérateur  <S-EXPR>'  <Operandes>)) =
          "(résultat (<S-EXPR>') %Opérateur résultat (<Operandes>))"

résultat (<S-EXPR> = %Atome) = "%Atome"

résultat (Operandes> = <S-EXPR>) = résultat (<S-EXPR>)

résultat (<Operandes>= <Operandes>' <S-EXPR>) =
          "résultat (<Opérandes>')
              opération <opérandes>  résultat  (<S-EXPR>)"
```

On remarque dans la dernière règle une apostrophe (') à la suite du non-terminal <Operandes> ; c'est la convention que nous avons adoptée pour désigner la deuxième occurence de ce non terminal dans la production syntaxique - la première occurence étant la partie gauche de cette production - ; nous suivons en cela la convention du système DELTA, qui note la n-ième apparition d'un symbole (en comptant de gauche à droite) en faisant suivre ce symbole de n-1 apostrophes. D'autre part, la notation employée pour la définition d'un attribut n'est utilisable que pour les attributs synthétisés, pour lesquels on sait que l'on veut dénoter, en écrivant A (<NT> = ...), l'attribut A du non-terminal <NT>, partie gauche de la production syntaxique ; cela n'est pas possible pour les attributs hérités. Nous allons donc réécrire cette définition en adaptant les conventions du système DELTA, où l'on donne les définitions d'attributs à

la suite de la règle syntaxique à laquelle ils s'appliquent ; nous introduisons en même temps l'attribut standard PTEXT, qui, appliqué au nom d'un terminal, rend son texte tel qu'il apparaît dans le programme traité :

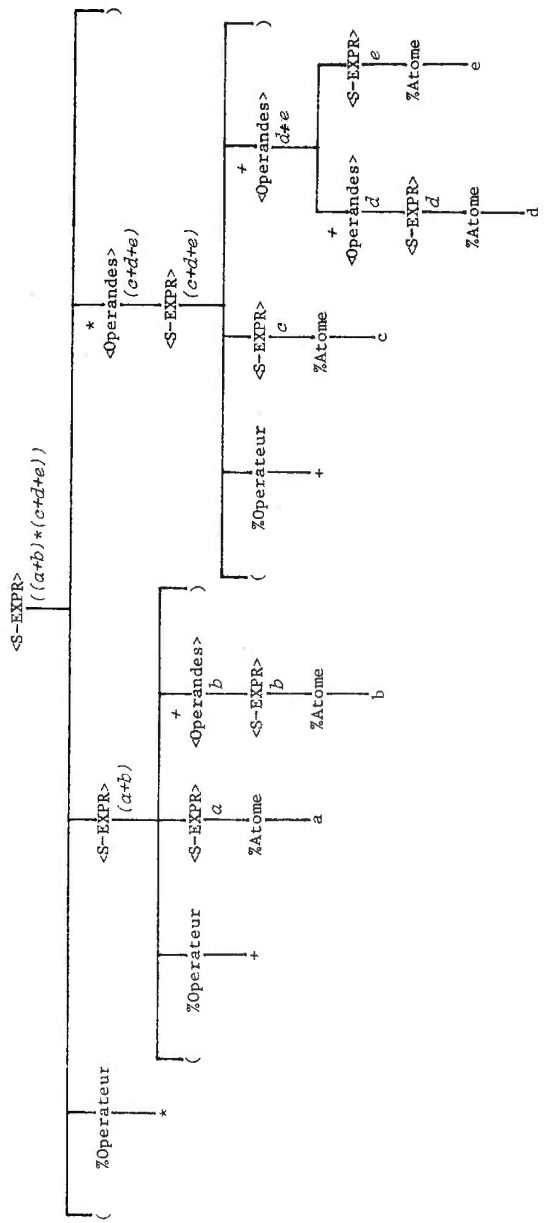
```
<S-EXPR> =  (%Opérateur  <S-EXPR>'   <Operandes>)
   résultat (<S-EXPR>) = "(résultat(<S-EXPR>')
                              PTEXT (%Opérateur) résultat (<Opérandes>))"

<S-EXPR> =  %Atome
   résultat (<S-EXPR>) = "PTEXT (%Atome)"

<Operandes> = <S-EXPR>
   résultat (<Operandes>) = résultat (<S-EXPR>)

<Operandes> = <Operandes>'  <S-EXPR>
   résultat (<Operandes>) = "résultat (<Operandes>')
                                opération (<Opérandes>)résultat (<S-EXPR>)"
```

Nous pouvons maintenant spécifier l'attribut "opération", qui rappelons-le véhicule le texte de l'opérateur en cours de traitement :

```
<S-EXPR> =     (%Opérateur     <S-EXPR>'    <Operandes>)
  opération (<Operandes>) = "PTEXT  (%Operateur)"

<Operandes> = <Operandes>'  <S-EXPR>
  opération (<Operandes>') = opération (<Operandes>)
```

L'attribut opération est ainsi calculé dans la première règle et ensuite transmis sans modification tout au long de la liste des opérandes.

Cette spécification des deux attributs est suffisante pour nous permettre de donner un exemple. On trouvera à la page suivante le schéma de l'arbre syntaxique obtenu en suivant la grammaire que nous avons donnée sur le texte (* (+ a b) (+ c d e)) . Nous avons décoré cet arbre par les valeurs des attributs associés à ses noeuds.



Arbre décoré représentant le texte source (* (+ a b) (+ c d e)) :

- les attributs hérités sont signalés au-dessus et à gauche des non-terminaux,
- les attributs synthétisés étant signalés au-dessous et à droite.

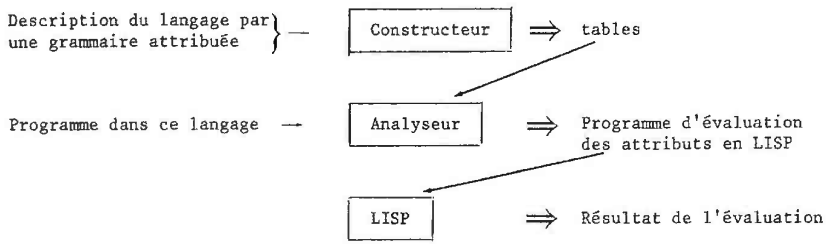
Nous n'avons pas indiqué les dépendances entre attributs pour ne pas alourdir la figure.

Si toutefois nous désirons utiliser le système DELTA pour traiter cet exemple, nous devons exprimer cette spécification différemment: le métalangage que nous venons d'utiliser pour la description des attributs n'est pas connu du système. En fait DELTA n'analyse pas les descriptions d'attributs, ce qui permet d'utiliser le langage qui semble le plus adapté au traitement que l'on désire effectuer. Nous n'entrerons pas dans le détail du fonctionnement de DELTA ; précisons simplement que l'utilisation de DELTA se fait en quatre étapes, la première étant une construction de tables à partir de la description du langage à traiter et des attributs, les trois autres étapes formant l'évaluation sémantique à effectuer pour chaque texte source.

Ces quatre étapes peuvent se définir ainsi :

- construction des tables sémantiques - à partir des définitions d'attributs -, syntaxiques et lexicographiques ;
- construction de l'arbre syntaxique correspondant au programme traité, et détermination des relations de dépendance entre les attributs ;
- tri topologique des attributs, avec élimination des identités et des impasses - une impasse est une chaîne d'attributs non reliée à un attribut de la racine de l'arbre syntaxique -, et production d'un programme d'évaluation des attributs (ce programme est écrit dans le langage choisi pour la description des attributs) ;
- évaluation des attributs, par l'exécution de ce programme.

Comme nous le verrons plus loin, nous avons choisi d'utiliser le langage LISP comme langage de définition des attributs. Nous avons donc réécrit l'exemple précédent pour DELTA au moyen de LISP ; cette description se trouve en annexe D, ainsi que la séquence des commandes utilisées pour transformer le texte vu dans la figure précédente. Nous pouvons schématiser cette séquence par la figure page suivante.



0.4 Plan de cette thèse

Cette introduction nous a permis, après un rappel sur l'historique de la compilation, d'exposer ce que recouvrait le terme de "méta-compilation" et quelles spécifications étaient nécessaires à la définition d'un compilateur ; nous avons ensuite montré que les attributs étaient un bon outil pour le traitement des contraintes contextuelles et le calcul de la valeur sémantique d'un programme - bien qu'ils ne suffisent évidemment pas pour produire un compilateur à partir de sa description formelle -.

Cette thèse est consacrée à la présentation du système Perluette de production de traducteurs ; ce système est fondé sur une sémantique formelle des langages de programmation, décrite dans [Gau 80], et permet l'utilisation d'attributs pour le traitement des contraintes contextuelles.

Le chapitre I est une présentation générale de cette sémantique, ainsi que des méthodes de spécification et de preuve que nous utilisons ; nous y traitons un exemple simple, mais complet. Le chapitre II décrit la structure du système ainsi que celle des traducteurs qu'il produit. Le chapitre III est consacré à la description des modules du système et de leur implantation - par attributs -. Le chapitre IV enfin présente la description détaillée d'un langage de programmation réaliste, qui comporte des procédures et des tableaux d'entiers.

CHAPITRE I

METHODES : PRESENTATION GENERALE

Ce chapitre expose les idées principales qui ont présidé à la définition du méta-compilateur Perluette. Une de ces idées fondamentales a été exposée dans l'introduction : pour définir un compilateur, il est indispensable de spécifier de façon formelle non seulement les langages en présence - langage source et langage objet - mais encore les choix d'implantation des objets du langage source en terme d'objets du langage cible. Notre méthode de description des langages source et objet comportant la spécification d'un type abstrait algébrique pour chacun d'eux, ces choix d'implantation seront exprimés comme la représentation d'un type abstrait algébrique par un autre.

Dans ce chapitre nous allons tout d'abord étudier plus particulièrement la question de la vérification d'un compilateur ; en fait nous allons voir comment la spécification d'un compilateur, telle qu'elle est donnée à Perluette permet de vérifier par ailleurs que les choix de représentation sont corrects. Nous présenterons ensuite notre méthode de spécification au moyen d'un exemple de langage comportant des expressions entières et des tableaux à une dimension. On trouvera une présentation plus détaillée de la méthode dans la thèse de Marie-Claude Gaudel [Gau 80] qui a servi de base théorique à notre travail.

I.1 Principes

Le système Perluette est une application de la théorie de la sémantique des langages de programmation présentée dans la thèse de Marie-Claude Gaudel. Cette théorie permet, en plus de la spécification d'un compilateur en vue de sa réalisation, d'établir une preuve de correction de la méthode de traduction choisie. Nous étudions ici, de manière schématique, comment les éléments de cette preuve sont intégrés à la spécification du compilateur.

Nous reprendrons ici à notre compte le diagramme fondamental de Lockwood Morris ; dans son article sur la structuration et la preuve des compilateurs [Mor 73] , il a introduit une approche dont l'expresssion graphique est maintenant connue et généralement admise. Nous ne nous étendrons pas sur les domaines qu'il utilise ; son approche peut se résumer ainsi [Gau 80]: "Les langages source et objet sont considérés comme des algèbres hétérogènes, dont les ensembles sont les différentes compositions syntaxiques (par exemple, à partir d'un identificateur et d'une expression, on obtient une instruction par l'opération *affectation*). La compilation se décrit comme un homomorphisme du langage source vers le langage objet. A ces langages sont associées des fonctions sémantiques qui, à chaque programme, associent une signification".

L'avantage de cette approche est de restreindre le principe de la preuve de validité du traducteur obtenu par cette méthode : il suffit en effet, pour la réaliser, de démontrer que le diagramme suivant commute, c'est-à-dire que

représentation o $sémantique_s$ = $sémantique_o$ o compilation
(ceci étant l'égalité de deux homomorphismes).

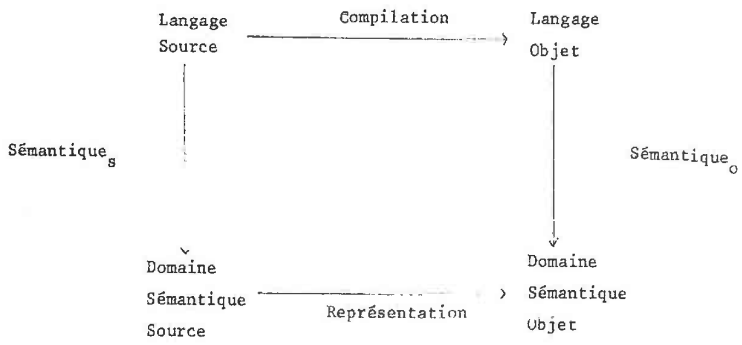


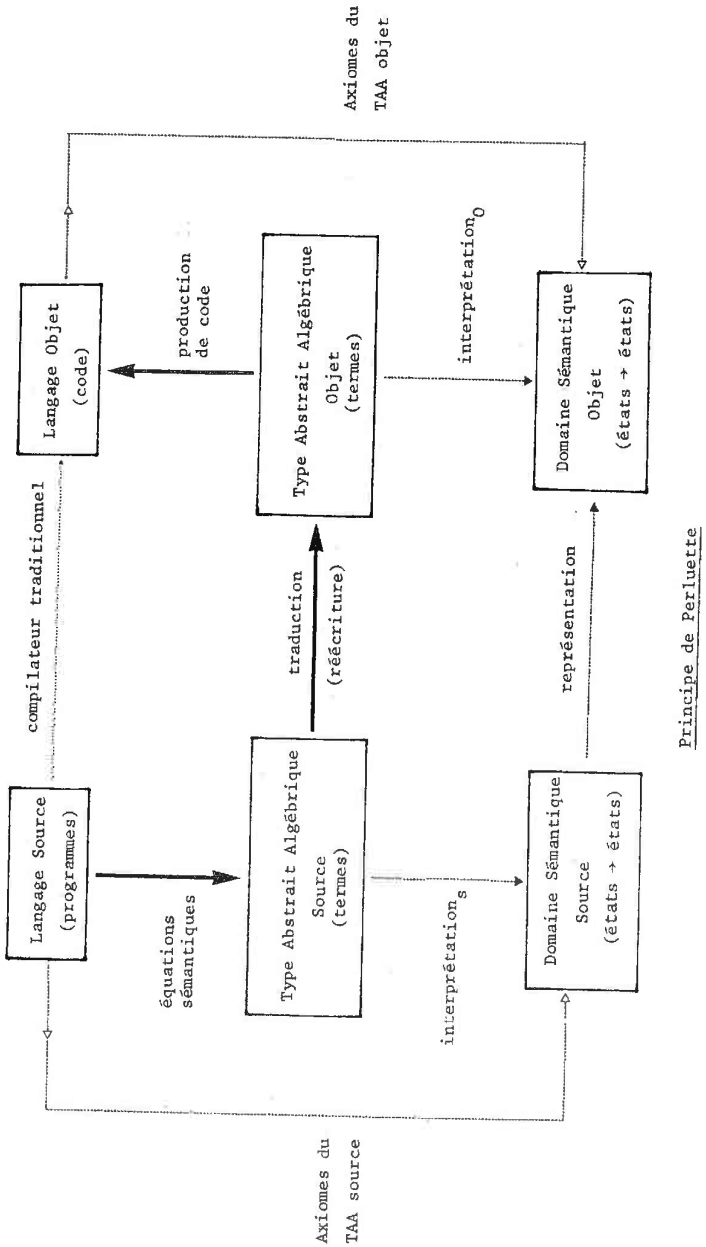
Diagramme de Lockwood Morris

Notre optique dans le système Perluette est de réaliser directement la compilation comme une composition des trois flèches du bas de ce diagramme. Il nous suffit de ce fait de considérer, non pas les valeurs du Domaine Sémantique Source, mais des dénotations de ces valeurs : ces dénotations seront ensuite réécrites sous forme de dénotations de valeurs du Domaine Sémantique Objet. Si le compilateur est correct, les valeurs ainsi dénotées doivent être équivalentes.

Cela signifie, en pratique, que les compilateurs produits par Perluette manipulent des termes et les réécrivent. La preuve d'un compilateur fait intervenir une sémantique de ces termes - ou encore une interprétation (au sens de la Logique) - ; un moyen maintenant classique de spécifier la sémantique d'un tel ensemble de termes est de se donner sur cet ensemble un certain nombre d'axiomes - c'est-à-dire de définir un type abstrait [BG 77,GH 78] - et de considérer l'algèbre initiale associée à ce type abstrait algébrique [GTW 78]. Dans la plupart des cas, les objets de cette algèbre sont des "changements d'états".

En résumé, les propriétés des objets manipulés par un langage sont décrites par des axiomes ; un état est alors un ensemble de prédicats valides qui vérifient ces axiomes et le caractérisent. Notre approche peut se schématiser par un diagramme similaire au précédent ; dans ce diagramme :

- les objets constitutifs de chaque domaine sont indiqués entre parenthèses sous le nom de ce domaine ;
- les flèches continues indiquent une fonction du compilateur ;
- les flèches formées de traits discontinus sont des éléments destinés à la preuve.



I.2. Un exemple

L'objet de nos travaux étant plus pratique que théorique, nous avons choisi de présenter notre approche au moyen d'un exemple simple de spécification d'un compilateur : nous définissons pour cela un langage de programmation comportant des variables entières et des tableaux d'entiers à une dimension ; les bornes inférieure et supérieure de ces tableaux sont données lors de leur déclaration.

I.2.1. Description du langage à traduire

Une syntaxe de ce langage, sous une forme similaire à la BNF, est la suivante :

```
< Programme > =< Liste de déclarations > ";"  < Liste d'instructions > "." ;
< Liste de déclarations > = < Déclaration >
                          | < Liste de déclarations > ";" <Déclaration> ;
< Déclaration > = "entier" %id
                 |"tableau" "[" %entier ".." %entier "]" %id ;
< Liste d'instructions >= < Instruction >
                         |<Liste d'instructions > ";" < Intruction > ;
< Instruction > = < Affectation >
                 | ... ;
< Affectation >=  %id ":=" < Expression >
                 | %id "[" < Expression > "]"  ":="  < Expression > ;
< Expression > =  < Expression > "+" < Terme >
                 | ....
                 | - <Terme > ;
...
< Facteur > = %entier
             |%id
             |%id "[" < Expression > "]"      |   "(" < Expression > ")" ;
...
```

Les instructions classiques sur les tableaux sont donc disponibles dans ce langage : on peut accéder à la valeur d'un élément de tableau ou changer cette valeur. De même, les opérations courantes sur les entiers, telles l'addition ou la multiplication, sont utilisables.

La sémantique de ce langage va être donnée sous la forme d'un type abstrait algébrique ; ce type abstrait - que nous appellerons "Type Source"- nous permettra de définir les axiomes que nous avons mentionnés précédemment et constituera la définition des termes que peuvent produire les "équations sémantiques"- voir le schéma précédent - : le Système Formel Source est en fait constitué de l'ensemble des termes du Type Source.

Le langage que nous définissons est constitué d'entiers, de piles, d'identificateurs et d'instructions - au sens large : nous considérons qu'une déclaration est similaire à une instruction - ; le Type Source comportera donc un nom de sorte pour chacun de ces types d'objets, ainsi que des opérations sur ces sortes.

Intéressons-nous, tout d'abord, au type *Entier* : cinq opérations sont possibles, correspondant aux cinq opérations mathématiques classiques (addition, soustraction, multiplication, division et calcul de l'élément opposé) ; nous le spécifions ainsi :

Type *Entier* :

op (*Entier*, *Entier*) → *Entier* : *Plus*, *Moins*, *Mult*, *Div*. ;
op (*Entier*) → *Entier* : *Opposé* ;

Ceci signifie que les quatre opérations *Plus*, *Moins*, *Mult* et *Div* ont le même profil : à partir de deux *Entiers*, elles rendent un *Entier*, et que l'opération *Opposé*, à partir d'un *Entier*, rend un *Entier*. Ceci est suffisant pour le système Perluette, auquel on ne donnera pas d'autre renseignement sur ces opérations : cette spécification permet en elle-même de construire des termes du type *Entier*, car par convention les constantes de tout type sont dénotées par leur texte entre apostrophes précédé de leur type - ainsi l'élément neutre pour l'addition est noté "*Entier '0'*" -.

Cependant, pour pouvoir prouver que notre compilateur est correct, il nous faut associer une sémantique à ce type *Entier* ; nous le faisons au moyen d'axiomes - supposant ici pour la clarté de l'exposé que nos entiers sont les entiers relatifs mathématiques (non bornés) - ; nous ne donnerons pas cette axiomatisation dans son intégralité, car elle est fastidieuse et de peu d'intérêt ici.

Axiomes

Pour tous i,j dans *Entier* :

Plus (i, Entier '0') = i
Plus (i, Plus (j, Entier '1')) = Plus (Plus (i, j), Entier '1')
Plus (i, j) = Plus (j, i)
...
Mult (i, Plus (j, Entier '1'))= Plus (Mult (i, j), i)
...

Pour que cette spécification soit complète, il nous faut envisager le cas de la "division par zéro" ; nous le faisons par la définition d'un "cas d'échec". Pour cette définition, nous devons ajouter parmi les opérations sur les entiers une opération permettant de tester la nullité d'un entier ; comme l'axiomatisation complète des entiers nécessite une opération d'égalité entre eux, et comme, en outre, le test d'égalité est présent dans presque tous les types abstraits, nous définissons l'opération *Egal*, qui à partir de deux *Entiers*, rend un élément du type prédéfini *Logique* :

op (*Entier, Entier*) → *Logique : Egal* ;

Cette opération s'axiomatise aisément :

Egal (i, Plus (i, Entier '1')) = faux
Egal (Plus (i, Entier '1'), Plus (j, Entier '1')) = Egal (i, j)
Egal (Entier 'A', Entier'B')= (A ≡ B)
...

La division par zéro peut donc être spécifiée :

restriction

Egal (i, Entier '0') ⟹ Echec *(Div ; j,i)*

Ceci signifie que l'opération *Div* n'est pas définie si son second argument vérifie la propriété à gauche de la flèche d'implication. Le traitement de ce cas d'échec devra être présent dans le compilateur, c'est-à-dire que le programme objet ne devra pas s'exécuter au-delà d'une division par zéro.

Enfin, pour être tout-à-fait rigoureux, il serait nécessaire de donner la syntaxe des constantes entières et de lui associer une sémantique faisant correspondre à chaque texte de constante un terme du type Entier ; cela pourrait être réalisé de la manière suivante , similaire à ce qui est fait plus loin pour les expressions :

%entier = < Chiffre > | %entier < Chiffre > ;
< Chiffre > = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;

T : %entier ∪ Chiffre → termes de type *Entier*

T ⦇0⦈ = *zéro*
T ⦇1⦈ = *un*
T ⦇2⦈ = *Plus (un,un)*
...
T ⦇9⦈ = *Plus (T* ⦇8⦈*,un)*
T ⦇%entier Chiffre⦈ = *Plus (T* ⦇Chiffre⦈*, Mult (T* ⦇%entier⦈*,*
Plus (T ⦇9⦈*, un))*

En fait, cette fonction *T* est considérée comme prédéfinie et notée comme nous l'avons signalé précédemment ; ceci permet une meilleure lisibilité des fonctions sémantiques et garantit une plus grande efficacité du compilateur. En outre, le texte des constantes du programme peut ainsi être utilisé lors de leur traduction - voir en I.3 (spécification de la traduction)-.

Une autre opération est également considérée comme prédéfinie : l'égalité sur les textes de constantes, que nous avons notée ci-dessus "≡".

Nous pouvons maintenant définir la partie du type abstrait associée aux identificateurs qui repèrent des entiers - que nous considérons comme différents dans leur essence des indentificateurs de tableau. Ils se spécifient ainsi :

Type *Id-ent* :

op *(Id-ent)* → *Entier : Valeur ;*
op *(Id-ent,Id-ent)* → *Logique : Egal ;*

Comme précédemment, considérant que nous voulons prouver que notre compilateur est correct, nous associons des axiomes à cette syntaxe de type abstrait :

axiomes

Pour tous *id1, id2* dans *Id-ent* :

Egal (id1,id2) ⊃ *Egal(Valeur(id1),Valeur(id2))*
Egal (Id-ent 'A', Id-ent 'B') = (A ≡ B)

Le premier de ces axiomes indique que l'égalité de deux identificateurs implique l'égalité de leurs valeurs entières, le second définit l'égalité de deux identificateurs comme étant l'égalité de leurs textes - il n'y a pas d'autre cas d'égalité d'identificateurs car notre langage ne possède pas de possibilité d' "équivalence" entre identificateurs, ni d' "alias" comme dans certains langages procéduraux.

On peut trouver surprenant que nous utilisions le même nom *Egal* pour des opérations différentes : nous considérons que cela est de peu d'importance, toutes ces opérations ayant des profils similaires (à partir de deux éléments d'un même type, elles rendent un élément du type standard *Logique*), et pouvant être syntaxiquement différenciées par leur domaine ; l'utilisation de ce nom générique simplifie, en outre, la lecture - ainsi que l'écriture - des axiomes où il intervient. Signalons de plus

que ce nom d'opération n'interviendra pas dans les termes construits par le compilateur car son existence ne se justifie que pour la preuve que nous désirons établir.

Estimant le lecteur quelque peu familiarisé avec nos notations à ce point, nous donnons,ci-dessous, la partie du Type Source associée aux identificateurs de tableaux :

Type *Id-tab* :

op *(Id-tab, Entier)* → *Entier : Accès-ième ;*
op *(Id-tab)* → *Entier : Binf, Bsup ;*
op *(Id-tab,Id-tab)* → *Logique : Egal ;*

axiomes

"similaires à ceux de *Id-ent* pour l'opération *Egal* ; en outre :"
pour tout *idt* dans *Id-tab* :

Binf (idt) ≤ Bsup (idt)

Pour ce dernier axiome, la rigueur voudrait que nous définissions sur les *Entiers* une opération *Inf-egal*, à deux opérandes *Entiers* et de résultat *Logique*, au moyen, par exemple , des axiomes suivants :

pour tous *a,b* dans *Entier* :

Egal(a,b) ⊃ *Inf-egal(a,b)*

Inf-egal(a,b) ⊃ *Inf-egal(a,Plus(b,Entier '1'))*

Nous pensons préférable de nous accorder l'utilisation de l'opérateur infixé ≤ pour une meilleure lisibilité.

restriction

Pré *(Accès-ième ; idt,i) = (Binf (idt) ≤ i) ∧ (i ≤ Bsup (idt))*

On voit ici apparaître le second moyen que nous nous donnons de spécifier les cas d'erreur dans un type abstrait : les préconditions. Alors que les échecs - comme dans la spécification de la division entière plus haut - indiquent les cas de terminaison normale de la représentation d'une opération, les préconditions permettent de restreindre les termes utilisables sur le type abstrait en limitant le domaine d'applicabilité des axiomes. Ces deux types de restrictions sont donc de natures différentes : les cas d'échecs doivent être envisagés et traités par l'implanteur du type abstrait, les préconditions, en revanche, étant pour lui vérifiées par hypothèse et à la charge de l'utilisation de ce type abstrait.

Ainsi, dans le cas d'un tableau, le résultat de l'accès à un élément inexistant (en dehors des bornes qui ont été spécifiées) est indéfini : l'utilisateur du type abstrait - le programmeur - doit s'assurer que tout indice qu'il utilise est correct, l'implanteur du type *Id-tab* ayant en revanche le libre choix de son traitement.

Nous disposons maintenant de tous les éléments nécessaires à la définition des termes associés aux expressions de notre langage. Un exemple d'un tel terme est celui associé à l'expression

T[X[I+3]]

qui s'écrit ainsi :

Accès-ième (Id-tab 'T',
Accès-ième (Id-tab 'X',
Plus (Valeur (Id-ent 'I'), Entier '3')))

On peut remarquer sur cet ensemble que nous supposons que l'expression est correcte, c'est-à-dire, que d'une part, elle est précédée de déclarations appropriées - de l'identificateur I comme entier, et de T et X comme tableaux - et que d'autre part, les expressions qui y sont utilisées comme indices le sont à juste titre. La vérification de cette seconde contrainte est, comme nous l'avons dit ci-dessus, sous la responsabilité du programmeur puisqu'il ne s'agit pas d'un cas d'échec ; quant aux contraintes contextuelles, comme nous l'avons signalé au cours de l'introduction, elles sont prises en compte au moyen d'attributs sémantiques lors du traitement du programme à traduire ; nous n'envisageons pas ce traitement au cours de ce chapitre, cette partie du traducteur étant étudiée dans les chapitres suivants.

Pour spécifier la correspondance entre le texte source et les termes du Type Source, nous allons définir une fonction sémantique qui, à partir de chaque catégorie syntaxique de la grammaire du langage donnée plus haut, rend un terme :

V : Expression ∪ ... ∪ Facteur → termes de type *Entier*

V ⟦Expression + Terme⟧ = *Plus* (V ⟦Expression⟧ , V ⟦Terme⟧)

...

V ⟦- Terme⟧ = *Opposé* (V ⟦Terme⟧)

...

V ⟦entier⟧ = *Entier* 'entier'

V ⟦id⟧ = *Valeur* (*Id-ent* 'id')

V ⟦id [Expression]⟧ = *Accès-ième* (*Id-tab* 'id' , V ⟦Expression⟧)

V ⟦(Expression)⟧ = V ⟦Expression⟧

Certaines des opérations utilisées dans ces équations sémantiques n'ont pas été définies au moyen d'axiomes ; il en est ainsi, par exemple, pour *Valeur* et *Accès-ième*.C'est qu'en fait il s'agit là d'opérations que l'on peut qualifier de "modifiables", le but des instructions du langage étant précisément de modifier le résultat de telles opérations. En effet, contrairement à une opération telle que *Plus* qui, appliquée à deux entiers, rend toujours le même entier, l'opération *Valeur* appliquée à un identificateur rendra un résultat dépendant de l'"état" du programme.

Précisons ceci par un exemple : quelle qu'ait été la valeur associée à l'identificateur I, l'effet de l'instruction d'affectation "I:=0" est d'introduire la propriété "*Valeur (Id-ent 'I') = Entier '0'*" et de supprimer toutes les propriétés qui pourraient être en contradiction avec celle-ci.

On voit que les instructions ont en quelque sorte pour effet de changer le type abstrait sur lequel on travaille, puisqu'elles en modifient les axiomes et donc les opérations. Ceci fait du type associé à ces instructions - le type *Modif* spécifié plus loin - le "type d'intérêt" de notre Type Source. On trouvera une présentation théorique complète de

cette propriété dans [Gau 80]. Il nous suffira pour notre exemple de rappeler brièvement et de manière intuitive les points suivants :

- un état est un ensemble de formules valides ne comportant pas de variable libre, formé de toutes les formules que l'on peut déduire de l'ensemble des axiomes du Type Source en utilisant les propriétés de l'égalité ;

- si S est un état et a un axiome, S+a est le plus petit état contenant les formules de S et les formules dérivables de a ;

- on dispose d'une opération sur les états, nommée appl, permettant d'appliquer une modification à un état : appl (m,S) = S' ;

- on dispose d'une modification, dite "substitution généralisée", de nom subst, permettant de rendre compte des modifications d'opérations telles *Valeur* ou *Accès-ième* ; par définition, si λ est une liste de termes et μ un terme compatibles avec le profil de l'opération f, appl (subst (f(λ),μ),S) définit un nouvel état S' construit à partir de S : S' contient toutes les formules de S qui ne font pas intervenir l'opération f - pour éviter toute contradiction -, et l'opération f de S' diffère de l'opération f de S par sa valeur μ au point λ (λ et μ étant "calculés dans S").
  La définition formelle de ce nouvel état est :

$$\text{appl (subst } (f(\lambda),\mu),S) = \{P \mid P[f'/f] \in S + a_f,\}$$

  où a_f, est l'axiome f'(x) = si eg (x,λ) alors μ sinon f (x)
  et f' est un nom d'opération de même profil que f, soumis aux mêmes préconditions et n'appartenant pas au Type Source.

Munis de ces outils, nous sommes en mesure de spécifier la partie du Type Source associée aux instructions de notre langage.

Type *Modif* :

op (*Id-ent, Entier*)	→	*Modif : Affecter* ;
op (*Id-ent, Entier, Entier*)	→	*Modif : Déclarer-tab, Changer-ième*
op (*Modif, Modif*)	→	*Modif : Seq* ;
op ( )	→	*Modif : Continuer* ;

Définitions

pour tous *ide* dans *Id-ent*, *idt* dans *Id-tab*,
i,j,e dans *Entier*, m_1, m_2 dans *Modif* :

Affecter (ide,e)	=	subst (*Valeur(ide)*, *e*) ;
Changer-ième (idt,i,e)	=	subst (*Accès-ième(idt,i)*, *e*) ;
Déclarer-tab (idt,i,j)	=	subst (<*Binf(idt)*, *Bsup(idt)*>, <*i,j*>) ;
appl (*Seq* (m_1,m_2), S)	=	appl (m_2, appl(m_1, S))
appl (*Continuer* (), S)	=	S

Ces définitions peuvent se lire ainsi :

- l'effet de *Affecter (ide,e)* est de modifier l'opération *Valeur* ; ceci est semblable à la sémantique axiomatique de Hoare [Hoa 69] : par exemple à l'instruction "I := I + 1" est associé le terme *Affecter (Id-ent 'I', Plus (Valeur (Id-ent 'I'), Entier '1'))* ; La sémantique en est

  subst (*Valeur (Id-ent 'I'), Plus (Valeur (Id-ent 'I'), Entier '1'))*

  Supposons que l'état S du programme avant cette affectation comporte la formule *Valeur (Id-ent 'I') = Entier '0'*. (f1)
  Pour s'assurer que le nouvel état comporte une formule, on applique la définition de subst à cette formule. Ainsi, pour

  Valeur (Id-ent 'I') = Entier '1' (f2)

  on obtient par substitution *Valeur' (Id-ent 'I') = Entier '1'* (f3)

  L'état intermédiaire est formé par S+$a_{\text{valeur}'}$, où $a_{\text{valeur}'}$ est l'axiome

Valeur' (x) = si eg (x,Id-ent 'I') alors Plus (Valeur (Id-ent 'I'), Entier '1')
sinon Valeur (x)

  Puisque f1 appartient à S, on a également, d'après les axiomes sur *Plus*,

  Plus (Valeur (Id-ent 'I'), Entier '1') = Entier '1'

D'après la définition de +, on a donc dans $S+a_{valeur'}$ la formule f3 et par conséquent f2 appartient bien au nouvel état.

- l'effet de l'opération *Changer-ième* est tout-à-fait similaire ;
- l'opération *Déclarer-tab* permet de donner leurs valeurs aux opérations *Binf* et *Bsup* ; la substitution qui la définit est "simultanée".
- *Seq* est notre opération de concaténation des *Modifs* par composition séquentielle.
- enfin *Continuer* est une opération sans effet sur l'état du programme, qui nous servira pour donner la sémantique des déclarations d'identificateurs repérant des entiers.

Nous pouvons maintenant donner la fonction sémantique associée à un programme :

S : Programme ∪ Liste de Déclaration ∪ ... ∪ Affectation ∪ ...

→ termes de type *Modif*

S ⟦Liste des déclarations ; Liste d'instructions⟧ =

Seq (S⟦Liste de déclarations⟧, S⟦Liste d'instructions⟧) ;

S ⟦Liste de déclarations ; Déclaration⟧ =

Seq (S⟦Liste de déclarations⟧, S⟦Déclaration⟧) ;

S ⟦Liste d'instructions ; Instruction⟧ =

Seq (S⟦Liste d'instructions⟧, S⟦Instruction⟧) ;

S ⟦entier id⟧ = *Continuer ()*

S ⟦tableau [$entier_1$ .. $entier_2$] id⟧ =

Déclarer-tab (Id-tab 'id', V ⟦$entier_1$⟧, *V* ⟦$entier_2$⟧*)*

S ⟦id := Expression⟧ = *Affecter (Id-ent 'id', V* ⟦Expression⟧

S ⟦id [$Expression_1$] := $Expression_2$⟧ =

Changer-ième (Id-tab 'id', V ⟦$Expression_1$⟧, *V* ⟦$Expression_2$⟧*)*

...

La valeur sémantique de tout programme de notre langage est donc un terme de type Modif, obtenu au moyen des définitions que nous venons de donner des fonctions sémantiques S et V. Par exemple, au programme

```
entier I ; tableau [0..1] T ;
I := 2 ;
T [0] := I + 1.
```

on associe le terme suivant :

```
Seq (Seq (
        Continuer ()                                   "la déclaration de I"
      , Déclarer-tab (Id-tab 'T', Entier '0', Entier '1')) "celle de T"
   ,Seq (
        Affecter (Id-ent 'I', Entier '2')              "I:=2"
      ,Changer-ième (Id-tab 'T' , Entier '0'           "T[0]:=..."
                   , Plus (Valeur (Id-ent 'I'), Entier'1'))))"... I+1"
```

La spécification de notre langage source est maintenant complète. Nous allons donc spécifier le langage objet, but de la traduction des programmes sources, avant de décrire la façon dont les termes sources, comme celui que nous venons de donner, doivent être traduits en des termes de ce langage objet.

I.2.2. Le langage objet

Il s'agit d'un langage "de bas niveau", que nous choisissons le plus simple possible : il permet de manipuler des valeurs hexadécimales et des adresses - que nous regroupons dans un type "union" *contenu* -, de charger ces contenus dans un registre et de les ranger à une adresse ; une adresse peut être indexée par une valeur ; les opérations usuelles - addition, soustraction,... - sont disponibles sur les valeurs.

Le Type Objet, type abstrait contenant ces différentes sortes et décrivant ces caractéristiques, est donné ci-dessous ; les opérations d'égalité qui y apparaissent sont, comme précédemment dans le Type Source, nécessaires pour l'écriture des axiomes mais sans existence effective dans le langage.

Type *contenu* = union (*valeur, adresse*) :

op (*valeur,valeur*) → *valeur : plus, moins, mult, div;*
op (*adresse*) → *contenu : cm ;*
op (*adresse, valeur*) → *adresse : index ;*
op (*contenu, contenu*) → *logique : égal ;*

axiomes

pour tous *i,j* dans *valeur*, *a* dans *adresse* :

plus (i, valeur '0') = i
...
index (a, valeur '0') = a
index (index (a,i),j) = index (a,plus (i,j))
...
égal (adresse 'A', adresse 'B') = (A = B)
...

restrictions

égal (j,valeur '0') ⇒ Echec *(div ; i,j)*
égal (a,adresse 'nil') ⇒ Echec *(cm ; a)*
égal (a,adresse 'nil') ⇒ Echec *(index ; a,i)*

fin *contenu* ;

Type *registre* :

op (*registre*) → *contenu : cr*

fin *registre* ;

Type *modif* :

op (*registre, contenu*) → *modif : charger ;*
op (*adresse, contenu*) → *modif : ranger ;*
op (*modif, modif*) → *modif : seq ;*
op ( ) → *modif : nop ;*

définitions

pour tous *r* dans *registre*, *c* dans contenu, *a* dans *adresse*, *m1, m2* dans *modif* :

Charger (r,c) = subst *(cr(r),c)* ;
Ranger (a,c) = subst *(cm(a),c)* ;
appl *(seq(m1,m2)),S)* = appl *(m2,*appl*(m1,S))* ;
appl *(nop (),S)* = S

fin *modif*.

Ce langage est extrêmement simple ; nous l'avons délibérement choisi le plus pauvre possible pour éviter d'avoir à énoncer une longue suite d'axiomes inutiles pour notre propos. En revanche, le fait d'avoir un type *registre* et non plusieurs registres spécialisés introduit une plus grande souplesse lors de la description qui suit de la traduction des termes sources en des termes objets, puisque l'allocation effective de registres peut ainsi être reportée à la phase ultime de production du code à partir du terme objet.

I.2.3. Spécification de la traduction

Nous en venons maintenant au coeur de la description d'un compilateur : la spécification de la réécriture des termes sources en des termes objets. Nous allons ici décrire les choix que nous faisons pour l'implantation des objets du Type Source, choix qui déterminent cette réécriture des termes obtenus à partir des programmes à traduire. Cette description, qui consistue une des originalités de notre méthode, est d'une importance capitale : c'est en effet ici que nous allons spécifier, par exemple, si les éléments d'un même tableau sont implantés de manière contiguë ou par une liste de triplets (indice, valeur non nulle, lien) ; cette spécification n'a pas sa place dans la description du langage source et il est donc naturel qu'elle fasse l'objet d'une description séparée.

Nous avons, pour cet exemple, choisi de représenter les tableaux par des zones contiguës dans la mémoire, leurs éléments étant atteints par une indexation relative à l'élément d'indice nul - élément éventuellemnt fictif ; par conséquent :

- chaque identificateur de tableau est représenté par une adresse constante, dont la valeur est déterminée lors de la phase de traduction ; cette adresse est celle de l'élément de ce tableau d'indice le plus faible ;
- tout identificateur de variable entière est représenté par une adresse également constante, le contenu du mot ainsi adressé étant la valeur couramment associée à cet identificateur.

Nous allons donc spécifer cette représentation ; dans le système Perluette, cette représentation se présente sous la forme d'une liste de rubriques, chaque rubrique étant destinée à donner la représentation d'un type ou d'une opération du Type Source en termes d'éléments du Type Objet.

Dans chaque rubrique associée à un type source est donné le nom du type objet par lequel on le représente ; lorsque le type source comporte des constantes - le type *Entier* par exemple -, la représentation de ces constantes est décrite.

Dans le cas d'une opération, des noms sont donnés à ses opérandes et la représentation de l'opération est décrite au moyen d'opérations

du Type Objet et des représentations de ces opérandes. Par ailleurs, si un opérande est une constante, le texte de cette constante est accessible et peut être manipulé par des fonctions - écrites en LISP, voir les chapitres suivants -.

Nous noterons, par convention, dans la spécification de la représentation les noms des variables et fonctions LISP utilisées en majuscules ; nos commentaires sont donnés entre guillemets ("), chaque fonction LISP étant décrite par un tel commentaire lors de son utilisation.

Représentation

Type *Entier* : *valeur* ; "un entier est représenté par sa valeur hexa-décimale"

repr *Entier* 'N' = *valeur* 'CONVERTIR (*N*)' "la fonction CONVERTIR, à partir de la chaîne de caractères qui dénote un entier décimal, retourne la chaîne de caractères dénotant le même entier en hexadécimal".

fin *Entier* ;

Type *Id-Ent* : *adresse* ; "un identificateur d'entier est représenté par une adresse constante"

repr *Id-ent* 'I' = *adresse* 'ALLOUER-ENTIER (*I*)' "la fonction ALLOUER-ENTIER retourne l'adresse associée à un identificateur d'entier ; la chaîne de caractères qui dénote cet identificateur est recherchée dans la table des identificateurs déjà alloués : si elle s'y trouve, l'adresse correspondante est rendue, sinon elle y est entrée avec une nouvelle adresse, qui est rendue"

fin *Id-ent* ;

Type *Id-tab : adresse ;* "un identificateur de tableau est représenté par l'adresse de son élément d'indice le plus faible".

repr *Id-tab 'T'* = *adresse* 'ALLOUER-TABLEAU (*T*, BINF(*T*), BSUP(*T*))'

"la fonction ALLOUER-TABLEAU fonctionne sur le même principe que ALLOUER-ENTIER ; les deux paramètres supplémentaires sont les bornes du tableau, qui permettent de déterminer la taille de la zone à allouer et de calculer l'adresse de l'élément d'indice zéro ; BINF et BSUP sont des fonctions annexes - voir les implantations de *Binf*, *Bsup* et *Déclarer-tab*."

fin *Id-tab* ;

Type *Modif : modif* "Les instructions du langage source sont naturellement représentées par des instructions du langage objet."

fin *Modif* ;

"Nous donnons maintenant la représentation des opérateurs dans l'ordre où ils ont été décrits dans le Type Source."

repr op *(Entier i, Entier j) → Entier : Plus* =
plus (repr *i*, repr *j*)

"l'opération *Plus* du Type Source est représentée par l'opération du Type Objet *plus* ; les opérandes de *plus* sont les représentations de ceux de *Plus*."

"de même pour *Moins*, *Mult* et *Div*, représentés respectivement par *moins*, *mult* et *div*."

repr op *(Entier i) → Entier : Opposé* =

repr *Moins (Entier '0', i)*

"le langage objet ne possédant pas l'opération correspondante, nous utilisons la représentation de l'opération *Moins*."

repr op *(Id-ent ident) → Entier : Valeur* =

cm (repr *ident*)

"la valeur de l'entier repéré par un identificateur est contenue dans le mot dont l'adresse est associée à cet identificateur."

repr op *(Id-tab idt, Entier i) → Entier : Accès-ième* =

cm (index(adresse 'ELEMENT-0(TEXTE(*idt*))', repr *i))*

"ELEMENT-0 est une fonction chargée de retrouver, à partir du texte d'un identificateur de tableau, l'adresse de son élément - qui peut être fictif - d'indice zéro ; ceci nous évite d'effectuer une soustraction à l'exécution.
TEXTE est une fonction standard de Perluette qui, appliquée à la représentation d'une constante, retourne le texte de cette constante."

"Bien que *Binf* et *Bsup* n'apparaissent pas dans les équations sémantiques, nous donnons leurs représentations utilisées lors de la preuve :"

repr op *(Id-tab 'idt') → Entier : Binf* =

valeur 'BINF(*idt*)'

"BINF est une fonction, écrite en LISP, qui, à partir du texte d'un identificateur de tableau, recherche dans la table BORNES, remplie lors de la représentation de *Declarer-tab*, la valeur de la borne inférieure associée à cet identificateur."

repr op *(Id-tab 'idt')* → *Entier : Bsup* =

valeur 'BSUP (*idt*)'

"similaire à BINF"

"Enfin les opérations associées au type *Modif*:"

repr op *(Id-ent ident, Entier e)* → *Modif : Affecter* =

ranger (repr *ident*, repr *e*)

repr op *(Id-tab 'idt', Entier 'i', Entier 'j')* → *Modif : Déclarer-tab* =

(* GARDER-BORNES (*idt,i,j*) *)

"par convention, ce qui se trouve entre (* et *) est un traitement à effectuer 'à la compilation', c'est-à-dire un traiment d'un aspect statique de la représentation ; ici, nous utilisons la procédure GARDER-BORNES, qui remplit la table BORNES à l'entrée *idt* avec les valeurs qui lui sont passées en arguments ; cette table permet de mémoriser les valeurs des bornes associées à un identificateur de tableau et est utilisée ensuite par BINF, BSUP et ELEMENT-O." "Le terme associé à la représentation de *Déclarer-tab* est une modification sans effet:"

nop ()

repr op *(Id-tab idt, Entier i, Entier e)* → *Modif : Changer-ième* =

ranger (*index*(*adresse* 'ELEMENT-O (TEXTE (*idt*))', repr *i*), repr *e*)

"même commentaire que pour *Accès-ième* ; noter en outre la différence d'en-tête avec *Déclarer-tab* : ici les termes de type *Entier* à considérer ne sont pas forcément des constantes."

repr op *(Modif m1, Modif m2)* → *Modif : Seq* =

seq (repr *m1*, repr *m2*)

repr op *()* → *Modif : Continuer* = *nop ()*

fin de la représentation

Cette spécification de la représentation, bien qu'assez longue, est très facile à écrire et montre bien la manière systématique dont elle est réalisée. Pour terminer la spécification de notre traducteur, il nous resterait à décrire la production de code à partir des termes objets engendrés par la phase du traducteur correspondant à cette représentation ; nous ne le ferons pas ici, en raison de la longueur d'une telle description et de sa simplicité ; en outre, cette description est inutile à notre propos et on en trouvera un exemple au chapitre II (en "II.2.2.2 : la description du code"). Il nous manque également un moyen de construire des termes du Type Source à partir des programmes du langage : comme nous l'avons précisé dans l'introduction et comme on le verra plus loin, cela est réalisé au moyen d'attributs sémantiques ; pour les mêmes raisons nous ne donnerons pas ici de description des attributs nécessaires à notre langage (voir "II.2.1.2. : la grammaire attribuée du langage source" et le chapitre IV). Nous pouvons cependant considérer cette spécification comme suffisamment complète pour notre propos, qui est de montrer comment

onpeut démontrer que la partie la plus importante d'un traducteur - la "représentation" ci-dessus - est correcte. Nous allons donc effectuer - en partie - cette preuve sur notre traducteur.

I.3. Preuve de la représentation

Reprenons le schéma de principe de Perluette - fin de I.1 - : nous venons de définir notre traducteur comme étant la suite :

équations sémantiques - traduction - production de code ;

nous désirons maintenant prouver la validité de la phase "traduction" de ce schéma : nous considérons cette phase comme valide si la représentation, induite des choix d'implantation par les interprétations des Systèmes Formels Source et Objet, est correcte (c'est-à-dire conserve les axiomes qui composent les états - voir [Gau 80] , chapitre VI : Preuves d'Implantation-). Nous rappelons, tout d'abord, le principe du déroulement d'une telle preuve, avant de montrer son application sur la représentation de notre langage.

I.3.1. La méthode de preuve

Elle est formée de trois étapes :

a) On considère, tout d'abord, le type abstrait obtenu en supprimant dans le Type Source les modifications (le type *Modif*) et les opérations modifiables - comme *Valeur* et *Accès-ième* - ; on démontre que la représentation de ce type abstrait est correcte, c'est-à-dire que ses axiomes sont conservés. C'est au cours de cette démonstration que sont utilisées les représentations d'opérations qui n'apparaissent pas dans les équations sémantiques mais sont utilisées dans les axiomes - ainsi *Binf* et *Bsup* sur les *Id-tab* -. Cette démonstration amène à dégager des "invariants de représentation", propriétés que doit vérifier la représentation pour que les états soient correctement représentés - voir ci-dessous.

b) On considère ensuite la représentation des opérationsmodifiables :

- l'égalité sur le domaine de définition d'une opération modifiable doit être représentée par l'égalité dans le Type Objet ;

- la représentation de l'état initial - engendré par les axiomes du type abstrait de a) ci-dessus -doit être "acceptable" : si deux opérations du Type Source sont représentées par la même opération du Type Objet, les domaines de définition de cette opération dans les deux cas doivent être distincts.

On introduit ainsi de nouveaux invariants de représentation : ainsi pour notre exemple où les opérations *Valeur* et *Accès-ième* sont toutes deux représentées par *cm*("contenu mémoire"), les adresses manipulées par *cm* doivent être distinctes dans les deux cas. Cette propriété se formulerait, dans un compilateur classique, de la manière suivante : les adresses allouées pour un tableau et pour une variable sont toujours différentes... ce qui est naturel.

c) On s'intéresse enfin aux modifications :

- la représentation de toute modification doit conserver les invariants de représentation établis en a) et b) ci-dessus ;
- pour toute modification dont la définition est donnée en termes de substitution généralisée (voir I.2.1.), sa représentation doit être équivalente (au sens de l'équivalence dans les états) à cette même substitution sur les représentations ; en d'autres termes, si l'on note ρ la fonction de représentation et m une modification définie par $m = \underline{subst}\ (f\ (\lambda),\mu)$, on doit avoir l'équivalence suivante :

$$\underline{appl}\ (\rho(m),S_1) \simeq \underline{appl}\ (\underline{subst}(\rho(f(\lambda)),\rho(\lambda)),S_1)$$

c'est-à-dire que, S_1 étant une représentation acceptable d'un état S, les deux termes de chaque côté du signe $\simeq$ doivent être des représentations acceptables de l'état $\underline{appl}\ (m,S)$.

- dans le cas plus simple des modifications définies en termes de appl, comme les compositions de modifications, on a simplement à démontrer que, si S_1 est une représentation acceptable d'un état S, alors l'état $\underline{appl}\ (\rho(m),S_1)$ est une représentation acceptable de l'état $\underline{appl}\ (m,S)$.

Ainsi découpée, la preuve d'une implantation est assez simple à établir, malgré la longueur des développements nécessaires. Nous allons le voir sur notre exemple.

I.3.2. Réalisation de la preuve

I.3.2.1. Types et opérations non modifiables

Le Type *Entier* est correctement représenté par le type *valeur* : seuls les noms des opérations sont différents, ainsi que les dénotations des constantes - le cas d'échec sur la division par zéro étant présent dans le Type *valeur* - ; ceci n'est, bien entendu, vérifié que si la fonction CONVERTIR est correcte, ce que nous supposerons établi. Le cas (un peu particulier) de l'opération *Opposé* ne pose aucun problème : on a dans le Type Source *Opposé (i) = Moins (Entier 'O', i)* pour tout *i Entier*, par conséquent la représentation de *Opposé* par celle de *Moins* est parfaitement correcte.

Le type *Id-ent* comporte une seule opération non modifiable, *Egal*, avec l'axiome *Egal (Id-ent 'A', Id-ent 'B')= (A ≡ B)*.
La représentation doit conserver l'égalité : par conséquent, la fonction ALLOUER-ENTIER, qui détermine la représentation des identificateurs d'entiers, doit retourner la même adresse pour deux appels différents avec le même argument (ce qui est le propre de toute fonction) et deux adresses différentes pour deux arguments différents - elle doit être, en d'autres termes, injective : les adresses allouées à deux identificateurs différents doivent être différentes.
De même que ci-dessus pour CONVERTIR, nous supposerons ces propriétés vérifiées.

Le type *Id-tab* se traite de façon similaire : ALLOUER-TABLEAU doit être une fonction injective ; en outre l'axiome d'inégalité entre *Binf* et *Bsup* doit être conservé : BINF et BSUP accédant au tableau BORNES rempli par GARDER-BORNES lors du calcul de la représentation de la modification *Déclarer-tab*, cette démonstration doit se faire sur GARDER-BORNES - ou sur la description par attributs du traitement des contraintes contextuelles, puisque les bornes des tableaux de notre langage sont des constantes entières -.

I.3.2.2. Opérations modifiables

L'opération *Valeur* est représenté par l'opération *cm*, également modifiable. Comme nous l'avons signalé dans le paragraphe de description de la méthode de preuve (I.3.1. ci-dessus), puisque l'opération *cm* représente aussi *Accès-ième*, son domaine de définition doit être différent dans chaque cas. Par conséquent, pour tout entier *e* valide comme indice d'un identificateur de tableau T (répondant à la précondition sur *Accès-ième)* on doit avoir, pour tout identificateur d'entier :

repr *ident* ≠ *index (adresse* 'ELEMENT-O (T)', repr *e)*

De même, pour deux identificateurs de tableaux T et U différents et des entiers *e1* et *e2* valides comme indices, on doit avoir :

index (adresse 'ELEMENT-O (T)', repr *e1)*

≠ *index (adresse* 'ELEMENT-O(U)', repr *e2)*

Ces deux propriétés doivent être démontrées à partir des textes des fonctions annexes ALLOUER-ENT, ALLOUER-TAB et ELEMENT-O, en tenant compte de la précondition sur *Accès-ième*. Nous supposerons, comme précédemment, ces propriétés vérifiées par ailleurs.

La précondition sur *Accès-ième* n'a pas à être envisagée d'autre manière dans la preuve - il en irait autrement si nous avions décrit la propriété qu'elle énonce sous forme d'un cas d'échec.

I.3.2.3. Modifications

Nous devons dans un premier temps vérifier que la représentation de toute modification conserve les invariants de représentation énoncés ci-dessus ; ces invariants portant sur des constantes - les adresses sont calculées par ALLOUER-ENTIER et ELEMENT-O, les entiers sont compris entre BINF et BSUP pour chaque tableau-, les modifications ne peuvent les changer ; s'ils sont vérifiés dans l'état initial - ce qui se démontre sur le texte des procédures et fonctions annexes d'allocation et de calculs à la compilation -, ils sont donc vérifiés dans tout état obtenu à partir de cet état initial.

Nous allons maintenant, pour chaque modification élémentaire - une modification élémentaire est une modification sans opérande de type *Modif* -, démontrer que sa représentation effectue bien les substitutions par

lesquelles elle a été définie et est sans autre effet sur l'état.

. Vérification de *Affecter*

La modification *Affecter* est définie par

Affecter (ide, e) = subst *(Valeur (ide),e)*

et représentée par *ranger* (repr *ide*, repr *e*).

D'après la définition de *ranger*, ceci est équivalent à

subst (*cm* (repr *ide*),repr *e*)

c'est-à-dire subst (repr *Valeur (ide)*, repr *e*).

. Vérification de *Changer-ième*

La définition en est subst *(Accès-ième(idt,i),e)*,
la représentation
ranger (index (adresse 'ELEMENT-O (TEXTE(idt))', repr *i*, repr *e)*
Abrègeant l'adresse qui intervient ici en *A-O(idt)*, il vient,
d'après la définition de *ranger* :
subst (*cm (index (A-O(idt)*, repr *i)*, repr *e)*)
c'est-à-dire subst (repr *Accès-ième (idt, i,* repr *e)*.
La substitution voulue est donc effectuée. En vertu des invariants de représentation, elle est seule effectuée - puisque l'adresse *index (A-O(idt)*, repr *i)* ne peut intervenir dans la représentation d'aucune autre opération que *Accès-ième (idt,i)* d'après I.3.2.2. *Changer-ième* est donc correctement représentée.

. Vérification de *Déclarer-tab*

La définition de cette modification est :

Déclarer-tab (Id-tab 'T', Entier 'I', entier 'J')
= subst *(<Binf (Id-tab 'T'), Bsup (Id-tab 'T')>,*
<Entier 'I', Entier 'J'>)

Sa représentation est :

(* GARDER-BORNES (T,I,J)*) *nop ()*

Cette représentation fait intervenir des traitements à effectuer "à la traduction". Sa validité est donc subordonnée à la validité de ces traitements : il nous faut donc déterminer en quoi doivent consister ces traitement pour que la représentation de *Déclarer-tab* soit correcte.

D'après ce qui a été dit en I.3.1.c et en utilisant les représentations de *Binf*, *Bsup* et *Entier*, la représentation de *Déclarer-tab* est correcte si

appl (repr *Déclarer-tab (Id-tab 'T', Entier 'I', entier 'J')*,S)
~ appl (subst (< *valeur 'BINF(T)'*, *valeur 'BSUP(T)'*>,
< *valeur 'CONVERTIR(I)'*, *valeur'CONVERTIR(J)'*>),S)

Il s'agit donc d'associer des constantes à des constantes par l'intermédiaire des fonctions annexes BINF et BSUP. Par conséquent, la représentation de *Déclarer-tab* est correcte si l'on a

GARDER-BORNES (T,i,j)= subst (<*valeur* 'BINF(T)', *valeur* 'BSUP(T)'>,<*i,j*>)

ce qui est donc une propriété à démontrer sur les textes de ces trois procédures.

La vérification de la représentation de *Continuer* par *nop* est immédiate, ainsi que celle de la modification composée *Seq* par *seq*. Ceci conclut donc la preuve de l'implantation de notre langage. On voit qu'une telle preuve, malgré sa longueur, est relativement simple à établir - mis à part le cas particulier des traitements statiques que l'on peut considérer comme des optimisations et dont la complexité est inhérente à leur nature - ; un système automatique d'aide à la démonstration serait néanmoins indispensable pour établir la preuve d'implantations de langages plus complexes.

Pour résumer ce chapitre de présentation de nos méthodes, disons que selon nous la description complète en vue de son élaboration par un système de méta-compilation et de sa preuve se compose de quatre parties :

- description du Langage Source, comprenant la définition d'un type abstrait algébrique, le Type Source, et des équations sémantiques qui permettent de calculer le terme associé à tout programme du langage ;

- description du Langage Objet par le Type Objet et des fonctions de calcul du code objet à partir des termes de ce Type Objet ;
- description de l'implantation des objets du Type Source par des objets du Type Objet, permettant de réaliser la traduction comme une réécriture de termes ;
- si l'on désire montrer que cette traduction est valide, il est nécessaire de donner les axiomes associés aux Types Source et Objet, ainsi que l'implantation des opérations du Type Source qui, bien que n'intervenant pas dans les équations sémantiques, sont nécessaires à l'axiomatisation. La traduction est alors prouvée correcte comme la représentation d'un type abstrait algébrique par un autre.

Nous renvoyons le lecteur au schéma qui se trouve à la fin du paragraphe un de ce chapitre pour une expression graphique de notre approche.

CHAPITRE II

UTILISATION DU SYSTEME

Le but du présent chapitre est de présenter le système Perluette tel qu'il doit être perçu par ses utilisateurs.

L'utilisation de tout système de méta-compilation s'effectue en deux étapes distinctes : production d'un traducteur à partir de sa description, puis utilisation du traducteur obtenu. La première étape, indépendante des programmes à traduire, consiste en général en une construction de tables destinées à être utilisées par le traducteur ; la forme que prennent ces tables est imposée par l'implantation choisie pour le programme traducteur.

Ce chapitre se compose donc de deux parties. Dans la première, nous exposons la structure et le mode de fonctionnement de tout traducteur produit par l'intermédiaire de Perluette ; ceci nous permet d'introduire le format choisi pour les différentes tables construites par le système lors de la première étape. Nous présentons ensuite la nature et la forme des données acceptées par Perluette pour produire ces tables.

II.1 Structure et fonctionnement du traducteur produit

II.1.1 Structure générale

Nous avons montré dans le chapitre I que la conception d'un traducteur nécessitait l'élaboration des trois définitions suivantes :

- le langage à traduire (ou langage source),
- le langage vers lequel on traduit (langage objet),
- les choix d'implantation, en termes du langage objet, des objets manipulés par le langage source.

Il est naturel que ceci soit reflété par la structuration du traducteur : celui-ci effectue donc la traduction suivant les trois phases ci-dessous :

- analyse du programme écrit en langage source, et détermination des objets qu'il manipule ;
- implantation de ces objets en termes du langage objet ;
- production du résultat de la traduction : en général un texte, ou phrase du langage objet - le "code".

Ayant choisi d'utiliser les types abstraits pour la description des langages source et objet, nous avons décidé que les informations transmises entre ces phases de traduction seraient les termes représentant la sémantique du programme en cours de traduction. Le résultat de la première phase est donc un terme du type abstrait associé au langage source, que la seconde phase traduit en un terme du type abstrait associé au langage objet. L'équivalence entre ces deux termes est assurée par la validité des choix d'implantation effectués.

Nous verrons plus loin comment sont implantés ces termes des types abstraits : nous les considérerons pour l'instant comme étant des arbres comportant aux noeuds des noms d'opérations, les arcs étant des relations "premier opérande",..., les feuilles étant des noms d'opérations zéro-aires.

II.1.2 La première phase

Le principe de cette phase est commun à de nombreux traducteurs conventionnels ; elle a le double rôle de vérifier l'appartenance du programme source au langage source et de le traduire en une forme interne plus aisément manipulable que son texte.

Nous venons de décrire brièvement la forme interne qui a été adoptée : il s'agit de l'arbre représentant un terme du type abstrait source qui peut être associé au programme. Remarquons que les feuilles de cet arbre sont des constantes, dont le texte est, pour la plupart d'entre elles, contenu dans le programme source (exemples : les indentificateurs, les numéraux, les chaînes de caractères).

Le second rôle de cette première phase est un rôle de vérification syntaxique, en comprenant sous ce vocable les notions de "grammaire à contexte libre" et de "contraintes contextuelles" (structure de blocs pour l'utilisation des identificateurs, compatibilité entre les opérandes d'une expression, etc.). Si l'on se réfère à la terminologie courante, on peut dire que ce rôle de vérification syntaxique pose tout à la fois un problème d'analyse lexicale et syntaxique et un problème de traitement de la sémantique statique. Ces deux questions pouvaient être résolues par l'emploi de diverses méthodes ; comme nous disposions du système Delta, système de traitement d'attributs sémantiques, associé au système Syntax d'analyse lexicale et syntaxique, nous avons choisi de l'utiliser au cours de la première phase de tout traducteur. Ceci permet d'utiliser un analyseur syntaxique très performant couplé à un module de traitement d'attributs sémantiques, ce qui fournit à l'utilisateur de Perluette un moyen commode de traiter les contraintes contextuelles.

Bien entendu, cette phase sera seule exécutée si le programme traité n'est pas correct : le processus de traduction n'ira pas au-delà si le programme n'appartient pas au langage source, que ce soit par sa syntaxe ou à cause de manquements aux règles contextuelles. La deuxième phase est assurée de n'avoir à traiter que des termes sources corrects, ce qui justifie ce que nous allons exposer.

II.1.3 La deuxième phase

Son rôle est de produire, à partir du "texte intermédiaire" résultant de la première phase, un "texte intermédiaire" qui sera la donnée de la troisième phase. Le premier représente un terme du type abstrait associé au langage source, le second un terme du type abstrait associé au langage objet.

Il s'agit donc ici de la phase de traduction véritable, où interviennent les choix de représentation qui ont été faits lors de la conception du traducteur. Ces choix ayant été faits indépendamment du contexte dans lequel apparaissent les opérations, la traduction peut se

faire noeud à noeud et feuille à feuille dans l'arbre ; il s'agit donc en fait d'un parcours de l'arbre "en postfixé", où un noeud n'est traité qu'après que tous ses descendants l'aient été. Ce processus est similaire à un appel de fonction, dont on évalue tous les arguments avant de l'appeler - appel par valeur - l'évaluation d'un argument pouvant faire appel à une fonction. Nous avons choisi pour cette étape de traduction, en raison de cette similitude, de considérer le premier texte intermédiaire comme un programme écrit dans un langage applicatif ; chaque sous-arbre est un appel de fonction, dont le résultat est ce sous-arbre traduit. Le nom de la fonction est donné par le noeud racine du sous-arbre, ses arguments étant les sous-arbres descendant de ce noeud. Les feuilles de l'arbre, qui sont les opérations zéro-aires du langage, sont traitées de la même manière, avec la différence que les paramètres de ces opérations n'ont pas à être évalués - les constantes sont en effet paramétrées par leur texte, suivant la remarque faite en II.1.2.

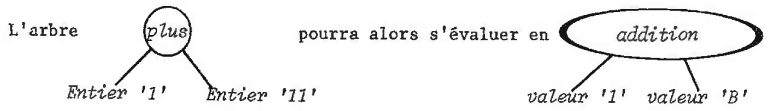
Cette traduction est par conséquent accomplie par interprétation de l'arbre engendré par la première phase, l'évaluation d'un sous-arbre ayant pour résultat la représentation, en termes du type abstrait objet, de l'opération associée à ce sous-arbre.

Par exemple, supposons que le type abstrait source comporte les déclarations suivantes :

```
type Entier ;
   op (Entier, Entier) → Entier : Plus ;
```

alors que le type abstrait associé au langage objet manipule des valeurs hexa-décimales :

```
type valeur ;
   op (valeur, valeur) → valeur : addition ;
```

Ainsi, à partir du texte source "1 + 11", la première phase du traducteur produira le terme "*Plus (Entier '1', Entier '11')*" (représenté par le premier arbre ci-dessus), terme qui s'évaluera en "*addition (valeur '1', valeur 'B')*" au cours de la deuxième phase (Ce second terme, représenté par le deuxième arbre, sera lui-même évalué au cours de la troisième phase, et produira le texte "charger '1'X ; ajouter 'B'X", ou encore "charger 'C'X", ou tout autre chose suivant le degré d'optimisation du producteur de code et le contexte).

Cette "interprétation d'arbre" peut être réalisée de manière très simple par l'emploi d'un langage tel que LISP. En effet, la structure d'"expression symbolique" (S-expression) de LISP permet une manipulation aisée de la structure d'arbre (un terme d'un type abstrait est facilement exprimable sous la forme d'une S-expression) ; la récursivité "naturelle" de LISP permet d'exprimer aisément l'interprétation - telle que nous venons de la définir - d'un tel arbre en fonction de ses sous-arbres ; enfin, comme l'annoncent ses auteurs, "LISP peut interpréter et exécuter des programmes écrits sous la forme de S-expressions et l'exécution d'un programme peut donc se poursuivre par l'exécution du programme qu'il a engendré".

Les deux arbres représentant les termes manipulés par la deuxième phase (terme source et terme objet) sont donc implantés par des S-expressions ; l'interprétation de la S-expression associée au terme source produit une S-expression ; celle-ci est l'implantation du terme objet associé au terme source, implantation définie par les choix de représentation effectués par l'utilisateur de Perluette.

II.1.4 La troisième phase

La troisième phase d'un traducteur produit par Perluette est la phase, bien connue, dite "de production de code". A partir de l'arbre abstrait résultat de la phase précédente, un programme du langage objet est créé ; c'est au cours de cette phase que se font la plupart des optimisations habituelles - allocation de registres, suppression des sauts "en cascade", etc.

Le fonctionnement de cette phase est similaire à celui de la deuxième phase, que nous venons de décrire : cette deuxième phase produit une "S-expression", terme du type objet ; la troisième phase, par l'évaluation de cette "S-expression", produit un texte du langage objet auquel elle peut être associée.

II.1.5 Vue d'ensemble

Pour résumer ce qui précède, le fonctionnement de tout traducteur produit par Perluette se fait en trois phases :

- vérification de l'appartenance du programme source au langage source, et production d'un arbre abstrait dont les noeuds et les feuilles sont des opérations du type abstrait associé au langage source ;
- "interprétation" de cet arbre, l'évaluation d'un sous-arbre ayant pour racine une opération étant la représentation de cette opération en termes du type abstrait associé au langage objet ; cette phase produit donc un arbre similaire au précédent, entièrement formé de termes du type abstrait objet ;
- "production de code", par interprétation de l'arbre ci-dessus.

Les vérifications contextuelles et l'analyse du texte source nécessaires à la première phase de la traduction peuvent être traitées par un système d'attributs sémantiques couplé à un analyseur syntaxique - tel le système Delta.

Les arbres représentant les termes des types abstraits source et objet manipulés peuvent être implantés par des "expressions symboliques" et interprétés comme des programmes en vue de leur traduction - voir le langage LISP et les S-expressions.

II.2 Méthode d'utilisation de Perluette

Cette partie est consacrée à l'utilisation du méta-compilateur. Elle n'est pas un manuel d'utilisation de Perluette, mais elle est conçue comme une introduction à ce système pour toute personne désireuse de construire un traducteur par son intermédiaire.

Nous présentons donc ici les principes qui président à la construction des données destinées à décrire un traducteur pour Perluette.

Une remarque tout d'abord : comme le laissait entendre tout ce que nous avons vu jusqu'ici, ces données sont divisées en trois groupes : la description du langage source, la description du langage objet, les choix d'implantation des termes du type abstrait associé au langage source en fonction de ceux du type abstrait associé au langage objet. Ces descriptions ne sont pas soumises à Perluette dans un ordre quelconque, pas plus qu'elles ne sont conçues indépendamment : il faut connaître parfaitement les langages source et objet - et les types abstraits associés - pour pouvoir décider de l'implantation. C'est pourquoi il est nécessaire de fournir à Perluette tout d'abord les descriptions des deux langages - et là l'ordre est indifférent, puisque ces descriptions n'interagissent pas - avant de lui décrire les choix d'implantation que l'on effectue.

La présente description de l'utilisation de notre système est par conséquent subdivisée en : langage source, langage objet puis choix d'implantation.

II.2.1 Le langage source

Ainsi qu'il a été dit dans le chapitre I, l'idée de base du système Perluette est de formaliser la compilation comme la représentation d'un type abstrait par un autre. La définition du langage source, comme celle du langage objet, comporte par conséquent la description d'un type de données abstrait qui lui est associé.

D'autre part, comme on l'a vu en II.2 le traitement du programme source pose un certain nombre de problèmes - analyses lexicale et syntaxique, détection et correction d'erreurs contextuelles - que nous résolvons par l'emploi d'attributs sémantiques. La description de ces attributs doit donc être faite par l'"écrivain" de compilateur utilisateur de Perluette.

Enfin, les équations sémantiques associées aux catégories syntaxiques du langage - voir le chapitre I - doivent elles aussi être définies. Elles s'intègrent de façon naturelle dans les attributs sémantiques, et sont donc décrites comme tels.

La définition du langage source est donc donnée à Perluette sous la forme d'un type de données abstrait, définissant les types d'objets manipulés par ce langage et les opérations sur ces types, et d'une grammaire à attributs sémantiques, définissant la syntaxe du langage, les vérifications contextuelles et les équations sémantiques associées à chaque règle de grammaire de cette syntaxe.

II.2.1.1 Le type de données source

Il s'agit d'une liste de "descriptions syntaxiques" de types abstraits : pour chaque type d'objet manipulé par le langage est donnée une liste d'opérations dont le résultat est un objet de ce type ou dont un opérande est de ce type.

Ces descriptions sont purement syntaxiques : aucune définition sémantique des opérations déclarées n'est donnée à Perluette. Les axiomes et préconditions habituellement associés aux types abstraits ne servent pas au système pour construire le traducteur : ils ne sont utilisés que lors de la preuve de ce traducteur, pour démontrer que les choix d'implantation sont corrects - c'est-à-dire que la représentation d'un type abstrait source en termes de types abstraits du langage objet conserve les axiomes du type source.

On trouvera en annexe une syntaxe - sous forme de grammaire en BNF - des types abstraits acceptés par Perluette. Voici à titre d'exemple un extrait du type abstrait associé au langage SIMPROC - langage qui est étudié en détail au chapitre IV.


```
type Entier :
     op (Entier)          → Entier : opposé ;
     op (Entier, Entier)  → Entier : plus, moins, mult, div
     fin Entier ;

type Identificateur = union (Id-val, Id-var, Id-tab, Id-proc) :
     op (Identificateur) → Id-proc : père ;
     op (Id-proc)        → Id-val : param-val ;
     op (Id-proc)        → Id-var : param-var ;
     op (Id-tab)         → Entier : borne-sup
     fin Ident ;

type Appel :
     op ( )      → Appel   : appel-en-cours ;
     op (Appel)  → Appel   : appelant ;
     op (Appel)  → Id-proc : nom
     fin Appel ;

...
```

Exemple de type abstrait :
le langage PROCSIM

Dans cet exemple sont déclarés trois types abstraits : les types *Entier*, *Identificateur* et *Appel*, ainsi que des opérations les manipulant. Ces déclarations sont, comme nous l'avons dit, purement syntaxiques : tout ce que l'on connait de l'opération *opposé*, par exemple, est son profil : *(Entier)* → *Entier*.

Cet exemple amène deux commentaires :

- certaines opérations peuvent avoir un domaine vide - ici l'opération *appel-en-cours* - ; il s'agit alors de variables ou de constantes sur le type qui est leur co-domaine ;

- les unions de types simples sont des abréviations permises par le système. Elles peuvent être très utiles pour la description d'objets pour lesquels les traitements possibles sont très proches : il est notamment très agréable, dans cet exemple, de considérer que les identificateurs d'entiers, de tableaux, de procédures, ..., sont regroupés en un type "*Identificateur*", tout en gardant une identité propre. Ceci permet de n'avoir qu'une seule opération *père* dans ce type abstrait, cette opération permettant de désigner la procédure dans laquelle un identificateur est déclaré, que ce soit une procédure ou un entier ; il est à noter que cela n'est pas équivalent à faire de *Identificateur* un type simple, car ici l'axiomatique de l'opération *param-val* par exemple n'aura pas à prendre en compte le cas où son paramètre serait autre chose qu'un identificateur de procédure.

La donnée du type abstrait associé au langage source est destinée à permettre à Perluette de vérifier la validité - syntaxique - des équations sémantiques : on s'assure en effet, dans la mesure du possible, que ces équations sémantiques décrivent des termes du Type Source, c'est-à-dire que les opérandes de toute opération ont des types correspondant au profil de cette opération.

Comme nous l'avons signalé plus haut - voir en II.1.2 -, les équations sémantiques sont incorporées dans une grammaire à attributs, que nous allons maintenant étudier.

II.2.1.2 La grammaire attribuée du langage source

Un traducteur classique peut être spécifié, en général, au moyen de trois attributs principaux :

- un attribut synthétisé qui construit la "table des symboles" (ou table des identificateurs) du programme à partir de leurs déclarations et de leurs occurences dans les instructions ;

- un attribut hérité véhiculant cette table de la racine vers les instructions ;
- un attribut synthétisé décrivant le "code" produit - sa valeur en un noeud étant une composition du code associé aux noeuds fils qui dépend de l'attribut précédent.

En plus de ces attributs principaux, sont définis des attributs "annexes", permettant de traiter par exemple le type d'une sous-expression, la vérification de la bonne utilisation des identificateurs ou l'état d'occupation des registres.

Ceci se retrouve en partie dans la grammaire à attributs destinée à Perluette ; la différence réside dans le fait que le but n'est pas de produire du code, mais un terme du type abstrait associé au langage. L'attribut mentionné ci-dessus décrivant le code est remplacé par un attribut qui construit ce terme à l'aide des équations sémantiques présentées au chapitre I.

De façon à simplifier cette construction du terme source pour l'utilisateur, le système Perluette est conçu pour accepter deux sortes de définitions d'attributs : les attributs habituels, tels ceux décrits ci-dessus - table des symboles, détermination du type d'une expression, ... - et les attributs liés aux équations sémantiques qui dépendent non seulement du langage, mais aussi du type abstrait qui lui a été associé.

Le premier type d'attribut a été présenté dans le chapitre d'introduction. Les informations véhiculées par de tels attributs sont plus syntaxiques que sémantiques - ils pourraient quasiment être omis si tous les programmes étaient corrects et utilisaient un jeu d'identificateurs standard. Nous ne les étudierons donc pas plus avant ; on pourra se reporter au chapitre IV pour des exemples de leur utilisation.

Nous allons maintenant nous intéresser aux attributs qui représentent les équations sémantiques du langage. Ces attributs, qui permettent lors de la première phase de la traduction de construire le terme du type abstrait que l'on associe au programme à traduire, sont en général purement synthétisés - c'est-à-dire que le terme associé à une phrase du

langage ne dépend que des termes associés à ses sous-phrases et d'informations contextuelles. De plus, ils ne tiennent pas compte d'éventuelles erreurs contextuelles car le terme source ne sera utilisé que si le programme est correct - voir la définition de SIMPROC en annexe : un attribut de synchronisation nous permet d'"arrêter" la première phase de traduction avant la génération de ce terme en cas d'erreur contextuelle.

Ces attributs de description des équations sémantiques sont caractérisés par le fait qu'ils sont les seuls à prendre en compte le type abstrait source. Un "métalangage de description d'attributs" a été défini pour simplifier leur écriture et assurer leur "validité syntaxique" vis-à-vis du type abstrait. Dans les cas simples, ce métalangage permet de reprendre pour la définition de ces attributs l'expression exacte des équations sémantiques qu'ils représentent.

Ainsi, si nous reprenons l'exemple de type abstrait présenté précédemment (en II.2.1.1), considérant qu'il est associé à un langage comportant des expressions entières, nous pourrions écrire l'équation sémantique suivante :

Valeur : Expression ∪ Terme ∪ ... → terme de type *Entier*
Valeur ⟦Expression + Terme⟧ = *Plus* (Valeur ⟦Expression⟧ , Valeur ⟦Terme⟧)

Ceci s'écrira, sous la forme d'une grammaire attribuée (avec les notations habituelles de Delta - voir par exemple l'annexe D - :

```
S ≠ Valeur (<Expression>,<Terme>,...) : Entier
<Expression> = <Expression> + <Terme> ;
  ≠ Valeur (<Expression>)
        = Plus (Valeur (<Expression>'), Valeur (<Terme>))
```


Ce n'est guère impressionant ; nous allons cependant anticiper sur le chapitre suivant qui décrit en détail la réalisation de Perluette et notamment le traitement subi par les attributs, et montrer la portion de grammaire qui est déduite de la précédente - comme signalé précédemment, c'est le langage LISP qui a été choisi comme étant le plus apte à manipuler les termes -. La description de l'attribut Valeur devient :

```
≠ Valeur (<Expression>)
    (SETQ Valeur (<Expression>)
          (LIST (QUOTE Plus)
                Valeur (<Expression>')  Valeur (<Terme>) ))
```

Le passage de la première description - faite par l'utilisateur pour Perluette - à la seconde - destinée à Delta - permet de vérifier que l'opération *Plus* possède bien le profil *(Entier, Entier)* → *Entier* donné dans le type abstrait.

Le métalangage que nous avons défini présente d'autres caractéristiques, notamment la possibilité d'utilisation de LISP - dangereuse - et surtout la faculté de "calculer" le nom de l'opération qui sera le noeud du terme en cours de construction. Cette dernière caractéristique s'expose plus facilement sur un exemple.

Reprenons donc l'exemple précédent d'une expression d'arithmétique entière ; les traitements contextuels - non présentés ici - sont les mêmes pour les deux opérations additives, *Plus* et *Moins*. Nous désirons donc, pour plus de concision dans la grammaire, regrouper leurs expressions syntaxiques en une seule règle, de la forme :

<Expression> = <Expression> <plus-moins> <Terme>.

Tous les attributs seront donc communs aux deux cas ; un problème se pose alors pour engendrer le terme correspondant à une expression ; ceci est prévu par le métalangage qui permet de synthétiser le nom de la racine de ce terme dans un attribut déclaré avec le profil adéquat - dans cet exemple, l'attribut Opérateur - :

```
S # Opérateur (<plus-moins>) : op (Entier, Entier) → Entier
S # Valeur (<Expression>, <Terme>, ...) : Entier
<Expression> = <Expression> <plus-moins> <Terme>  ;
  # Valeur (<Expression>)
      = opération calculée Opérateur (<plus-moins>)
                           (Valeur (<Expression>'), Valeur (<Terme>))
<plus-moins> = + ;
    # Opérateur (<plus-moins>)
          = Plus
<plus-moins> = - ;
    # Opérateur (<plus-moins>)
          = Moins
```

Le mot-clé opération calculée employé dans cet exemple permet, de façon générale, l'emploi de toute forme LISP destinée à rendre le nom de l'opération en cours de construction. Il est conseillé de restreindre son emploi au cas que nous venons de voir car le système Perluette, ne faisant pas d'hypothèses restrictives sur la sémantique des formes LISP, ne peut plus vérifier la validité de l'utilisation du nom d'une opération lors de l'emploi de telles formes. L'exemple suivant est "faux" de façon manifeste pour tout lecteur associant une sémantique classique à la forme COND, mais puisque COND peut être redéfini au choix de l'utilisateur de Perluette (ainsi que EQ, QUOTE,...) - mauvaise pratique ! - le système engendrera tout de même le calcul du terme :


```
# Valeur (<Expression>)
    = opération calculée (COND ((EQ Text (<plus-moins>) (QUOTE +))
                               (QUOTE Plus))
                           (T (QUOTE Opposé)))
                  (Valeur (<Expression>'), Valeur (<Terme>))
```

On trouvera en annexe une syntaxe en BNF du métalangage que nous avons défini pour les équations sémantiques. En plus de ce que nous venons d'exposer, s'y trouve la définition des formes LISP acceptées, définition incluant la possibilité d'utiliser une fermeture multiple de parenthèses par un chevron (>) et réciproquement. Cette syntaxe est, à peu de choses près, celle utilisée dans le système LISP de A. LUX, qui a été le premier LISP employé pour Perluette. Les formes :

```
                <cond ((eq a b) (plus c (plus d e>
et              (cond <(eq a b) <plus c <plus d e)
```

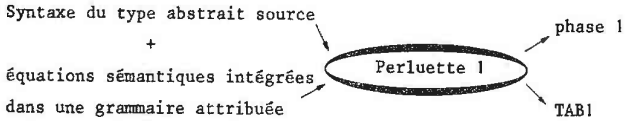
y sont équivalentes à la forme conventionnelle :

```
                ( cond ((eq a b) (plus c (plus d e))))
```

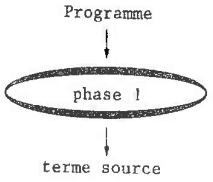
Le chapitre III indiquera comment sont faites les vérifications sur les profils des opérations lors du traitement de la grammaire attribuée du langage source. On trouvera dans le chapitre IV un exemple de ces grammaires. Rappelons leurs particularités qui les distinguent des grammaires attribuées habituelles :

- les attributs effectuant les vérifications contextuelles et la collecte d'informations doivent être décrits par une forme LISP ;
- un autre type d'attributs existe, permettant de construire le terme du type abstrait associé au programme source ; ces attributs peuvent être associés soit à un type, soit à un profil (domaine et co-domaine d'une opération) du type abstrait et sont décrits au moyen d'un méta-langage basé sur les schémas fonctionnels (parties droites des équations sémantiques).

Le schéma ci-dessous résume l'utilisation faite par Perluette des descriptions du langage source. Dans ce schéma, "TAB1" représente des informations destinées au traitement de la partie représentation, informations dont la nature sera précisée en II.2.3 ainsi qu'au chapitre III.



"phase 1" est la première phase du traducteur qui produit, à partir d'un programme écrit en langage source, un terme du type abstrait source :



II.2.2. Le langage objet

De façon similaire au langage source, le langage objet est décrit par la donnée du type abstrait qui lui est associé et d'informations permettant de réaliser la liaison entre les termes de ce type abstrait et un programme du langage objet. Ces informations sont cependant d'une nature très différente de celles que nous venons d'étudier pour le langage source, car le problème à résoudre est, en quelque sorte, réciproque du précédent : il s'agit d'obtenir un texte objet à partir d'un terme du type abstrait objet, contrairement à ce qui se passe pour le langage source, où l'on produit un terme du type abstrait source à partir du texte d'un programme. C'est l'opération désormais classique de décompilation d'un arbre abstrait.

Le problème est bien plus simple - il n'y a pas d'analyse syntaxique à affectuer, et de plus le terme objet est correct (si la partie "représentation" l'est). En fait, cette traduction peut s'effectuer, comme nous l'avons dit précédemment (voir II.1.4. : la troisième phase du traducteur) par interprétation de ce terme, interprétation réalisé par des appels par nom. Ceci est spécifié à Perluette par la description d'actions à effectuer sur un terme qui dépendent de la forme de celui-ci. Nous en donnons un exemple dans la deuxième moitié de cette partie.

II.2.2.1. Le type abstrait objet

Il est exactement semblable au type abstrait du langage source, donnant, pour chaque type d'objet du langage cible, les noms et profils des opérations qui permettent de le manipuler. La syntaxe de ces types abstraits tels qu'il sont acceptés par le système est donnée en annexe.

Nous nous contenterons donc de donner ici un exemple d'un tel type, associé à un langage cible conventionnel :

```
type contenu = union (adresse, valeur) :
   op (adresse)          → contenu : cm ;
   op (valeur, valeur)   → valeur  : add, sub, mul, div
fin contenu ;
type registre :
   op (registre) → contenu : cr ;
fin registre ;
```

```
type modif :
   op (registre, adresse)   →  modif :  load, store ;
   op (adresse)             →  modif :  load-indexé, store-indexé ;
   op (registre, valeur)    →  modif :  add-immédiat ;
   op (registre, contenu)   →  modif :  add-registre, affect-registre ;
   op (registre, registre)  →  modif :  transfert ;
   ...
fin modif ;
...
```

Dans cet exemple très simple, le langage objet manipule des adresses en mémoire et des valeurs -hexadécimales -, des registres spécialisés - accumulateur et index principalement-. Le "Type d'intérêt" de ce type abstrait est *modif*, qui représente en fait les opérations de base connues de la machine objet ; à chaque opération de ce type est associé un "mnémonique" - comme LOAD,ADD,... - par l'intermédiaire de la description du code dont l'étude est l'objet du paragraphe suivant .

Toutes les opérations déclarées dans le type objet n'apparaîtront pas pas dans le terme engendré par la deuxième phase de compilation (phase de représentation source → objet). En effet, il est peu problable que cette phase traite de l'allocation des registres par exemple, et donc que l'opération *transfert* soit explicitement engendrée dans le terme objet ; c'est cependant une opération de la machine objet qui sera utile lors de la production de code, afin de calculer par exemple un déplacement dans le registre de calcul pour l'utiliser ensuite dans le registre d'index. Inversement certaines opérations qui apparaissent dans les termes objets - comme *add* ou *cr* - ne possèdent pas de représentant direct dans le code, et leur implantation dépendra du contexte dans lequel elles apparaissent - voir l'exemple du paragraphe suivant.

II.2.2.2. La description du code

Il s'agit donc ici de la description des actions à effectuer à chaque noeud de l'arbre qui représente le terme objet pour traduire le sous- arbre correspondant en du code objet. Ces actions dépendent du sous-arbre - principalement des noeuds fils du noeud courant - et analysent donc sa structure pour "décider" du code à engendrer, par une sorte de

"reconnaissance de forme" très simple.

Bien entendu, le "générateur de code" correspondant doit pouvoir consulter son "état interne". Cela apparaît dans l'exemple qui suit, où les identificateurs précédés par le signe dièse (#) sont des "méta-identificateurs", désignant des méta-variables ou des méta-procédures.

Cet exemple concerne plus particulièrement l'opération *add* du type objet. Cet opération est à résultat dans le type *valeur* ; elle peut donc apparaître en opérande des opérations de type *modif*, *add-registre* et *affect-registre* - Nota bene : le type *modif* est le type d'intérêt pour la production de code et, par conséquent, seules les opérations dont le profil comporte ce type produisent un texte objet ; en outre l'opération *add-immédiat* n'apparaîtra pas dans les termes objet, l'allocation de registre n'étant faite qu'au cours de la phase "production de code" exposée en II.1.4.

Dans cet exemple, nous utilisons une méta-variable : # GARE ; elle représente une adresse de sommet de pile en mémoire, pile qui nous sert à garder des résultats intermédiaires ; deux méta-procédures la manipulent : #INCR et #DECR, aux noms évocateurs.

```
code (affect-registre ( <registre> , <contenu>)) =
 si <registre> ≡ registre 'A'  {l'accumulateur}
 alors cas (<contenu>) ≡
      adresse '<ad>' : LOIM,A <ad> . {load-immédiat}
      valeur '<val>' : LOIM,A <val>.
      cm (<contenu>) : cas <contenu> ≡
                       adresse '<ad>' : LOAD,A <ad> .
                       cm (<contenu>): # INCR (#GARE)
                                       STORE,X #GARE
                                       code (load(registre 'X' ,
                                                  cm (<contenu>)))
                                       {appel récursif :
                                        on calcule dans le registre
                                        d'index l'adresse de l'opérande}
                                       LOADX 0 {load-indexé}
                                       LOAD,X # GARE
                                       # DECR (# GARE).

                cr(registre 'X'): LOAD,X 0.
                                  {cas précédent simplifié}
```

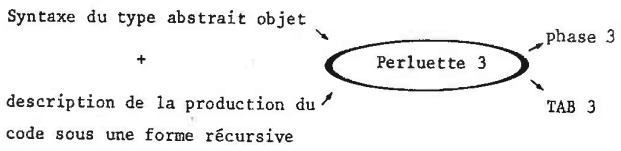
```
                    autre : erreur
                    fin cas

        cr (registre 'A') : .
        cr (registre 'X') : TRANS,X A.   {A← X}
        add (<contenu 1>,<contenu 2>):{on essaie d'optimiser}
                    si   <contenu 2> ≡ add (<contenu 3>, <contenu 4>)
                      alors code (affect-registre(registre 'A',
                                  add(add(<contenu 1>,<contenu 3>),
                                      <contenu 4>))).
                      sinon code (affect-registre(registre 'A',<contenu 1>)
                           code (add-registre(registre 'A',<contenu 2>)).
                      {le premier cas correspond à
                                  c1+(c3+c4) = (c1+c3)+c4}
        ... {sub, mul et autres}
        fin cas
    sinon {autres registres}
    ...
    fin si

code (add-registre (<registre>, <contenu>)) =
  si <registre> ≡ registre 'A'
   alors cas  <contenu> ≡
        valeur '<val>' : ADDI, A <val>      {add-immédiat}
        cm (<contenu>) : cas <contenu> ≡
                  adresse '<ad>' : ADD <ad>. {add-registre}
                  cm (<contenu>) : #INCR (# GARE)
                                   STORE,A  # GARE
                                   code (load (registre 'A', cm (<contenu>)))
                                   ADD  # GARE  {(A)+c1 est réalisé par
                                                 c1+(A)}
                                  # DECR # GARE.
                  autre : erreur
                  fin cas
```



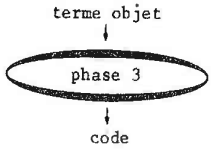
```
        add (<contenu 1>,<contenu 2>) :
                    code (add-registre (registre 'A',<contenu 1>))
                    code (add-registre (registre 'A',<contenu 2>)).
    ...
sinon  ...
fin si
```

A partir de cette description le ystème Perluette peut engendrer une fonction récursive CODE qui, travaillant sur un terme objet, produira un texte objet équivalent. Comme on le voit sur cet exemple, cette fonction peut analyser la forme que prennent ses arguments - il s'agit donc ici d'appel par nom - et, en fonction de cette forme, produire directement des mnénoniques ou se rappeler récursivement avec en argument un terme approprié, construit à partir des sous-termes de ses arguments d'origine. Certaines optimisations sont possibles et réalisées de façon très simple : ainsi dans l'exemple le terme qui constitue le second argument de l'opération *affect-registre* est réécrit dans le cas d'additions en chaîne, de façon à ne sauvegarder le contenu de l'accumulateur que lorsque cela est nécessaire.

Le fonctionnement du module de Perluette qui traite des descriptions du langage objet est résumé dans le schéma page suivante. Ce module produit notamment des informations destinées au traitement de la représentation ; ces informations, dont la nature sera étudiée dans le paragraphe suivant et au chapitre III, sont désignées par "TAB3".



"phase 3" désigne la troisième phase du traducteur, qui produit, à partir du terme du type abstrait, un programme en langage cible :



II.2.3. La représentation des termes sources par des termes objets

Cette partie de la description du traducteur est essentielle : elle établit la correspondance entre les deux langages et précise donc les choix que l'on prend pour l'implantation des objets manipulés par le langage source. Ces objets étant décrits par un type abstrait, la description de leur représentation est une suite de rubriques relatives à chaque type et opération de ce type abstrait susceptible d'apparaître dans un terme source.

La description de la représentation est utilisée par Perluette pour produire un ensemble de fonctions LISP, une fonction étant associée à un type ou une opération du type abstrait source. Le métalangage utilisé à l'intérieur des différentes rubriques de la représentation est par conséquent, comme pour les équations sémantiques étudiées précédemment, dérivé de LISP.

II.2.3.1. Représentation des types

La représentation d'un type source est donnée par le nom du ou des types le représentant, éventuellement suivi par un "schéma de représentation" des constantes de ce type. Dans ce schéma de représentation, le texte d'une constante est obtenu en décrivant cette constante et peut être manipulé

par appel d'une procédure - qui devra être écrite en LISP.

Ainsi, dans l'exemple qui suit, on déclare représenter les entiers du type abstrait donné en II.2.1.1. par les valeurs hexadécimales du type abstrait objet donné en II.2.2.1. ; les constantes sont transcrites par la fonction CONVHEXA, qui prend en argument un texte de constante entière.

```
type Entier : valeur ;
  repr Entier 'N' = valeur 'CONVHEXA (N)'
fin Entier
```

La représentation des types "union" n'est pas donnée explicitement. En fait, les types qui sont des unions d'autres types, comme *Identificateur*, constituent simplement une abréviation, comme nous l'avons vu précédemment. Les types constituants de l'union sont donc représentés indépendamment les uns des autres, d'autant que leurs représentations ont de grandes chances de différer notablement : on ne représente en général pas les identificateurs de procédures de la même manière que les identificateurs de variables entières.

II.2.3.2. Représentation des opérations

La représentation d'une opération peut être considérée comme une partie de la définition de la représentation du type résultat de cette opération. Elle est en général donnée sous la forme d'un opérateur du type abstrait, cet opérateur devant avoir comme type résultat un type objet compatible avec la représentation du type résultat de l'opération représentée - "compatible" et non "identique", en raison des unions de types dans les types abstraits source et objet. La spécification de la représentation des opérations est intrinsèquement récursive ; elle se fait, pour une opération du type source, en termes d'opérations du type objet et de la représentation des opérandes de cette opération. La représentation des opérandes peut donc être effectuée avant celle de l'opérateur lors de la deuxième phase de la traduction, comme nous l'avons vu en II.1.3.

Puisque la représentation d'une opération utilise les représentations de ses arguments, la syntaxe de sa description permet de nommer les paramètres de cette opération : cela se fait par rappel du profil de l'opération, comme on peut le voir sur l'exemple suivant, qui décrit la représentation de l'opération *Plus* du type abstrait source - les entiers étant, comme vu au paragraphe précédent, représentés par les valeurs hexadécimales du type objet - :

```
repr op (Entier i, Entier j) → Entier : Plus =
        add (repr i, repr j)
```

De même que pour la représentation des types sources, on peut avoir besoin du texte d'une constante lors de la représentation d'une opération. Ceci n'est bien entendu valide que pour les opérations dont l'argument ne peut être qu'une constante . Le même mécanisme de description des constantes que précédemment est utilisé à cette fin.

Par exemple, l'opération *Borne-sup*, qui permet d'obtenir la valeur de l'indice maximal d'un tableau, n'est jamais appliquée qu'à des identificateurs de tableaux constants ; sa représentation en tient donc compte, et peut se spécifier ainsi :

```
repr op   (Id-tab 'T') → Entier : Borne-sup =
                valeur 'BORNESUP (T)'
```

De même que dans la description du code, l'emploi de méta-variables et métaprocédures s'avère en général nécessaire lors de la description de la représentation des opérations. On peut ainsi engendrer des étiquettes au fur et à mesure des besoins, réaliser une allocation fictive de registre, et cetera. C'est par exemple le cas lors du traitement d'un terme *conditionnelle*, qui sera étudié plus en détail lors du chapitre IV, et dont nous donnons ci-dessous un extrait : la méta-fonction GENETI produit

à chaque appel un nouveau libellé d'étiquette ; les méta-instructions placées entre '(#' et '#)' sont des traitements à effectuer "à la compilation", parmi lesquels figure la déclaration de variables locales, introduite par le mot-clé local du métalangage.

```
repr op (Bool b, Modif m1, Modif m2) → Modif : Conditionnelle =
  (# local (NONEGAL, SORTIE)
     (SETQ  NONEGAL (GENETI))
     (SETQ  SORTIE  (GENETI))#)

compseq-5 (branchement-neg (étiquette 'NONEGAL', repr  b),     {if}
          repr m1, branchement (étiquette 'SORTIE')             {then}
          étiqueter (étiquette 'NONEGAL', repr m2),             {else}
          étiqueter (étiquette 'SORTIE', suite ))               {fin si}
```

La syntaxe complète des descriptions de représentations sera donnée en annexe. On remarquera que, de même que dans les descriptions des types abstraits source et objet, aucun élément n'est donné à Perluette pour conduire la preuve que le traducteur obtenu est correct. Cette preuve doit en effet être effectuée à part, le système Perluette n'ayant pour vocation que d'engendrer des compilateurs.

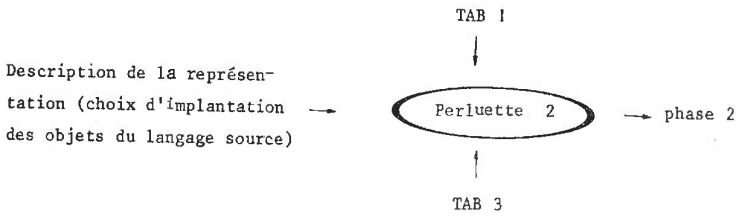
Nous avons signalé plus haut que les descriptions des types abstraits source et objet devaient être fournies à Perluette avant la description de la représentation ; en effet ces deux descriptions donnent lieu à la création des tables TAB1 et TAB3 mentionnées précédemment, qui permettent à Perluette de vérifier la "validité syntaxique" de la représentation choisie. Par validité syntaxique, nous entendons que les profils des opérations à représenter sont correctement rappelés et que les profils des opérations objets utilisées sont respectés.(Ainsi dans l'exemple ci-dessus, l'opération *Conditionnelle* doit apparaître dans le type abstrait source avec le profil *(Bool, Modif, Modif) → Modif*, l'opération branchement du type objet devant avoir le profil *(étiquette) → modif* - *Modif* étant un type source représenté par le type objet *modif* -.)

Ces deux tables constituent donc des informations syntaxiques pour le module de traitement de la description de la représentation. Leur nature est étudiée au chapitre suivant, qui traite de la réalisation effective du système Perluette.

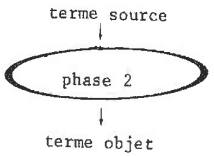
En résumé, la description de la représentation des termes du type abstrait source par des termes du type abstrait objet, qui est la manifestation de l'implantation choisie pour les objets et opérations du langage source, est utilisée par Perluette pour produire un ensemble de fonctions LISP ; chacune de ces fonctions est destinée à traduire un type de noeud de l'arbre implantant un terme source - en fait une S-expression - : à chaque opération ou type du Type Source correspond donc une telle fonction LISP, définie par l'implantation choisie, qui réécrit tout arbre dont la racine est cette opération ou ce type au moyen des réécritures de ses sous-arbres. Ainsi, si nous notons T la fonction de réécriture des termes et L(O) la fonction LISP associée à l'opération O, il vient pour la réécriture du terme $O(A_1,\ldots,A_n)$:

$$T(O(A_1,\ldots, A_n)) = \text{EVAL } (L(O)\ (T(A_1)\ldots\ T(A_n))).$$

Le schéma suivant décrit l'utilisation de Perluette relativement à la représentation :



"phase 2" est un module du traducteur final, chargé de la traduction des termes source en termes objets - formé notamment des fonction L(O) - :



II.3. Récapitulation

Il nous reste à faire une synthèse de l'utilisation générale de Perluette : trois descriptions sont requises :

- le langage source est décrit par un type abstrait, qui donne les profils des opérations pouvant apparaître dans les termes associés à des programmes source, et par une grammaire attribuée, qui, outre la syntaxe et les vérifications contextuelles, décrit les équations sémantiques permettant de construire ces termes à partir des textes des programmes ;
- le langage objet est aussi décrit par un type abstrait aux mêmes fins, et par un mécanisme de traduction de tout terme objet en un texte au moyen d'une fonction récursive opérant sur ce terme (par un mécanisme d'"appel par nom") ;
- enfin les choix d'implantation des objets manipulés par le langage source sont spécifiés par la représentation des termes sources en fonction de termes objets ; cette spécification est également récursive, un terme source étant "traduit" par son évaluation "en postfixé" - "appel par valeur".

Ceci se retrouve dans la figure suivante, qui reprend les différentes étapes d'un traducteur et d'utilisation du traducteur produit ; dans cette figure, comme dans la suite de cette thèse, les flèches simples (→) indiquent un transfert de données, les flèches doubles (⇒) indiquent une production (de module, de texte, et cetera) ; les boites carrées (□) sont des constituants (logiques) de Perluette, les boites rondes (o) les constituants du traducteur engendré. Cette figure constitue une récapitulation du fonctionnement logique du système - tel que l'utilisateur le perçoit - : en fait, comme cela est présenté dans le chapitre suivant, les différents modules qui appraissent ici ont été subdivisés lors de leur réalisation.

Nous espérons que la complexité apparente de ce schéma ne rebutera pas le lecteur ; les liaisons entre les modules du système sont en fait cachées à l'utilisateur - sauf en ce qui concerne l'ordre d'enchaînement de leur utilisation - comme lui sont cachés les termes intermédiaires manipulés par le compilateur engendré.

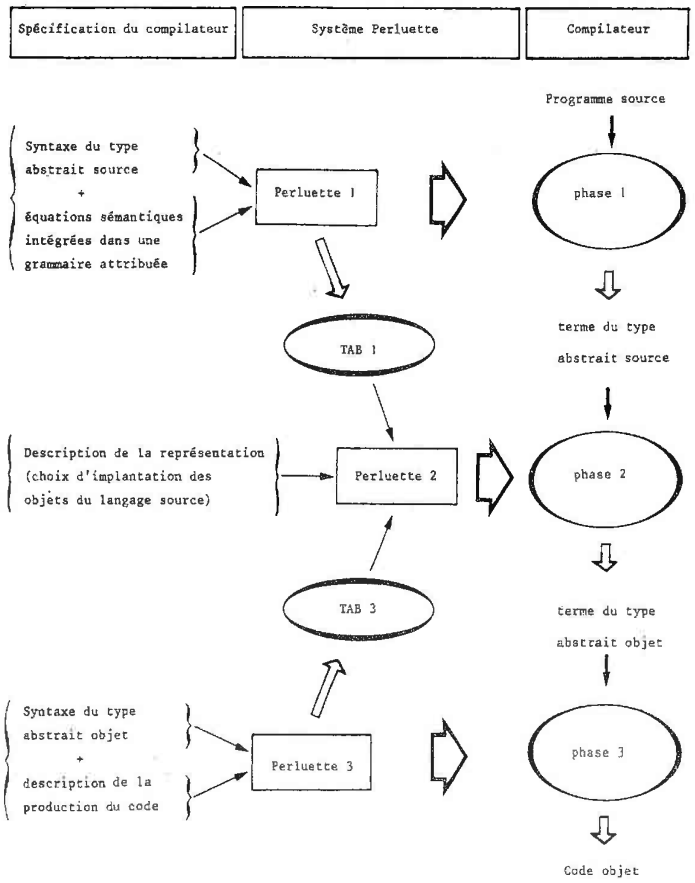


Schéma général de fonctionnement de Perluette

Nous allons maintenant, après cet exposé des principes d'utilisation de Perluette, décrire la réalisation effective du système : c'est l'objet du chapitre suivant, où les différents modules qui apparaissent sur ce schéma sont étudiés en détail.

CHAPITRE III

REALISATION DU SYSTEME

La réalisation d'un système de méta-compilation tel que Perluette pose en fait plusieurs problèmes distincts malgré leurs interactions ; il faut en effet déterminer les cinq points suivants :

- méthodes de spécification des langages source et objet, ainsi que des choix d'implantation des objets du langage source en termes de ceux du langage objet ;
- fonctionnement des traducteurs produits ;
- implantation de ces traducteurs ;
- fonctionnement du méta-compilateur ;
- implantation de ce dernier.

Nous avons, au cours des chapitres précédents, décrit nos réponses à trois de ces problèmes :

- le chapitre I a présenté la méthodologie mise en oeuvre par Perluette :
  - . le langage source est défini par un type abstrait algébrique, le Type Source, et une grammaire à attributs qui spécifie la syntaxe du langage et les vérifications contextuelles à effectuer sur les programmes ; cette grammaire comporte également des attributs représentant les équations sémantiques, qui permettent de construire le terme du Type Source que l'on associe à un programme ;
  - . le langage objet est décrit par un type abstrait algébrique, le Type Objet, et une fonction de production de code qui calcule le texte du langage objet associé à un terme du Type Objet ;

. l'implantation des objets du langage source en termes d'objets du langage objet est spécifiée comme la représentation d'un type abstrait algébrique (le Type Source) par un autre (le Type Objet) ;

- le fonctionnement des traducteurs produits a été décrit dans son principe en II.1 :
  . une première phase analyse le texte du programme et produit un terme du Type Source sous la forme d'un arbre ;
  . une deuxième phase réécrit ce terme source en un terme objet - par interprétation de l'arbre ;
  . la troisième phase du traducteur produit le code associé à l'arbre résultant de la deuxième phase ;
- le fonctionnement externe de Perluette, enfin, fait l'objet du sous-chapitre II.2 , où les descriptions que doit fournir l'utilisateur de Perluette sont présentées en détail.

Ce chapitre décrit les solutions que nous avons adoptées pour les deux autres points mentionnés ci-dessus, à savoir les choix qui ont été faits pour la réalisation effective du système. Il est divisé en trois parties : les deux premières sont consacrées aux implantations respectives des traducteurs produits et du méta-compilateur, la troisième partie développe pas à pas deux exemples simples illustrant le bien-fondé de ces choix.

III.1 Implantation des traducteurs produits par Perluette

Comme nous l'avons signalé en II.1.3, l'utilisation de Perluette s'appuie fortement sur une possibilité d'interprétation d'arbres : la seconde phase du compilateur produit réécrit le terme du Type Source engendré par la première phase en un terme du Type Objet destiné à la troisième phase, cette réécriture pouvant se faire très simplement par une forme d'évaluation du premier terme ; en effet, si, suivant la terminologie du langage LISP, on considère ces termes comme des S-expressions, la réécriture considérée est en fait une évaluation de la S-expression associée au terme source, évaluation qui produit la S-expression associée au terme objet.

Cette caractéristique nous a engagé à retenir le langage LISP comme langage hôte des traducteurs produits : son adéquation à la manipulation de termes n'est plus à démontrer et, s'il a été longtemps réservé au domaine de l'intelligence artificielle, les performances des nombreux systèmes LISP existants permettent son utilisation à d'autres fins - d'autant que nombre de ces systèmes comportent des compilateurs de LISP qui améliorent les performances des programmes par un facteur important.

Ce choix présente un avantage supplémentaire, en dehors des facilités de traitement d'arbres : la gestion de la mémoire est entièrement faite par le système LISP utilisé, ce qui simplifie grandement la tâche du système Perluette.

Nous avons donc décidé que les textes intermédiaires engendrés par les différentes phases des compilateurs produits seraient représentés par des listes LISP, leur interprétation (la réécriture) étant faite par l'évaluateur LISP utilisé. Pour préserver la consistance de l'ensemble, toutes les phases de ces compilateurs sont donc écrites autant que possible sous la forme de fonctions LISP ; ceci présente en outre l'avantage de permettre la transmission de tables entre les différentes phases - table des symboles en particulier.

La première phase cependant pose un problème particulier : celui du traitement lexical et syntaxique du programme à traduire. Le langage LISP n'est en effet pas très bien adapté au traitement de caractères ; d'autre part, comme nous l'avons signalé en II.2, nous disposons du système de traitements d'attributs DELTA [Lor 74], qui permet d'effectuer toutes les vérifications contextuelles lors de l'analyse syntaxique du programme source - DELTA est en outre associé au système SYNTAX qui produit des analyseurs syntaxiques très performants [Bou 80].

Nous avons par conséquent choisi, pour l'implantation de la première phase de tout compilateur, d'utiliser DELTA, fournissant ainsi à l'utilisateur de Perluette un moyen commode de traiter la syntaxe et les contraintes contextuelles de son langage. Cependant, la version

courante de DELTA, utilisée dans Perluette, n'est pas prévue pour communiquer directement avec un évaluateur des modèles sémantiques engendrés : à partir du programme source, l'analyseur de DELTA produit un texte comportant ces modèles, qui doit être ensuite fourni dans son intégralité au processeur chargé de son évaluation.

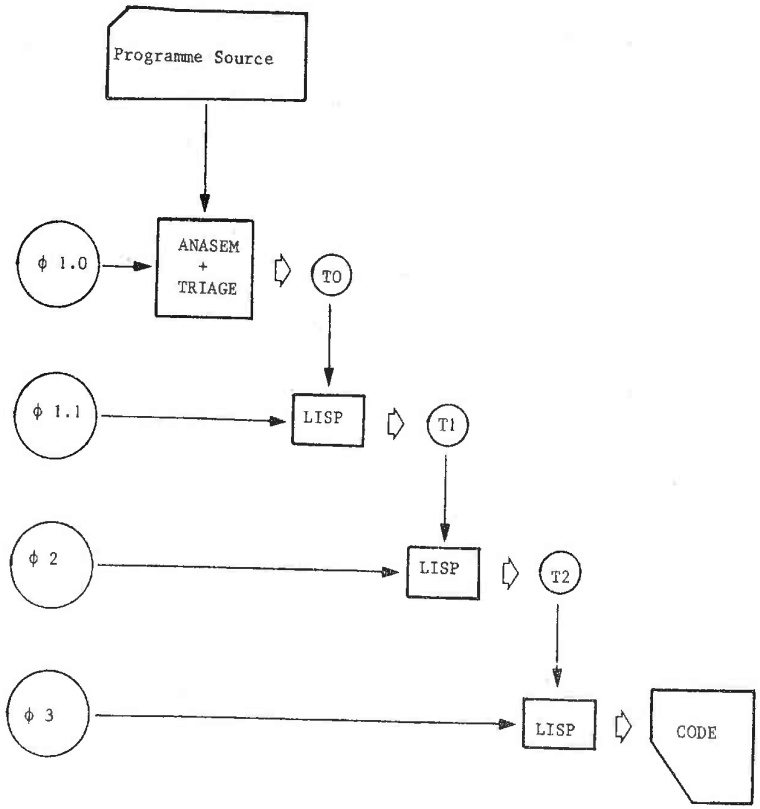
Ceci a pour conséquence de diviser la première phase des compilateurs produits en deux "passes" distinctes : la première, constituée par l'analyseur de DELTA (lui-même formé de deux modules : ANASEM pour l'analyse syntaxique avec production d'un graphe de dépendance des attributs, et TRIAGE qui effectue un tri topologique sur ce graphe) est chargée de l'analyse syntaxique du programme et de la création d'un premier texte intermédiaire, que nous noterons T0 ; la seconde "passe" est une évaluation de ce texte T0 par un sous-système LISP, décrit ci-dessous, qui produit à son tour un texte intermédiaire, T1. Ce texte T1 représente le terme du Type Source associé au programme.

Le sous-système LISP qui constitue la deuxième passe de cette première phase du compilateur est lié à un souci d'optimisation - en place et en temps d'exécution - de notre part. En effet, rappellons que Perluette I traduit les équations sémantiques intégrées dans la grammaire attribuée en des définitions d'attributs écrites en LISP. Au cours de ce traitement, chaque définition d'un tel attribut est construite comme un appel d'une fonction LISP, dont les arguments sont les attributs qui apparaissent dans l'équation sémantique. Par conséquent, le texte T0 ne contient pratiquement que des appels à ces fonctions, ce qui permet de réduire sa taille dans des proportions notables. Le sous-système LISP considéré est constitué par ces fonctions, dont le texte est obtenu à partir des équations sémantiques ; étant immuables pour chaque langage source, elles peuvent être compilées, d'où un gain appréciable en vitesse d'exécution. Un avantage annexe de l'existence de ce sous-système est qu'il favorise l'utilisation d'une programmation applicative de LISP, ce qui nous semble souhaitable pour une meilleure compréhensibilité de la description du langage source.

Les seconde et troisième phases des compilateurs produits sont implantées comme le laisse supposer la description que nous avons faite de leur fonctionnement : la seconde phase est une réécriture de T1 en un nouveau texte intermédiaire T2 par évaluation "par valeur" de T1 ; la troisième phase évalue T2 par un mécanisme "par nom" afin d'en assurer la "décompilation". Cela signifie en pratique que ces deux phases sont également réalisées par des sous-systèmes LISP. Le sous-système associé à la deuxième phase par exemple, qui est chargé de la réécriture de T1, est constitué d'un ensemble de fonctions engendrées par Perluette à partir de la description des choix de représentation. A chacune des opérations du Type Source qui peuvent apparaitre dans T1 est associée une de ces fonctions (sous forme d'une LAMBDA-expression) ; ces LAMBDA-expressions, par le mécanisme d'appel par valeur qu'elles impliquent, permettent de réaliser l'évaluation en postfixé décrite au chapitre précédent : une fonction n'est appelée que lorsque tous ses arguments ont été évalués, c'est-à-dire lorsque les sous-termes ont été réécrits. En revanche les fonctions associées aux types du Type Source sont des NLAMBDAs (dans la version courante du système, qui utilise le LISP-IRIS 80 de A. Lux [Lux 75] - au sujet de la multiplicité des langages LISP, voir [Der 79]) : une telle fonction est appelée sans que ses arguments soient évalués ; nous désirons qu'il en soit ainsi car l'argument d'une telle fonction est toujours un texte de constante, qu'il ne faut donc pas évaluer. Par ailleurs, la S-expression LISP construite par une telle fonction contient, outre le terme objet désiré, ce même texte de la constante source ; ce texte peut ainsi être utilisé par la suite, comme cela a été décrit en II.2.3.2 : la fonction de représentation associée à une opération dont l'argument est toujours une constante peut accéder au texte de cette constante.

Dans le shéma ci-dessous, qui résume l'implantation des compilateurs produits par Perluette, ces différents systèmes LISP sont désignés par $\phi 1.1$, $\phi 2$ et $\phi 3$; $\phi 1.0$ représente l'analyseur syntaxique et les différentes tables sémantiques de DELTA ; T0 est le texte intermédiaire engendré par DELTA : il s'agit d'une forme PROG, comportant les déclarations des attributs utilisés, ces attributs étant calculés au moyen

d'appels des fonctions de $\phi 1.1$; cette forme "retourne" l'un des attributs de l'axiome de la grammaire utilisée pour décrire le langage source, la valeur de cet attribut étant le texte T1 ; celui-ci est une liste LISP qui représente le terme source associé au programme ; son évaluation par le système $\phi 2$ produit la liste T2, représentant le terme objet ; le système $\phi 3$ enfin évalue T2 pour produire le code objet.



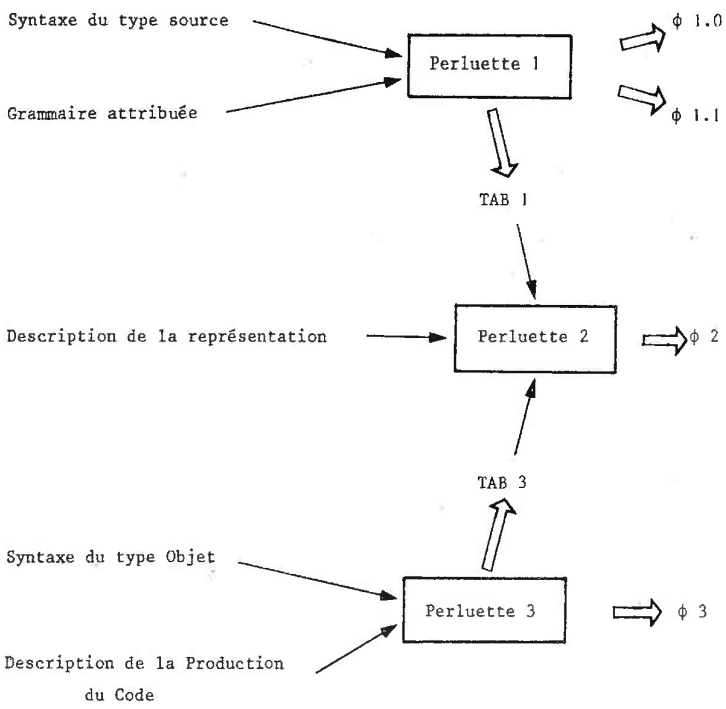
Dans ce schéma, les boites rectangulaires (□) représentent les évaluateurs utilisés (analyseur syntaxique du système SYNTAX et modules de traitement d'attributs du système DELTA d'une part(*), évaluateur LISP d'autre part). Les noms ésotériques que nous donnons aux parties du traducteur engendrées par Perluette - ϕ... - ont été introduits dans le schéma pour références ultérieures. Pour clore cette partie consacrée à l'implantation des traducteurs, nous donnons ci-dessous un exemple simplifié du contenu des textes TO, T1 et T2 lors de la traduction de l'expression I + II - nous avons remplacé dans TO les appels aux fonctions de $\phi 1.1$ par leur expansion, et omis dans T2 le texte des constantes - :

```
TO : (PROG (I001...)
           ...
           (SETQ I001 (LIST (QUOTE Entier) (QUOTE 1)))
           (SETQ I002 (LIST (QUOTE Entier) (QUOTE 11)))
           (SETQ I003 (LIST (QUOTE Plus) I001 I002))...)
T1 : ... (Plus (Entier 1) (Entier 11))...
T2 : ... (add (valeur 1) (valeur B))...
```

III.2 Implantation de Perluette

Nous nous intéressons maintenant à l'implantation que nous avons choisie pour le méta-compilateur, dont la fonction est de construire les traducteurs tels que nous venons de les décrire. Il est divisé conceptuellement en trois modules, dont nous rappelons les fonctionnalités (données et résultats) dans le schéma ci-dessous ; dans ce schéma, TAB1 et TAB3 représentent les informations construites par Perluette 1 et Perluette 3 respectivement à partir du Type Source et du Type Objet et destinées à Perluette 2 pour vérification de la validité syntaxique des choix d'implantation ; les modules désignés par ϕ... ont été décrits dans la première partie de ce chapitre, et représentent le traducteur engendré.

(*) ANASEM est le module d'analyse syntaxique qui produit un graphe de dépendance des attributs ; TRIAGE, après un tri topologique sur ce graphe, produit le programme d'évaluation des attributs que nous notons TO.



On voit que ces trois modules sont très fortement indépendants, la seule contrainte qui les lie a priori étant que le module de traitement de la représentation - Perluette 2 - doit pouvoir utiliser TAB1 et TAB3. Malgré cette indépendance relative, il nous a semblé logique - et en même temps plus pratique - d'en effectuer la programmation dans un seul et même langage, les traitements à effectuer étant du même ordre. Puisque DELTA devait intervenir pour la création de ϕ1.0, nous avons tout d'abord envisagé l'utilisation d'Algol 60, langage dans lequel sont programmés - pour des raisons historiques - les différents modules constitutifs de DELTA et SYNTAX. Mais la pauvreté des structures de données utilisables en Algol 60, ainsi que sa portabilité éminemment insuffisante, nous faisaient lui préférer PL/1 ou Pascal.

Nous avons finalement retenu ce dernier, pour sa portabilité et la sureté de programmation qu'il offre ; ceci nous a permis par ailleurs d'utiliser l'éditeur MENTOR de programmes Pascal, qui s'est avéré précieux comme outil de développement de programmes [HKM 77].

Etant donné le grand nombre de traitements à caractère syntaxique que doit effectuer Perluette, un analyseur syntaxique utilisant les tables engendrées par SYNTAX a été décrit en Pascal [Moi 79, DM 80] ; une version simplifiée de DELTA, que nous appellerons ici SDELTA, ne traitant que des attributs synthétisés "gauche-droite" a été ensuite mise en oeuvre [Lor 80] ; le constructeur sémantique de cette version, nommé TABACT, produit à partir de la définition de ces attributs en Pascal une procédure d'actions sémantiques que l'on introduit dans l'analyseur précédent, constituant ainsi un module d'analyse syntaxique et sémantique en une passe. Cette version de DELTA a été utilisée dans chacun des modules de Perluette, comme nous allons le voir.

III.2.1 Implantation de Perluette 1

Cette première partie de Perluette a pour fonction principale de traiter la grammaire attribuée qui décrit le langage source ; cette grammaire, nous l'avons vu précédemment, comporte, outre des attributs effectuant des vérifications contextuelles ou construisant des tables de symboles, les attributs qui décrivent les équations sémantiques, permettant de construire le terme source associé à un programme. Ainsi que nous l'avons signalé au chapitre précédent, un méta-langage basé sur LISP a été défini pour simplifier l'écriture de ces attributs ; ce méta-langage nous permet par ailleurs d'effectuer un certain nombre de vérifications quant à l'utilisation des opérations du Type Source : dans la plupart des cas en effet les équations sémantiques peuvent être transcrites sans utiliser LISP et Perluette 1 peut alors s'assurer que le profil de chaque opération est respecté. Ces vérifications sont effectuées par un module de Perluette 1 qui a été conçu comme un pré-processeur au constructeur sémantique de DELTA - TABDEF, écrit en Algol 60... - ; ce pré-processeur, que nous avons dénommé &1.1 - on verra pourquoi plus loin - accepte donc en entrée une grammaire à

attributs décrivant le langage source, vérifie la bonne utilisation des opérations du Type Source et produit une grammaire à attributs - très semblable à la précédente - dans laquelle le méta-langage des équations sémantiques a été tranduit en LISP.

&1.1 étant écrit au moyen de SDELTA, la version simplifiée de DELTA, un certain nombre de particularités ont dû être introduites dans la syntaxe des grammaires à attributs acceptées ; il nous a en effet semblé inutile de tenter de décrire les grammaires acceptées par TABDEF, puisque les erreurs éventuelles pouvaient être reportées par &1.1 dans la grammaire de sortie. Nous avons donc défini pour &1.1 une variante de ces grammaires qui nous permet d'ignorer tout ce qui n'est pas spécifiquement destiné à être traduit en LISP.

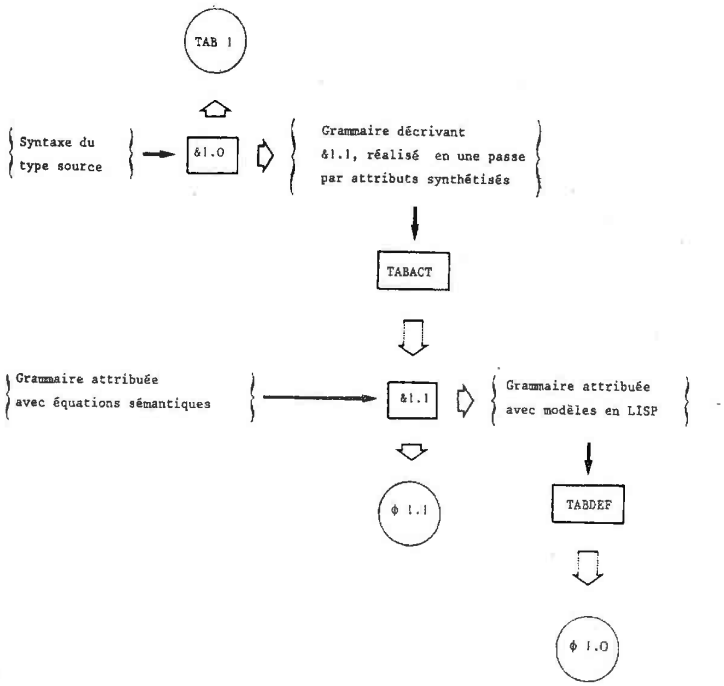
Cette variante des grammaires acceptées par TABDEF est caractérisée notamment par les particularités suivantes :

- lors de sa déclaration, tout attribut peut être associé soit à un type soit à un profil du Type Source (un profil étant constitué des domaines et du co-domaine d'un opérateur).
- les attributs "normaux", effectuant des vérifications sur le texte à traduire ou construisant des tables de symboles, doivent être décrits par une forme LISP, dont le résultat sera "affecté" - par un SETQ engendré par &1.1 - à l'attribut correspondant.
- les attributs associés à un type ou à un profil du Type Source sont décrits dans le méta-langage - décrit en annexe -.
- seules les définitions d'attributs encadrés par les symboles [et] sont concernées par ce qui précède - en fait c'est par ce biais que nous avons pu éviter de décrire les grammaires acceptées par TABDEF : tout texte débutant par] est recopié tel quel dans la grammaire engendrée par &1.1, jusqu'au [qui lui correspond - ; tout le reste de la grammaire est donc conforme au manuel d'utilisation de DELTA (à ceci près, bien entendu, que les attributs doivent être décrits en LISP).

Nous n'avons pas encore abordé le traitement du type abstrait source ; c'est qu'en fait il est réalisé par un autre module de Perluette 1, qui est chargé de produire TAB1 (destiné à Perluette 2) et une grammaire attribuée décrivant &1.1 destinée à TABACT (*). La raison d'être de l'étape supplémentaire que représente ce module - dénommé &1.0 - est double : d'une part Perluette 1 a deux traitements syntaxiques très différents à effectuer - sur le Type Source et sur la grammaire attribuée étudiée ci-dessus -, ce qui est impossible à réaliser simplement avec la version de l'analyseur syntaxique dont nous disposons ; d'autre part, lors de la réalisation d'un traducteur, les modifications que l'on désire apporter au Type Source sont extrèmement peu fréquentes alors qu'en revanche la grammaire associée à ce langage et les traitements sémantiques à effectuer évoluent beaucoup - à cause d'ambiguïtés syntaxiques à résoudre, d'erreurs dans le traitement,... - : il est donc profitable de ne pas avoir à ré-introduire le Type Source pour pouvoir modifier cette grammaire. Par ailleurs cela permet à &1.1 d'être beaucoup plus efficace, en raison de l'implantation de notre analyseur syntaxique : les mots-clés sont en effet reconnus par une technique d'adressage dispersé ; dans la grammaire produite par &1.0 à partir du Type Source, grammaire qui décrit &1.1, les noms des types et opérations du Type Source apparaissent comme des mots-clés, et le code interne de ces mots-clés, engendré par le constructeur syntaxique, sert dans &1.1 à coder les informations sur le Type Source - types constitutifs des unions et profils des opérations du langage.

Perluette 1 est donc réalisé suivant le schéma ci-dessous ; nous y avons omis la suite des constructions effectuées par les systèmes DELTA et SDELTA, ainsi que la compilation Pascal qui produit le module &1.1.

(*) : Rappelons que TABACT est le constructeur de SDELTA, alors que TABDEF est le constructeur sémantique de DELTA.



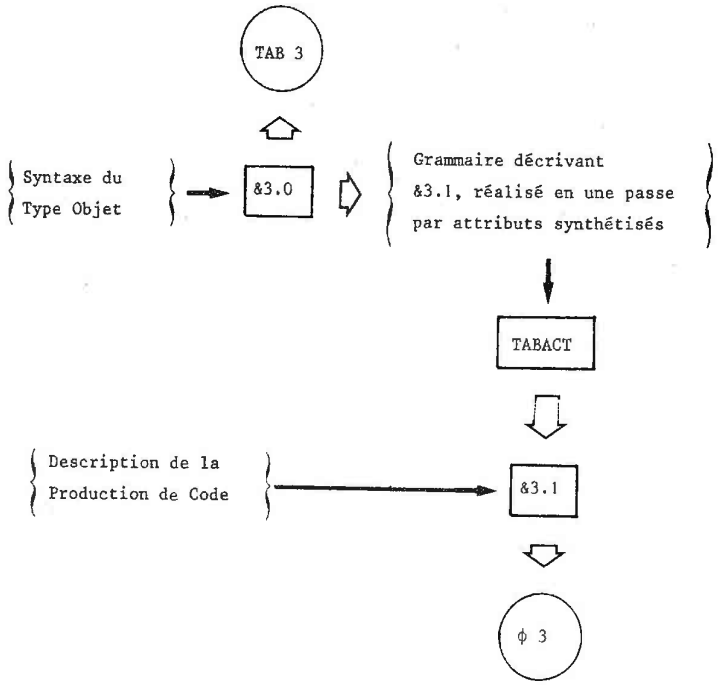
Le module &1.0 a été produit une fois pour toutes par l'intermédiaire de SDELTA ; TAB1 sera décrit en III.2.3, puiqu'il fait partie de l'implantation choisie pour Perluette 2.

III.2.2 Implantation de Perluette 3

Cette partie de Perluette a une fonction similaire à celle de Perluette 1 : son rôle est d'une part d'analyser la description du Type Objet qui lui est fournie par l'utilisateur et de tabuler cette description pour Perluette 2 (produisant TAB3, que nous étudions avec l'implantation de Perluette 2 plus loin), d'autre part de traiter la description de la production de code à partir des termes objets et d'en déduire le module $\phi 3$ du traducteur produit.

Cette similitude nous a engagé à adopter le même mode de fonctionnement : Perluette 3, qui est en cours de réalisation, comporte donc deux modules : &3.0, copié en grande partie sur &1.0 - puisque le Type Objet se décrit avec la même syntaxe que le Type Source - produit, outre TAB3 destiné à Perluette 2, une grammaire attribuée destinée à TABACT ; cette grammaire décrit les actions à effectuer, lors de l'analyse de la description de la production de code, pour produire les fonctions LISP qui constitueront $\phi 3$; le traitement de cette grammaire par SDELTA permet d'obtenir &3.1, similaire à &1.1, écrit en Pascal, dans lequel les profils des opérations du Type Objet et les types union sont conservés au moyen des codes internes associés par le constructeur syntaxique aux noms de ces types et opérations. &3.1 est cependant plus simple que &1.1 car les "mnémoniques" du code objet produit sont supposés valides : la seule vérification à effectuer est que l'analyse du terme objet est correcte vis-à-vis des profils des opérations du Type Objet.

Le schéma ci-dessous résume cette implantation ; il est très semblable à celui que nous avons donné ci-dessus pour Perluette 1, la différence résidant dans l'absence de production d'un analyseur syntaxique par DELTA. Ainsi que dans le schéma précédent, nous omettons la suite des opérations effectuées par le système SDELTA, ainsi que l'étape de compilation Pascal qui produit &3.1 à partir du texte engendré par SDELTA.



Comme pour Perluette 1, cette implantation permet une plus grande efficacité du module qui sera utilisé le plus souvent ; le contenu de TAB3 va être examiné dans la partie suivante, consacrée à Perluette 2.

III.2.3 Implantation de Perluette 2

Cette partie du méta-compilateur a pour données les descriptions des types abstraits source et objet et la description de la représentation des types et opérations du Type Source en termes de types et opérations du Type Objet. Ces données ont été décrites dans le chapitre précédent, et leurs syntaxes se trouvent en annexe.

Le Type Source et le Type Objet étant par ailleurs nécessaires aux deux autres parties du méta-compilateur, nous avons estimé qu'il était préférable de ne pas les traiter à nouveau dans Perluette 2. Nous avons donc décidé, ainsi que nous l'avons signalé précédemment, que Perluette 1 et Perluette 3 "tabuleraient" ces types abstraits pour Perluette 2. Pour ne pas avoir à relire ces tables à chaque exécution de Perluette 2

pour un même utilisateur, ces tables sont intégrées dans la description de Perluette 2 par attributs destinée à TABACT ; ceci permet, lorsque les types abstraits source et objet sont déterminés, de disposer d'un module de traitement de la représentation "statique" (compilé) donc efficace, dans lequel, comme pour &1.1 et &3.1, les codes internes des noms qui apparaissent dans les types abstraits sont utilisés pour coder les informations relatives à ces types.

La tabulation effectuée par Perluette 1 et Perluette 3 consiste donc en une production d'une partie de la grammaire à attributs décrivant Perluette 2 ; cette partie comporte toutes les informations syntaxiques et sémantiques nécessaires. Ainsi pour TAB1 - produit à partir du Type Source - on trouve une formulation du profil de chaque opération, sous la forme acceptée par SDELTA, qui permet de s'assurer lors du traitement des choix d'implantation que ces opérations sont utilisées selon leur déclaration dans le Type Source. De même TAB3 contient les noms de tous les types et opérations du Type Objet, à chaque opération étant associé un attribut qui véhicule son profil, à fins de vérification également. TAB1 et TAB3 diffèrent toutefois notablement quant à leur contenu, car lors du traitement des choix d'implantation les types et opérations du Type Source sont utilisés "en reconnaissance" - donner la représentation d'une opération revient à dire "si le terme source a la forme suivante, le remplacer par ce terme objet" - tandis que ceux du Type Objet sont utilisés "en production".

Prenons un exemple : nous utilisons un type abstrait qui contient la déclaration suivante :

```
type entier
    op (entier, entier) > entier : plus
    ...
fin entier
```

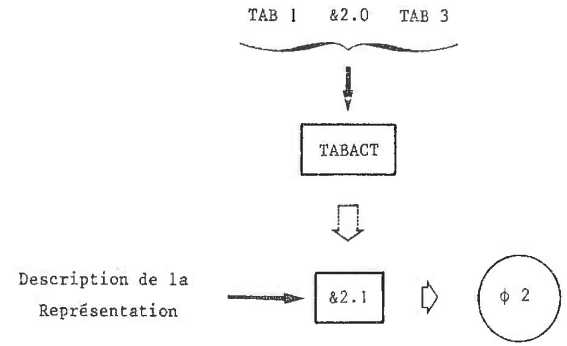
&1.0, à partir de cette description, produira un module TAB1 contenant notamment les règles syntaxiques suivantes :

```
<type source> = "entier" ;
<opération à représenter> = "("  <entier>  ","  <entier>  ")"
                            "→" "entier"  ";"  "plus" ;
<entier> = "entier" %id
         | "entier" "'" %id "'"
<opération source> = "plus"
```

En revanche, &3.0 produirait à partir de la même description les règles suivantes :

```
<type objet> = "entier" ;
<opération objet> = "plus"
```

Le schéma ci-dessous décrit l'obtention et l'utilisation d'un module de traitement des choix d'implantation ; par analogie avec les deux autres parties du système, nous avons dénommé &2.1 ce module et &2.0 la partie de la grammaire attribuée qui le décrit qui est immuable - indépendante des Types Source et Objet - :



Le module ϕ2 engendré contient les fonctions d'interprétation du terme source ; celles-ci ont été décrites dans la première partie de ce chapitre. Nous donnons ici un exemple d'une telle fonction : si, comme dans le chapitre I, nous représentons l'opération *Plus* du Type Source par l'opération *plus* du Type Objet de la manière suivante :

repr op *(Entier i, Entier j)* → *Entier : Plus* =
plus (repr *i*, repr *j*)

la fonction de réécriture associée à *Plus* dans ϕ2 aura la définition suivante :

```
Plus : (LAMBDA (i j)
               (LIST (QUOTE plus) i j))
```

Les fontions de traduction des constantes sont un peu différentes ; comme nous l'avons signalé plus haut en effet les constantes posent un problème particulier : d'une part le mécanisme d'appel par valeur utilisé pour la traduction ne peut être utilisé dans leur cas, puisque leur texte - qui est le sous-terme du terme qui leur est associé - ne doit pas être évalué ; d'autre part, toujours en raison des appels par valeur, la traduction de l'arbre T1 (terme source) est faite selon un parcours strictement postfixé, alors qu'une opération doit pouvoir accéder aux textes de ses paramètres si ceux-ci sont des constantes. Ceci est réalisé au moyen de NLAMBDAs, comme sur l'exemple ci-dessous, où l'on représente les *Entiers* du Type Source par les *valeurs* hexadécimales du Type Objet.

```
type Entier : valeur ;
     repr Entier 'N' = valeur 'CONVERTIR (N)'
fin  Entier
```

donne lieu à la définition de fonction de réécriture suivante :

```
Entier : (NLAMBDA (N)
                  (LIST (QUOTE valeur)
                        (CONVERTIR N)
                        N))
```

On voit donc que le texte de la constante entière N est accessible, après la réécriture de cette constante, comme le "CADDR" de sa représentation. L'accès à ce texte est fait par l'intermédiaire de "TEXTE" lors de la description de la représentation - voir un exemple au chapitre I -, la phrase TEXTE (repr *i*) étant traduite par &2.1 en (CADDR *i*).

III.3 Exemples simples de fonctionnement

Nous développons ici, de deux manières différentes, un exemple de traitement de l'addition dans un langage de programmation usuel ; cet exemple est destiné à montrer l'intérêt de certains des choix d'implantation que nous avons faits pour les traducteurs produits par Perluette, ainsi qu'à préciser de façon pragmatique le fonctionnement de la traduction.

Pour la simplicité de l'exposé, nous omettons notamment le traitement des erreurs et les particularités de Perluette et de DELTA.

III.3.1 Traitement de l'addition indépendamment des autres opérations

Les Types Source et Objet ainsi que les choix d'implantation sont, pour ces deux exemples, les suivants :

Type Source :
```
type Entier :
     op (Entier, Entier) → Entier : Plus, Moins, Mult, Div
     ...
fin Entier ...
```

Type Objet :
```
type valeur :
     op (valeur, valeur) → valeur : plus, moins, mult, div
     ...
fin valeur ...
```

Représentation :
```
type Entier : valeur ;
     repr Entier 'N' = valeur 'CONVERTIR (N)'
fin Entier ;
repr op (Entier i, Entier j) → Entier : Plus =
        plus (repr i, repr j)
fin Plus ...
```

Les grammaires attribuées décrivant les équations sémantiques font toute la différence entre nos deux exemples. Nous avons ici :

```
...
S ≠ modèle : Entier
...
<E> ≈ <E>' + <T>
    ≠ modèle (<E>) = Plus (modèle (<E>'), modèle (<T>))
<E> = <E>' - <T>
    ≠ modèle (<E>) = Moins (modèle (<E>'), modèle (<T>))
<T> = <F>
    ≠ modèle (<T>) = modèle (<F>)
<F> = %nombre
    ≠ modèle (<F>) = Entier '%nombre'
...
```

La grammaire destinée à DELTA construite par &1.1 est alors :

```
...
S # modèle
...
<E> = <E>' + <T>
    # modèle (<E>)
        (SETQ modèle (<E>)
              (LIST (QUOTE Plus) modèle (<E>') modèle (<T>)))
<E> = <E>' - <T>
      ...
<F> = %nombre
    # modèle (<F>)
        (SETQ modèle (<F>)
              (LIST (QUOTE Entier) (QUOTE TEXT (%nombre ))))
...
```

Pour la clarté de la description, nous avons ici remplacé les appels engendrés par &1.1 à des fonctions de ϕ1.1 par leur expansion. TEXT, utilisé dans le dernier modèle, est un attribut standard de DELTA qui, appliqué à un symbole terminal, rend la chaîne de caractères qui est son texte.

D'autre part, à partir des Types Source et Objet et de la Représentation donnés précédemment, Perluette 2 construit les fonctions suivantes qui constituent ϕ2 :

```
Entier : (NLAMBDA (N)
                  (LIST (QUOTE valeur) (CONVERTIR N) N))
Plus :   (LAMBDA (i j)
                 (LIST (QUOTE plus) i j))
```


Lors de l'analyse du texte source ... 1 + 11..., DELTA produit le texte suivant - remplaçant les noms d'attributs par des noms de variables - :

```
T0 : ...
     (SETQ I001 (LIST (QUOTE Entier) (QUOTE 1)))
     (SETQ I002 (LIST (QUOTE Entier) (QUOTE 11)))
     (SETQ I003 (LIST (QUOTE Plus) I001 I002))
     ...
```

La "aliste" associée à ce programme sera donc, après son évaluation :

```
(... (I003 . (Plus (Entier 1) (Entier 11)))
     (I002 . (Entier 11))
     (I001 . (Entier 1))...)
```

La seconde phase de traduction est pour cet exemple équivalente à la forme LISP (EVAL I003 aliste) et entraîne donc l'évaluation de

T1 : *(Plus (Entier 1) (Entier 11))*

En utilisant le contenu de ϕ2, on obtient, si CONVERTIR est correct, l'évaluation de

```
(LIST (QUOTE Plus)
      (LIST (QUOTE valeur) (QUOTE 1) (QUOTE 1))
      (LIST (QUOTE valeur) (QUOTE B) (QUOTE 11)))
```

d'où T2 : *(plus (valeur 1 1) (valeur B 11)*

Cet exemple montre assez bien l'originalité de la méthode employée pour la phase de réécriture du terme source : la transformation d'arbre abstrait qui en est l'objet est faite par interprétation directe de l'arbre - cette méthode est sûrement naturelle pour qui utilise LISP mais n'a, à notre connaissance, jamais encore été appliquée à la compilation.

En plus de sa simplicité et de sa concision, notre méthode a l'avantage de la puissance : c'est l'objet du paragraphe suivant, où nous reprenons cet exemple d'une manière peut-être plus réaliste.

III.3.2 Traitements couplés de l'addition et de la soustraction

Les Types Source et Objet et la Représentation sont les mêmes qu'au paragraphe précédent. Le traitement du programme par attributs et notamment l'écriture des équations sémantiques sont en revanche différents : la description des expressions est commune aux opérateurs "+" et "-" du langage, et un attribut synthétisé véhicule une information permettant de savoir dans quel cas on se trouve. Cet attribut est déclaré à &1.1 en tant que nom d'opération de même profil que *Plus* et *Moins* ; ceci permet à &1.1 d'effectuer les mêmes traitements sur la grammaire ci-dessous que sur la précédente et ce de façon locale à chaque modèle.

```
...
S # modèle : Entier
...
S # opération : op (Entier, Entier) → Entier
...
<E> = <E>' <op . additif> <T>
    # modèle (<E>) =
        opération calculée opération (<op . additif>)
                           (modèle (<E>'), modèle (<T>))
...
<op . additif> = +
    # opération (<op . additif>) = Plus
<op . additif> = -
    # opération (<op . additif>) = Moins
...
```


A partir de cette description &1.1 engendre la grammaire suivante pour DELTA :

```
...
S # modèle
...
S # opération
...
<E> = <E>' <op . additif> <T>
    #modèle (<E>)
        (SETQ modèle (<E>)
              (LIST opération (<op . additif>)
               modèle (<E>') modèle (<T>) ))
...
<op . additif> = +
    #opération (<op . additif>)
        (SETQ opération (<op . additif>) (QUOTE Plus))
...
```

Le texte intermédiaire produit alors par DELTA lors de l'analyse du programme ... 1 + 11 ... est le suivant :

```
T0 : ... (SETQ I001 (LIST (QUOTE Entier) (QUOTE 1)))
         (SETQ I002 (QUOTE Plus))
         (SETQ I003 (LIST (QUOTE Entier) (QUOTE 11)))
         (SETQ I004 (LIST I002 I001 I003))...
```

et nous obtenons bien, après évaluation de T0, le même terme source que précédemment :

T1 : *(Plus (Entier 1) (Entier 11))*

Cet exemple simple permet de rendre assez bien compte du gain réalisé grâce à l'emploi de l'"opération calculée" : l'introduction d'autres attributs que "modèle" associés au non-terminal <E> (tels que la table des symboles ou le traitement des cas d'erreur) n'a dans ce cas à se faire que dans une seule règle de grammaire.

III.4 Conclusion

Ce chapitre présente un exemple de réalisation d'un méta-compilateur basé sur les idées qui ont été présentées au chapitre I. Comme cela a été mentionné précédemment, il s'agissait de montrer, au moindre coût possible, la faisabilité d'un tel système. On retrouve par conséquent dans la version actuelle de Perluette les caractéristiques d'un prototype : les outils disponibles ont été utilisés au maximum, aux dépens parfois de la simplicité du méta-compilateur. En revanche, la qualité des traducteurs produits a été une préoccupation constante ; elle est à l'origine de certains choix, comme l'utilisation de LISP qui apparaît comme fondamentale.

La réalisation de cette version, malgré la complexité du problème, a demandé approximativement trois hommes-années, c'est-à-dire un investissement comparable à l'écriture manuelle d'un compilateur simple : ceci n'aurait pas été envisageable sans utiliser des outils d'aide à la compilation tels SYNTAX, DELTA et SDELTA. A l'issue de cete expérience, dont le but était de démontrer l'applicabilité de la méthode présentée au chapitre I, l'intérêt des systèmes de méta-compilation apparaît donc clairement.

CHAPITRE IV

EXEMPLE : DESCRIPTION DE SIMPROC

Nous présenterons dans ce chapitre une description complète d'un langage réaliste et d'une implantation de ce langage. Cet exemple, utilisé pour la mise au point des différents modules de Perluette, montre que le formalisme qui est à la base de notre système permet de rendre compte assez aisément des problèmes non triviaux que posent par exemple les procédures et les accès aux variables en compilation.

IV.1. Le langage SIMPROC

Le nom de ce langage est un acronyme pour " langage simple comportant des procédures". Simproc est apparenté à Pascal en ce que tous les identificateurs doivent être déclarés (mais pas obligatoirement avant leur première utilisation dans le texte du programme), les procédures peuvent être emboîtées, récursives et mutuellement récursives ; des variables entières et des tableaux d'entiers peuvent être déclarés ; les paramètres des procédures peuvent être passés par valeur ou par référence. Nous permettons en outre que lors de l'exécution d'un programme plusieurs identificateurs référencent la même variable - par le biais du passage de paramètre par référence et de l'accès aux variables globales -. Les instructions disponibles dans SIMPROC sont l'affectation, la composition conditionnelle d'instructions et l'appel de procédure.

Une première version de la description de SIMPROC a fait l'objet d'un rapport de recherche [GDM 78].

Une description syntaxique de ce langage destinée à DELTA se trouve en annexe. La description que nous donnons ci-dessous est toutefois plus simple et plus lisible :

```
<programme>    = "programme"  %ident ":"
                     <déclarations> ";" <instructions> ;
<déclarations> = <déclaration> {";" <déclaration>} ;
<déclaration>  = "entier"  %ident
               | "tableau" %ident "[" %nombre "]"
               | "proc" %ident "(" "val" %ident "," "var" %ident ")" ":"
                     [<déclarations> ";"] <instructions> "fin" "proc" ;
<instructions> = <instruction> {";" [<instruction>]} ;
<instruction>  = <variable> ":=" <expression>
               | "si" <expression> "égale" <expression>
                    "alors" [<instructions>]
                    "sinon" [<instructions>]
                    "fin" "si"
               | "exécuter" %ident "(" <expression> "," <variable> ")" ;
<variable>     = %ident | %ident "[" <expression> "]" ;
<expression>   = <terme> {( "+" | "-") <terme>} ;
<terme>        = <facteur> {( "*" | "/") <facteur>} ;
<facteur>      = <variable> | %nombre
               | "(" <expression> ")"
               | "(" "-" <facteur> ")" ;
```

Pour simplifier l'exposé et supprimer de nombreux tests dans les équations sémantiques, nous supposerons ici que tous les identificateurs apparaissant dans les déclarations sont différents et que toute procédure a un paramètre passé par valeur et un paramètre par référence ; en outre nous ne considérerons que les programmes qui répondent aux contraintes contextuelles - c'est-à-dire les programmes dont la compilation ne produit pas de message d'erreur -. Nous donnerons en IV.1.2. les axiomes du Type Source relatifs au cas où les identificateurs de procédures doivent être

distincts mais un même identificateur peut être déclaré dans deux procédures différentes.

Nous allons nous intéresser successivement aux expressions et aux instructions de SIMPROC ; ceci nous permettra de définir complètement le Type Source ; nous donnerons alors les équations sémantiques, qui sont par ailleurs transcrites dans la grammaire à attributs listée en annexe.

IV.1.1. Les expressions de SIMPROC

Les types *Entier* et *Logique* ont déjà été introduits dans le premier chapitre de cette thèse ; nous les supposerons donc connus.

Nous allons tout d'abord spécifier les identificateurs ; il en existe plusieurs types - variables simples, tableaux, procédures et paramètres passés par référence (les identificateurs de paramètres passés par valeur sont considérés comme des identificateurs de variables simples, comme en Pascal) -, d'où les définitions suivantes - que nous avons déjà vues au chapitre II - :

```
type Identificateur = union (Id-val, Id-var, Id-tab, Id-proc):
   op (Identificateur)                  → Id-proc : Père ;
   op ( )                               → Id-proc : Prog ;
   op (Id-proc)                         → Id-val  : Param-val ;
   op (Id-proc)                         → Id-var  : Param-var ;
   op (Id-tab)                          → Entier  : Borne-sup ;
   op (Identificateur, Identificateur)  → Logique : Egal
fin Identificateur
```

L'opération *Egal* est définie ainsi :

Egal (Identificateur 'A', Identificateur 'B') = (A ≡ B)

où ≡ désigne, comme au chapitre un, l'égalité textuelle de chaînes de caractères.

L'opération *Bornesup* associe à tout identificateur de tableau la valeur de sa borne supérieure. Les opérations *Param-val* et *Param-var*

associent à tout identificateur de procédure les identificateurs de ses paramètres.

L'opération *Père* permet de rendre compte de la localité des déclarations : elle associe à tout identificateur déclaré dans le programme l'identificateur de la procédure où il est déclaré ; le programme lui-même est considéré comme une procédure pour cette opération et nous désignerons par *Prog ( )* l'identificateur qui lui est associé. La définition de *Père* est la suivante :

Père (Param-val (ρ)) = ρ
Père (Param-var (ρ)) = ρ
Pre *(Père ; id) = ~ Egal (id, Prog ())*

Cette opération permet donc de traiter les aspects statiques de la portée des identificateurs. Dans un langage tel que SIMPROC se pose également un problème de portée dynamique : nous voulons spécifier qu'un identificateur local à une procédure repère une variable différente à chaque appel de cette procédure ; nous voulons également spécifier quelle est la variable repérée par un identificateur non local à une procédure. Pour cela nous allons introduire deux nouveaux types, le type "appel de procédure" et le type "variable".

Le type "appel de procédure" tout d'abord : on a lors de l'exécution d'un programme SIMPROC une suite de procédures "actives", chacune appelante d'une autre sauf la procédure active "en cours d'exécution" ; toutes ces activations de procédures sont différentes, bien que le même nom de procédure puisse apparaître plusieurs fois dans la suite. Nous spécifions ceci en associant à chaque appel de procédure son nom ainsi que l'appel qui le précède dans la suite des appels ; une opération nous permet en outre de connaître l'appel "en cours d'exécution", qui est le dernier la suite :

type *Appel* :

op ()	→	*Appel : Appel-en-cours ;*
op (*Appel*)	→	*Appel : Appelant ;*
op (*Appel*)	→	*Id-proc : Nom ;*
op (*Appel, Appel*)	→	*Logique : Egal ;*

axiomes

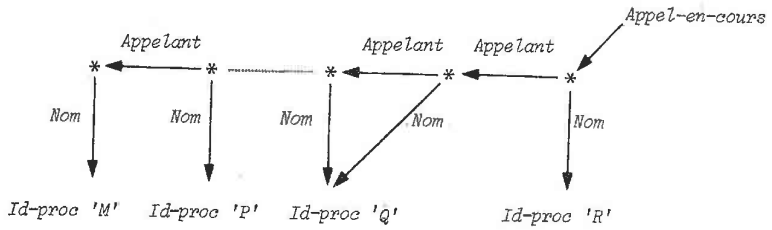
Egal (Appel-en-cours(), Appel-en-cours ()) = vrai
Egal (Appelant (a_1), Appelant (a_2)) = Egal (a_1, a_2)
Egal (Appelant (a), Appel-en-cours ()) = faux
Egal (a_1, a_2) = Egal (a_2, a_1)
Egal (a_1, a_2) ∧ Egal (a_2, a_3) ⊃ Egal (a_1, a_3)

restriction

Pre *(Appelant ; a) = ~ Egal (Nom(a), Prog ())*

fin *Appel*

La restriction donnée ici exprime le fait que le programme est toujours le premier appelant de la chaîne des appels. Un exemple d'une telle chaîne est donné par la figure suivante :



Dans cet exemple, le nom du programme est *M* et la procédure de nom *Q* est appelée récursivement.

Nous pouvons maintenant présenter le type "variable" et les relations entre les *Identificateurs*, les *Appels* et les *Variables*.

Dans le cas d'un identificateur de variable simple, on a :

type *Variable* :

op (*Id-val, Appel*) → *Variable* : *Désigne* ;
op (*Variable*) → *Entier* : *Valeur* ;
op (*Variable, Variable*) → *Logique* : *Egal* ;

axiomes

"Outre les axiomes usuels sur l'égalité, on a"

(1) $Egal\ (id_1, id_2) \wedge Egal\ (a_1, a_2) \supset Egal\ (Désigne\ (id_1, a_1), Désigne\ (id_2, a_2))$

(2) $\sim (Egal\ (id_1, id_2) \wedge Egal\ (a_1, a_2))$

$\Leftrightarrow (Egal\ (Nom\ (a_2), Père\ (id_1)) \wedge Egal\ (Nom\ (a_2), Père\ (id_2)))$

$\supset \sim Egal\ (Désigne\ (id_1\ a_1), Désigne\ (id_2, a_2))$

(3) $\sim Egal\ (Nom\ (a), Père\ (id)) \supset Egal\ (Désigne\ (id, a),$
$Désigne\ (id, Appelant\ (a))$

restriction

Pré (*Désigne ; id, a*) = *Visible (id, Nom (a))*

fin *Variable*

La signification de l'axiome (1) est simple : deux couples (identificateur, appel) désignent la même variable lorsqu'ils sont identiques.

L'axiome (2) exprime de manière très concise les trois règles suivantes :

- deux identificateurs différents désignent toujours deux variables différentes ;
- pour deux appels différents d'une même procédure - c'est-à-dire en cas d'appel récursif - les variables locales de cette procédure sont dupliquées.
- les variables locales à un appel sont différentes des variables locales à un autre appel (la règle précédente est un cas particulier de celle-ci).

Le cas des variables globales est spécifié par l'axiome (3) : la variable qui correspond à un identificateur non local est celle de l'appel de la procédure, à laquelle il est local, le plus récent ; ceci est traité en "remontant la chaîne des appels" - nous ne pouvons le spécifier de cette manière que parce que tous les identificateurs sont supposés différents-.

La précondition associée à *Désigne* permet de rendre compte des règles de portée des identificateurs. Nous avons introduit dans cette précondition l'opération *Visible*, dont la définition est la suivante :

op (*Identificateur, Id-proc*) → *Logique* : *Visible* ;

axiome

Visible (id, idp) = si *Egal (Père (id), idp)*
alors *vrai*
sinon si *Egal (idp, Prog ())*
alors *faux*
sinon *Visible (id, Père (idp))*

Cette opération permet de s'assurer que l'identificateur *id* est accessible dans la procédure *idp* et utilise la transitivité de l'accessibilité : un identificateur est accessible s'il est local ou accessible depuis la procédure englobante.

Il résulte de toutes ces définitions que la valeur sémantique d'un identificateur de variable simple rencontré dans une expression s'écrit simplement *Valeur (Désigne (Id-val 'id', Appel-en-cours ()))*

Nous pouvons maintenant étudier le cas des identificateurs de paramètres passés par référence. Nous introduisons un nouveau type, qui correspond aux variables particulières qui permettent d'accèder aux variables étudiées ci-dessus.

type *Ref-var* :

op (*Id-var, Appel*) → *Ref-var* : *Ref-Dés* ;
op (*Réf-var*) → *Variable* : *Référence* ;
op (*Réf-var, Réf-var*) → *Logique* : *Egal* ;

axiomes

"Outre les axiomes usuels sur l'égalité, on a :"

Egal (rv_1, rv_2) ⊃ *Egal* (*Référence* (rv_1), *Référence* (rv_2))

Egal (id_1, id_2) ∧ *Egal* (a_1, a_2) ⊃ *Egal* (*Ref-Dés* (id_1, a_1), *Ref-Dés* (id_2, a_2))

~(*Egal* (id_1, id_2) ∧ *Egal* (a_1, a_2))

∧ *Egal* (*Nom* (a_1), *Père* (id_1)) ∧ *Egal* (*Nom* (a_2), *Père* (id_2))

⊃ ~*Egal* (*Ref-Dés* (id_1, a_1), *Ref-Dés* (id_2, a_2))

~(*Egal* (*Nom* (*a*), *Père* (*id*)) ⊃ *Egal* (*Ref-Dés* (*id*, *a*) *Ref-Dés* (*id*, *Appelant* (*a*)))

restriction

Pre (*Ref-Dés* ; *id*, *a*) = *Visible* (*id*, *Nom* (*a*))

fin *Ref-var*

Ces axiomes et restriction sont très voisins de ceux que nous avons donnés pour le type *Variable* et l'opération *Désigne*. La valeur sémantique d'un "identificateur de paramètre passé par référence" rencontré dans une expression s'écrira

Valeur (*Référence* (*Ref-Dés* (*Id-var* 'id', *Appel-en-cours* ()))).

Il nous reste à spécifier les variables que permettent de désigner les identificateurs de tableau. Il suffit pour cela de définir dans le type *Variable* une opération *Elément* de la manière suivante :

op (*Id-tab*, *Appel*, *Entier*) → *Variable* : *Element* ;

axiomes

(4) *Egal* (idt_1, idt_2) ∧ *Egal* (a_1, a_2) ∧ *Egal* (e_1, e_2)

⊃ *Egal* (*Element* (idt_1, a_1, e_1), *Element* (idt_2, a_2, e_2)

(5) ~(*Egal* (idt_1, idt_2) ∧ *Egal* (a_1, a_2) ∧ *Egal* (e_1, e_2))

∧ *Egal* (*Nom* (a_1), *Père* (idt_1)) ∧ *Egal* (*Nom* (a_2), *Père* (idt_2))

⊃ ~*Egal* (*Element* (idt_1, a_1, e_1), *Element* (idt_2, a_2, e_2))

(6) ~ *Egal* (*Nom* (*a*), *Père* (*idt*))

⊃ *Egal* (*Elément* (*idt*, *a*, *e*), *Element* (*idt*, *Appelant* (*a*), *e*))

(7) ~ *Egal* (*Désigne* (*id*, a_1), *Element* (*idt*, a_2, *e*))

restriction

Pre (*Elément* ; *idt*, *a*, *e*) = *Visible* (*idt*, *Nom* (*a*))

∧ (*Entier* '1' ≤ *e*)

∧ (*e* ≤ *Bornesup* (*idt*))

Les axiomes (4), (5) et (6) sont similaires aux axiomes (1), (2) et (3) donnés plus haut pour les variables simples. L'axiomes (7) exprime une autre propriété de SIMPROC : un identificateur de variable simple ne peut pas désigner un élément de tableau.

La précondition sur *Elément* exprime d'une part les régles de portée statique et précise d'autre part quels sont les indices acceptables ; les erreurs d'indice n'ont pas à être traitées par l'implanteur du compilateur de SIMPROC. Nous avons pour l'écriture de ces cas d'erreur utilisé la même notation infixée du prédicat sur les entiers qu'au chapitre un, pour les mêmes raisons de lisibilité.

La description des opérations qui permettent de définir la sémantique des expressions de SIMPROC est maintenant complète. Rappelons les différentes valeurs sémantiques associées à l'accès aux valeurs dans ces expressions :

Valeur (*Désigne* (*Id-val* 'id', *Appel-en-cours* ()))
Valeur (*Référence* (*Ref-Dés* (*Id-var* 'id', *Appel-en-cours* ()))
Valeur (*Elément* (*Id-tab* 'id', *Appel-en-cours* (), *e*))

Nous allons maintenant nous intéresser, comme annoncé plus haut, au cas où les identificateurs ne sont pas supposés tous différents. Nous traiterons ensuite les instructions de SIMPROC. Les équations sémantiques feront l'objet du paragraphe IV.1.4.

IV.1.2. Axiomatisation d'identificateurs non tous différents

Nous montrons dans ce paragraphe comment on peut supprimer une des restrictionsque nous avons imposées à SIMPROC afin de simplifier le Type Source. Nous allons supposer, non plus que tous les identificateurs sont distincts, mais simplement que les identificateurs de procédures doivent être différents : un même identificateur peut être déclaré dans plusieurs procédures. Nous décrivons ici les modifications à apporter au Type Source dans ce cas.

Nous supposons donc que seuls les identificateurs de procédures sont uniques. Il nous faut par conséquent scinder l'opération *Père* en deux :

- une opération *Englobant* va nous permettre de rendre compte de la structure des déclarations de procédures ; elle associe à tout identificateur de procédure l'identificateur de la procédure où elle est déclarée ;
- une opération *Local*, à résultat *Logique*, exprime la relation qui existe entre un identificateur de procédure et un identificateur local à cette procédure ; initialisée à *faux*, cette relation sera modifiée par les déclarations.

Le Type Source est donc modifié et contient notamment ceci :

op (*Id-proc*) → *Id-proc* : *Englobant* ;
op (*Identificateur, Id-proc*) → *Logique* : *Local* ;

axiomes

Local (Param-val (ρ), ρ) = vrai
Local (Param-var (ρ), ρ) = vrai
Local (Prog (), ρ) = faux
Local (ρ, Englovant (ρ)) = vrai

restriction

Pre (*Englobant* ; ρ) = ~*Egal (ρ, Prog ())*

Il nous faut également modifier le Type Source relativement aux *Appels*, de façon à pouvoir connaître directement l'appel actif d'une procédure englobante pour accéder à ses variables locales ; en effet, pour accéder à la variable correspondant à un identificateur global, nous ne pouvons plus remonter la chaîne des appels comme précédemment, puisque cet identificateur peut être local à plusieurs procédures. Nous enrichissons donc le Type Source de l'opération *Lien-statique* qui, à partir d'un appel d'une procédure, rend l'appel le plus récent de sa procédure englobante :

op (*Appel*) → *Appel* : *Lien-statique* ;
op (*Id-proc, Appel*) → *Appel* : *Chercher-appel* ;

axiomes

Lien statique (a) = Chercher-appel (Englobant (Nom (a)), a)
Chercher-appel (ρ, a) = si *Egal (Nom (a), ρ)*
alors *a*
sinon *Chercher-appel (ρ, Appelant (a))*

restriction

Pre (*Lien-statique* ; *a*) = ~ *Egal (Nom (a), Prog ())*

Nous pouvons maintenant modifier les axiomes relatifs aux variables pour tenir compte de la multiplicité des occurences d'un identificateur dans la chaîne des appels ; dans l'axiome (2) précédent qui exprime les règles relatives aux variables locales, on tient compte de la relation *Local*, et dans l'axiome (3), qui spécifie l'accès aux variables globales, il suffit de remplacer l'opération *Appelant* par l'opération *Lien-statique*, qui permet de désigner directement l'appel désiré.

axiomes

(1) $Egal\ (id_1, id_2) \wedge Egal\ (a_1, a_2)$
$\supset Egal\ (Désigne\ (id_1, a_1), Désigne\ (id_2, a_2))$

(2)' $\sim(Egal\ (id_1, id_2) \wedge Egal\ (a_1, a_2))$
$\wedge\ (Local\ (id_1, Nom\ (a_1)) \wedge Local\ (id_2, Nom\ (a_2)))$
$\supset \sim Egal\ (Désigne\ (id_1, a_1), Désigne\ (id_2, a_2))$

(3)' ~ *Local (id, Nom (a))* ⊃ *Egal (Désigne (id, a),*
Désigne (id, Lien-statique (a)))

Les mêmes remplacements sont à effectuer dans les axiomes sur *Réf-Dés* et *Elément*.

Enfin la définition de l'opération *Visible* doit être adaptée aux nouvelles opérations sur les identificateurs :

Visible (id, idp) = si *Local (id,idp)*
alors *vrai*
sinon si *Egal (idp, Prog ())*
alors *faux*
sinon *Visible (id, Englobant (idp))*

On voit que permettre la déclaration d'un même identificateur dans plusieurs procédures ne nécessite aucun changement fondamental dans le Type Source ; qui plus est, les fonctions sémantiques n'en sont pas affectées : seuls les axiomes relatifs à certaines des opérations sur les identificateurs ont dû être modifiés légèrement, comme devraient l'être les définitions des modifications traitant des déclarations - à titre d'exemple, déclarer l'identificateur *idv* dans la procédure *idp* n'est plus défini par subst (*Père (idv), idp*), mais par subst (*Local (idv, idp), Logique 'vrai'*)-. Nous allons néanmoins dans la suite utiliser le Type Source tel qu'il a été décrit au paragraphe précédent, car les traitements à effectuer lors de la représentation sont plus simples à décrire dans le cas où les identificateurs sont tous différents.

IV.1.3. Les instructions de SIMPROC

De même qu'au chapitre I, nous allons définir un type *Modif* : la valeur sémantique de toute instruction ou déclaration sera un terme de ce type ; un terme de type *Modif* sera également associé à tout identificateur de procédure, par l'intermédiaire de l'opération *Possède*, terme représentant la valeur sémantique du corps de cette procédure.

Comme dans les paragraphes précédents nous allons procéder en plusieurs étapes. Nous considérons tout d'abord les opérations relatives aux déclarations de SIMPROC :

type *Modif* :

op (*Id-val, Id-proc*) → *Modif : Déclarer-ent* ;
op (*Id-tab, Id-proc, Entier*) → *Modif : Déclarer-tab* ;
op (*Id-proc, Id-proc, Id-val, Id-var, Modif*) → *Modif : Déclarer-proc* ;
op (*Id-proc*) → *Modif : Possède* ;

définitions

Déclarer-ent (idv, idp) = subst (*Père (idv), idp*)
Déclarer-tab (idt, idp, e) = subst (<*Père (idt), Borne-sup (idt)*>,<*idp, e*>)
Déclarer-proc (idp, idp_1, idv, idr, m)
= subst (<*Père (idp), Param-val (idp), Param-var (idp), Possède (idp)*>,
<*idp_1, idv, idr, m*>)

Ces définitions sont assez simples à lire ; elles décrivent les caractéristiques des identificateurs déclarés, au moyen de substitutions simultanées. On remarquera par ailleurs que nous ne définissons pas d'opération d'égalité entre modifications : seule nous intéresse l'identité, qui permet de rendre compte de l'exécution du corps d'une procédure, par la règle

Possède (idp) = m ∈S ⟹ appl (*Possède (idp)*, S) = appl (*m*, S).

Cette absence d'égalité entre modifications assure que l'effet de subst (*Possède (idp), m*), qui est une partie de la définition de *Déclarer-proc*, est d'introduire la seule formule *Possède (idp) = m* dans l'état courant.

Nous spécifions maintenant les autres modifications présentes dans le Type Source associé à SIMPROC, sauf l'appel de procédure qui sera étudié plus loin.

op (*Variable, Entier*) → *Modif : Affecter* ;
op (*Id-proc*) → *Modif : Initialiser* ;
op (*Modif, Modif*) → *Modif : Seq* ;
op (*Entier, Entier, Modif, Modif*) → *Modif : Cond* ;
op ( ) → *Modif : Continuer*

définitions

Affecter (v,e) = subst (*Valeur (v), e*)

Initialiser (idp)=subst (<*Nom (Appel-en-cours ()), Prog ()*>,<*idp, idp*>)

appl (*Seq* (m_1, m_2), S) = appl (m_2, appl (m_1, S))

(*Egal (i, j)* = *vrai*) ∈S ⟹ appl (*Cond (i, j,* m_1, m_2), S) = appl (m_1, S)

(*Egal (i, j)* = *faux*) ∈S ⟹ appl (*Cond (i, j,* m_1, m_2), S) = appl (m_2, S)

appl (*Continuer ()*, S) = S

L'opération *Initialiser* est chargée de la mise en place initiale de la chaîne des appels. Elle ne sera donc utilisée que dans la valeur sémantique du programme.

Nous en venons enfin à l'appel de procédure, qui constitue en général le point délicat de la description d'un langage : on doit en effet créer un nouvel environnement - un nouvel *Appel* - pour que les identificateurs locaux désignent de nouvelles variables, procéder au passage des paramètres - sans oublier que l'évaluation des expressions doit se faire dans l'ancien environnement-, puis exécuter le corps de la procédure (obtenu par *Possède*) et enfin revenir à l'*Appel* précédent. La modification *Exécuter* que nous allons décrire effectue ces opérations ; nous utilisons pour sa définition deux modifications auxiliaires, introduites pour des raisons de lisibilité, qui seront spécifiées ensuite.

op (*Id-proc, Entier, Variable, Id-proc*) → *Modif* : *Exécuter* ;

op (*Id-proc, Entier, Variable*) → *Modif* : *Appeler* ;

op ( ) → *Modif* : *Retourner* ;

définition

Exécuter (idp, e, v, idp_2) = *Seq (Seq (Appeler (idp, e, v), Possède (idp)), Retourner ())*

restriction

Pre (*exécuter ;* idp_1, *e, v,* idp_2) = *Visible* (idp_1, idp_2)

La précondition définie ici exprime les règles de visibilité usuelles. La modification *Retourner* est simple à décrire : nous voulons que, dans l'état appl (*Retourner ( )*, S), la valeur obtenue par *Appel-en-cours ( )* soit la même que celle de *Appelant (Appel-en-cours ( ))* dans l'état S. Cela suffit puisque tous les accès qui peuvent être modifiés lors de l'exécution d'une procédure sont paramètrés par *Appel-en-cours*. *Retourner* se spécifie donc ainsi :

définition

Retourner () = subst (*Appel-en-cours (), Appelant (Appel-en-cours ()))*

L'effet de *Appeler* est plus complexe à décrire ; nous allons spécifier tout d'abord cet effet vis-à-vis des *Appels*, le passage de paramètres se traitant ensuite sans difficultés.

Appeler effectue un changement d'environnement : après ce changement, l'*Appel Appelant (Appel-en-cours ())* est le même que *Appel-en-cours ()* dans l'environnement précédent, et le nom du nouvel *Appel* est l'identificateur de la procédure appelée. Par conséquent, si nous posons S' = appl (*Appeler (idp, e, v)*, S), S' peut être caractérisé par

Nom (Appel-en-cours ()) = *idp* ∈S'

P ∈ S ⟹ *P* [*Appelant (Appel-en-cours ())* / *Appel-en cours ()*] ∈S'

Pour exprimer ceci en terme de subst, nous utilisons un stratagème qui se réfère à la définition de subst donnée au chapitre un : nous allons nous servir du nom d'opération auxiliaire qui apparaît dans cette définition pour désigner l'*Appel* créé. Le changement d'environnement effectué par *Appeler (idp, e, v)* est alors défini par

subst (<*Appelant (Appel-en-cours'())*, *Nom (Appel-en-cours'())*, *Appel-en-cours ()*>,
< *Appel-en-cours ()* , *idp* , *Appel-en-cours'()*>)

D'après la définition de subst, S' sera défini par :

$$S' = \{P \mid P[^{Appelant'}/_{Appelant}, {}^{Nom'}/_{Nom}, {}^{Appel\text{-}en\text{-}cours'}/_{Appel\text{-}en\text{-}cours}] \in S$$

$$+ Appelant'(x) = \underline{si}\ Egal(x, Appel\text{-}en\text{-}cours'()) \ \underline{alors}\ Appel\text{-}en\text{-}cours(\,) \ \underline{sinon}\ Appelant(x)$$

$$+ Nom'(x) = \underline{si}\ Egal(x, Appel\text{-}en\text{-}cours'()) \ \underline{alors}\ idp \ \underline{sinon}\ Nom(x)$$

$$+ Appel\text{-}en\text{-}cours' = Appel\text{-}en\text{-}cours'\}$$

Cette utilisation de la définition de subst comme productrice d'un nom d'opération intermédiaire n'est peut-être pas très satisfaisante sur un plan esthétique mais nous permet, on le voit, de spécifier assez clairement et simplement la création d'une nouvelle valeur de type *Appel*. La spécification complète de la modification *Appeler*, comportant le passage de paramètres, est maintenant simple : il suffit d'évaluer les arguments dans l'ancien environnement et de les affecter à des variables du nouvel environnement. *Appeler* est donc définie ainsi :

définition

Appeler (idp, e, v)

= subst (⟨*Appelant (Appel-en-cours' (), Nom (Appel-en-cours'())*, *Appel-en-cours' ()*, *Valeur (Désigne (Param-val (idp), Appel-en-cours' ()))*, *Référence (Ref-Dés (Param-var (idp), Appel-en-cours' ()))*⟩, ⟨*Appel-en-cours(), idp, Appel-en cours' (), e, v*⟩)

Le Type Source associé à SIMPROC est maintenant complètement défini. Nous donnons dans la partie qui suit les fonctions qui permettent de calculer, pour chaque règle syntaxique de la description de SIMPROC, le terme du Type Souce qui lui est associé.

IV.1.4. Les fonctions sémantiques

La sémantique des programmes SIMPROC va être donnée au moyen de quatre fonctions sémantiques ; leurs profils sont les suivants:

T : Programme → terme de type *Modif*

M : *Id-proc* × {Déclarations ∪ Déclaration ∪ Instructions ∪ Instruction} → terme de type *Modif*

V : *Id-proc* × Variable → terme de type *Variable*

E : *Id-proc* × {Expression ∪ Terme ∪ Facteur} → terme de type *Entier*

Les symboles tels que Programme, Déclarations, Variable ou Expression utilisés dans ces profils représentent les non-terminaux qui nous ont servi dans la description de la syntaxe de SIMPROC au début de ce chapitre. Le premier paramètre des fonctions *M*, *V* et *E*, de type *Id-proc*, permet d'identifier la procédure à laquelle la partie de programme considérée est locale ; ce paramètre n'est pas absolument nécessaire à l'écriture des fonctions sémantiques puisqu'il représente le résultat de *Nom (Appel-en-cours ())* en tout point d'un programme ; son utilisation nous permet cependant de garder son caractère "statique" à l'opération *Nom* et donc d'éviter d'avoir à la représenter, ce qui mènerait à des implantations inefficaces. La solution adoptée incite l'implanteur du compilateur à traiter ce paramètre comme un attribut (voir l'attribut ENGLOBANT dans l'annexe consacrée à la grammaire attribuée de SIMPROC).

Pour faciliter la lecture, nous noterons S_p ⟦NT⟧ la fonction sémantique S ⟦*Id-proc 'p'*, NT⟧ - S pouvant être M, V ou E -.

T ⟦programme %ident : Déclarations ; Instructions⟧

= *Seq (Seq (Initialiser (Id-proc '%ident')*, $M_{\%ident}$ ⟦Déclarations⟧), $M_{\%ident}$ ⟦Instructions⟧)

Rappelons que la modification *Initialiser* est chargée de la mise en place de la chaîne des *Appels*.

M_p ⟦Déclaration$_1$; Déclaration$_2$⟧ = *Seq* (M_p ⟦Déclaration$_1$⟧, M_p ⟦Déclaration$_2$⟧)

M_p ⟦entier %ident⟧ = *Déclarer-ent* (*Id-val* '%ident', *Id-proc* 'p')

M_p ⟦tableau %ident [%nombre] ⟧ =

Déclarer-tab (*Id-tab*'%ident', *Id-proc* 'p', *Entier* '%nombre')

M_p ⟦proc %ident$_1$ (val %ident$_2$, var %ident$_3$) :

Déclarations ; Instructions fin proc⟧ =

Déclarer-proc (*Id-proc* '%ident$_1$', *Id-proc* 'p', *Id-val* '%ident$_2$', *Id-var* '%ident$_3$',

Seq ($M_{\%ident_1}$ ⟦Déclarations⟧, $M_{\%ident_1}$ ⟦Instructions⟧))

M_p ⟦ ⟧ = *Continuer* ()

Cette dernière équation sémantique nous permet de prendre en compte de manière simple les différentes productions vides présentes dans la syntaxe de SIMPROC, tant au niveau des déclarations que pour les instructions.

M_p ⟦Instruction$_1$; Instruction$_2$⟧ = *Seq* (M_p ⟦Instruction$_1$⟧, M_p ⟦Instruction$_2$⟧)

M_p ⟦Variable := Expression⟧ = *Affecter* (V_p ⟦Variable⟧, E_p ⟦Expression⟧)

M_p ⟦exécuter %ident (Expression, Variable)⟧ =

Exécuter (*Id-proc* '%ident', *Ep* ⟦Expression⟧, V_p ⟦Variable⟧, *Id-proc* 'p')

M_p ⟦si Expression$_1$ égale Expression$_2$ alors Instructions$_1$ sinon Instructions$_2$ fin si⟧ =

Cond (E_p ⟦Expression$_1$⟧, E_p ⟦Expression$_2$⟧,

M_p ⟦Instructions$_1$⟧, M_p ⟦Instructions$_2$⟧)

Le cas des variables simples, traité ci-dessous, nécessite l'utilisation d'une construction conditionnelle pour connaître le type associé à un identificateur. Ceci sera implanté dans le compilateur par une recherche en table et une conditionnelle LISP - voir l'annexe précitée -.

V_p ⟦%ident⟧ = si %ident ∈ *Id-val*

alors *Désigne* (*Id-val* '%ident', *Appel-en-cours* ())

sinon *Référence* (*Ref-Dés* (*Id-var* '%ident', *Appel-en-cours* ()))

V_p ⟦%ident [Expression] ⟧ = *Elément* (*Id-tab* '%ident', *Appel-en cours* ()

E_p ⟦Expression⟧)

La sémantique des expressions enfin s'écrit de manière simple :

E_p ⟦Terme$_1$ + Terme$_2$⟧ = *Plus* (E_p ⟦Terme$_1$⟧, E_p ⟦Terme$_2$⟧)

E_p ⟦Terme$_1$ - Terme$_2$⟧ = *Moins* (E_p ⟦Terme$_1$⟧, E_p ⟦Terme$_2$⟧)

E_p ⟦Facteur$_1$ * Facteur$_2$⟧ = *Mult* (E_p ⟦Facteur$_1$⟧, E_p ⟦Facteur$_2$⟧)

E_p ⟦Facteur$_1$ / Facteur$_2$⟧ = *Div* (E_p ⟦Facteur$_1$⟧, E_p ⟦Facteur$_2$⟧)

E_p ⟦Variable⟧ = *Valeur* (V_p⟦Variable⟧)

E_p ⟦%nombre⟧ = *Entier* '%nombre'

E_p ⟦(Expression)⟧ = E_p ⟦Expression⟧

E_p ⟦(- Facteur)⟧ = *Opposé* (E_p ⟦Facteur⟧)

IV.2. Structure du langage objet

Le langage objet spécifié ici est une extension de celui qui a été présenté précédemment ; outre les valeurs hexadécimales et les adresses, il permet de manipuler des étiquettes - pour se brancher à des instructions - et également des codes de condition - résultats de tests de comparaison sur des valeurs, utilisés par les instructions de branchement conditionnel -. L'introduction de ces nouveaux types nous amène à modifier l'axiomatisation donnée plus haut pour ce langage objet ; nous allons donc reprendre complètement la description que nous en avons faite.

Une remarque s'impose tout d'abord : les types spécifiées ici, à part le type *modif*, comportent tous une opération d'égalité. Cette opération est nécessaire pour écrire les axiomes conditionnels et les restrictions mais n'a pas d'autre raison d'être ; en effet notre langage objet ne manipule pas d'expressions logiques mais des codes de condition, et cette égalité n'a donc pas de représentant dans le code. Il s'agit là d'une opération auxiliaire du Type Objet. Elle ne peut par conséquent en aucun cas servir à représenter une opération du Type Source. Afin de bien mettre l'accent sur cette propriété fondamentale, nous noterons cette opération *égal* sous la forme infixée =.

Les registres et les adresses pouvant contenir des valeurs hexadécimales, des adresses ou des étiquettes, le type *contenu* va être défini comme l'union de ces types simples. Pour la simplicité de l'exposé, nous supposerons que nous disposons d'un nombre infini de registres.

type *registre* :

op (*registre*) → *registre* : *succ* ;
op (*registre, registre*) → *logique* : *égal* ;

axiomes

$(succ\ (r_1) = succ\ (r_2)) = (r_1 = r_2)$
(registre 'R1' = registre 'R2') = (R1 = R2)

fin *registre* ;

type *contenu* = (*valeur, adresse, étiquette*) :

op (*valeur, valeur*) → *valeur* : *plus, moins, mult, div* ;
op (*adresse, valeur*) → *adresse* : *index* ;
op (*adresse*) → *contenu* : *cm* ;
op (*registre*) → *contenu* : *cr* ;
op (*contenu, contenu*) → *logique* : *égal* ;

axiomes

plus (v, valeur 'o') = v
...
"l'axiome qui définit div sera donné dans le type code-cond"
index (a, valeur 'o') = a
$index\ (index\ (a, v_1), v_2) = index\ (a, plus\ (v_1, v_2))$
...

restrictions

$(v_2 = valeur\ 'o') \Rightarrow$ Echec (*div* ; v_1, v_2)
$(a = adresse\ 'nil') \Rightarrow$ Echec (*index* ; *a, v*)
$(a = adresse\ 'nil') \Rightarrow$ Echec (*cm* ; *a*)

fin *contenu* ;

Le type code de condition, spécifié ci-dessous, permet d'exprimer le résultat de la comparaison de deux valeurs. Lors d'une telle comparaison par l'instruction *comparer*, définie plus loin, ce résultat est contenu dans un "registre interne", auquel on peut accéder par l'opération *cond*. Le type *code-cond* comporte trois constantes, de textes *pp, eg, pg* (pour "plus petit", "égal", "plus grand").

type *code-cond* :

op (*valeur, valeur*) → *code-cond* : *test* ;
op () → *code-cond* : *cond* ;
op (*code-cond, code-cond*) → *logique* : *égal* ;

axiomes

test (v_1, *plus* (v_2, *valeur* '1')) = **si** v_1 = *plus* (v_2, *valeur* '1')
alors *code-cond* 'eg'
sinon si v_1 = v_2
alors *code-cond* 'pp'
sinon *test* (v_1, v_2)

test (v_1, *moins* (v_2, *valeur* '1')) = **si** v_1 = *moins* (v_2, *valeur* '1')
alors *code-cond* 'eg'
sinon si v_1 = v_2
alors *code-cond* 'pg'
sinon *test* (v_1, v_2)

"Nous pouvons maintenant spécifier l'opération *div*, dont le résultat dépend des signes de ses opérandes : "

div (v_1, v_2) = **si** *test* (v_1, *valeur* '0') = *code-cond* 'pg'
alors si *test* (v_2, *valeur* '0') = *code-cond* 'pg'
alors si *test* (*moins* (v_1, v_2), *valeur* '0')
= *code-cond* 'pg'
alors *plus* (*div* (*moins* (v_1, v_2), v_2),
valeur '1')
sinon *valeur* '0'
sinon *moins* (*valeur* '0',
div (v_1, *moins* (*valeur* '0', v_2)))
sinon si *test* (v_2, *valeur* '0') = *code-cond* 'pg'
alors *moins* (*valeur* '0',
div (*moins* (*valeur* '0', v_1), v_2))
sinon *div* (*moins* (*valeur* '0', v_1),
moins (*valeur* '0', v_2))

fin *code-cond* ;

Nous allons maintenant spécifier le type *modif*, qui est le "type d'intérêt" du Type Objet. La difficulté principale dans cette spécification réside dans l'utilisation des étiquettes ; nous définissons ici deux prédicats, *définit* et *utilise*, qui nous permettent de définir assez simplement les branchements ainsi que les étiquetages de modifications. Les branchements ne seront d'ailleurs spécifiés que dans les cas qui sont nécessaires, c'est-à-dire pour les compositions de modifications susceptibles de représenter des *Modifs*.

type *modif* :

op (*registre, contenu*)	→	*modif* : *charger* ;
op (*adresse, contenu*)	→	*modif* : *ranger* ;
op (*valeur, valeur*)	→	*modif* : *comparer* ;
op (*étiquette*)	→	*modif* : *aller-à* ;
op (*code-cond, étiquette*)	→	*modif* : *aller-si* ;
op (*étiquette, modif*)	→	*modif* : *étiqueter* ;
op (*modif, modif*)	→	*modif* : *seq* ;
op ( )	→	*modif* : *nop* ;
op (*étiquette, modif*)	→	*logique* : *définit, utilise*

définitions

charger (*r, c*)	=	**subst** (*cr* (*r*), *c*)
ranger (*a, c*)	=	**subst** (*cm* (*a*), *c*)
comparer (c_1, c_2)	=	**subst** (*cond* (), *test* (c_1, c_2))

"Hormis les cas explicitement examinés ci-dessous, les prédicats *définit* et *utilise* ont partout la valeur *faux* ; intuitivement, *définit* (*e, m*) est vrai si *m* comporte une modification étiquetée par *e*, et *utilise* (*e, m*) indique si *m* comporte un branchement à l'étiquette *e*, non définie dans *m*."

définit (e_1, *étiqueter* (e_2, *m*)) = (e_1 = e_2) ∨ *définit* (e_1, *m*)

définit (*e*, *seq* (m_1, m_2)) = *définit* (*e*, m_1) ∨ *définit* (*e*, m_2)

$utilise\ (e_1,\ aller\text{-}à\ (e_2)) = (e_1 = e_2)$

$utilise\ (e_1,\ aller\text{-}si\ (cc,\ e_2)) = (e_1 = e_2)$

$utilise(e_1,\ étiqueter\ (e_2,\ m)) = \sim (e_1 = e_2) \wedge utilise\ (e_1,\ m)$

$utilise\ (e,\ seq\ (m_1,\ m_2)) = \sim (définit\ (e,\ m_1) \vee définit\ (e,\ m_2)) \wedge (utilise\ (e,\ m_1) \vee utilise\ (e,\ m_2))$

Nous pouvons maintenant spécifier les modifications susceptibles d'intervenir dans les termes objets, nous servant notamment du prédicat *utilise*. Dans les définitions qui suivent, afin de permettre une meilleur lisibilité, nous utiliserons la notation $SEQ\ (m_1,\ ..,\ m_n)$ pour $seq\ (m_1,\ seq(...\ m_n))$. *SEQ* peut être considéré comme une macro-opération, dont la définition serait, suivant le nombre de ses arguments::

$SEQ\ (\ ) = nop\ (\ )$

$SEQ\ (m) = m$

$SEQ\ (m_1,\ ...,\ m_n) = seq\ (m_1,\ SEQ\ (...,\ m_n))$

"Nous spécifions tout d'abord que son étiquetage ne change pas la signification d'une modification : "

$\sim utilise\ (c,\ m) \in S \Longrightarrow \underline{appl}\ (étiqueter\ (e,\ m),\ S) = \underline{appl}\ (m,\ S)$

"Nous allons maintenant traiter les compositions séquentielles."

$(\underline{pour\ tout}\ e,\ \sim utilise\ (e,\ m_1) \in S) \Longrightarrow \underline{appl}\ (seq\ (m_1,\ m_2),\ S) = \underline{appl}\ (m_2,\ \underline{appl}\ (m_1,\ S))$

$\underline{appl}\ (seq\ (étiqueter\ (e,\ m_1),\ m_2),\ S) = \underline{appl}\ (étiqueter\ (e,\ seq\ (m_1,\ m_2)),\ S)$

"Les deux règles ci-dessous définissent les branchements inconditionnels, respectivement pour les instructions conditionnelles et les procédures : "

$\underline{appl}\ (SEQ\ (aller\text{-}a\ (e),\ m_1,\ étiqueter\ (e,\ m_2)),\ S) = \underline{appl}\ (étiqueter\ (e,\ m_2),\ S)$

$(\underline{pour\ tout}\ e,\ \sim utilise\ (e,\ m_2) \in S)$

$\Longrightarrow \underline{appl}\ (SEQ\ (aller\text{-}à\ (e_2),\ étiqueter\ (e_1,\ m_1),\ aller\text{-}à\ (e_3),\ étiqueter\ (e_2,\ m_2),\ aller\text{-}à\ (e_1),\ étiqueter\ (e_3,\ m_3)),\ S)$

$= \underline{appl}\ (SEQ\ (étiqueter\ (e_2,\ m_2),\ étiqueter\ (e_1,\ m_1),\ étiqueter\ e_3,\ m_3)),\ S)$

"Les branchements conditionnels, qui nous serviront à représenter les instructions conditionnelles, sont spécifiés par les deux règles suivantes : "

$cc = cond\ (\ ) \in S$

$\Longrightarrow \underline{appl}\ (SEQ\ (aller\text{-}si\ (cc,\ e),\ m_1,\ étiqueter\ (e,\ m_2)),\ S) = \underline{appl}\ (étiqueter\ (e_1,\ m_2),\ S)$

$\sim (cc = cond\ ()) \in S$

$\Longrightarrow \underline{appl}\ (SEQ\ (aller\text{-}si\ (cc,\ e),\ m_1,\ étiqueter\ (e, m_2)),\ S) = \underline{appl}\ (seq\ (m_1,\ étiqueter\ (e,\ m_2)),\ S)$

"Enfin l'instruction vide :"

$\underline{appl}\ (nop\ (),\ S) = S$

$\underline{fin}\ modif$.

Le Type Objet, que nous allons utiliser pour représenter le langage SIMPROC, est maintenant complet. Nous n'avons pas totalement spécifié les branchements du langage objet, puisque nous ne nous intéressons qu'aux programmes objets susceptibles de représenter des programmes SIMPROC. De même l'environnement des programmes objets a été omis ici, la description des systèmes d'exploitation n'étant pas notre propos : nous ne nous sommes pas donné les éléments permettant de représenter la modification *Initialiser* de SIMPROC, chargée notamment de mettre en place la zone de mémoire libre à l'exécution.

Anticipant un peu sur la dernière partie de ce chapitre, où nous décrivons la représentation adoptée pour les Appels, nous décrivons ici, pour clore cette partie consacrée au langage objet, les deux macro-opérations qui nous sont nécessaires pour sauvegarder et restaurer les registres de base lors des appels de procédure. Ces deux macro-opérations permettent de traiter I registres à partir du registre R , la zone de mémoire utilisée débutant à l'adresse A :

```
RANGER-MULT (A, R, 0) = nop ()
RANGER-MULT (A, R, I) = seq (ranger (A, cr (R)),
                             RANGER-MULT (index (A, valeur '1'), succ (R),
                                          (- I 1)))
```

CHARGER-MULT (R, O, A) = nop ( )
CHARGER-MULT (R, I, A) = seq (charger (R, cm (A)),
CHARGER-MULT (succ (R), (- I 1),
index (A, valeur '1')))

IV.3. Implantation de SIMPROC

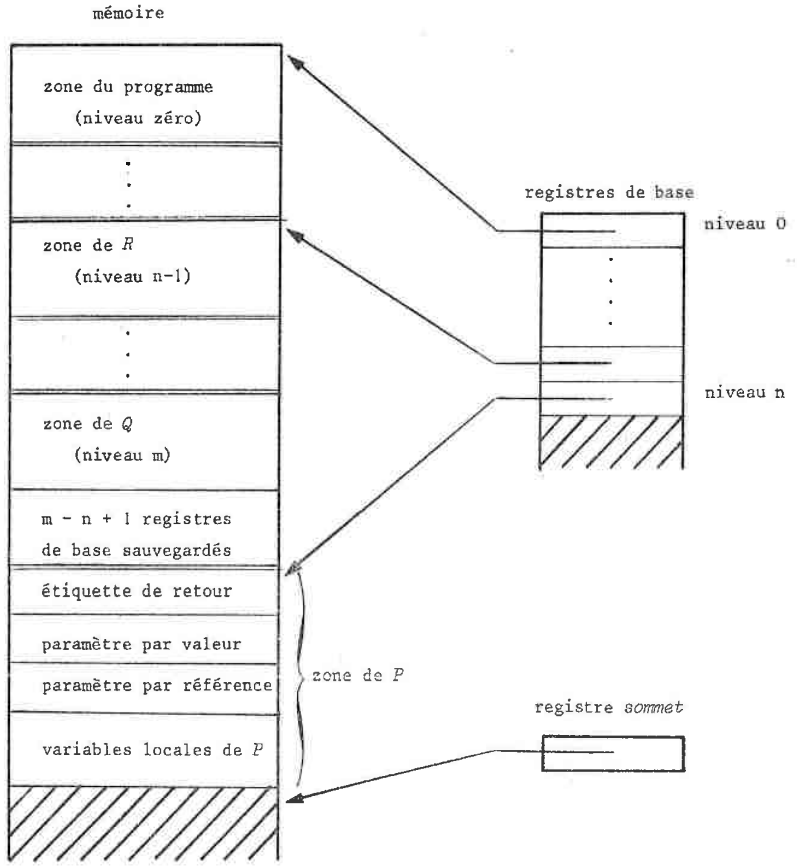
Notre but n'étant pas de prouver que notre implantation est correcte mais de donner un exemple d'une implantation de SIMPROC, nous spécifierons ici les représentations des seules opérations qui apparaissent dans les équations sémantiques. Les autres opérations en effet ne servent qu'à l'axiomatisation et ne peuvent intervenir dans le terme source.

Un des intérêts majeurs de cet exemple est qu'il témoigne de l'aisance avec laquelle nous pouvons spécifier une implantation souvent utilisée dans les compilateurs traditionnels. L'implantation que nous avons choisie pour les appels de procédure est en effet très classique :

- à chaque activation de procédure à l'exécution correspond une zone de mémoire ; la mémoire est gérée comme une pile de telles zones ;
- chacune de ces zones est subdivisée ainsi :
  - . le premier mot contient l'étiquette de retour,
  - . les deux mots suivants contiennent les valeurs des paramètres de la procédure associée à la zone,
  - . le reste de la zone est consacré aux variables locales de cette procédure, puis, le cas échéant, aux sauvegardes des registres de bases effectuées comme décrit ci-dessous ;
- lors de l'exécution de toute procédure, afin de permettre l'accès aux variables globales, un certain nombre de registres, dits de base, contiennent les adresses des zones associées aux activations les plus récentes des procédures englobantes ; si nous définissons son niveau comme le nombre de procédures englobant une procédure, pendant l'exécution d'une procédure de niveau m sont utilisés (m+1) registres de base. Lors de l'appel d'une procédure, ceux de ces registres qui doivent être modifiés pour la procédure appelée sont sauvegardés dans la zone de l'appelant, et leurs valeurs restaurées lors du retour ;

- un registre particulier est utilisé : le registre de nom *sommet* contient l'adresse du premier mot libre au-delà de la dernière de ces zones.

La figure ci-après décrit l'état de la mémoire pendant l'exécution d'une procédure de nom *P*, de niveau n, appelée par une procédure de nom *Q*, de niveau m ; dans cette figure, seuls les liens les plus significatifs ont été matérialisés ; le *Père* de *P* a pour nom *R*.



On voit qu'un appel va être représenté par l'adresse de la zone qui lui est associée, les identificateurs de variables entières ou de tableaux par un déplacement dans cette zone ; les variables seront obtenues, pour chaque appel, par indexation par rapport au début de la zone qui le représente.

Comme au chapitre un, les *Entiers* sont représentés par les *valeurs*.

type *Appel* : *adresse* "celle du début d'une zone"
fin *Appel* ;
type *Id-val* : *valeur* ;
repr *Id-val 'I'* = *valeur* 'ALLOUER-ENTIER (I)'
"de façon similaire à la fonction introduite au chapitre un, cette fonction ALLOUER-ENTIER retourne le déplacement associé à *I* ; si *I* n'existe pas dans la table des identificateurs, il y est introduit."
fin *Id-val* ;
type *Id-var* : *valeur* ;
repr *Id-var 'X'* = *Valeur 'z'*
"le déplacement associé à un paramètre passé par référence est toujours le même".
fin *Id-var* ;
type *Id-tab* : *valeur* ;
repr *Id-tab 'T'* = *Valeur* 'ALLOUER-TABLEAU (*T*, BSUP (*T*))
"BSUP est une fonction d'accès dans une des tables construites lors de la première phase - voir l'attribut DECLARATIONS en annexe E, ligne 241 à 250 en particulier".
fin *Id-tab* ;
type *Id-proc* : *étiquette* ;
"pour repérer le corps de la procédure"
repr *Id-proc 'P'* = *étiquette 'P'*
fin *Id-proc* ;
type *Variable* : *adresse* ;
"obtenue par indexation sauf dans le cas d'un *Ref-var*"
fin *Variable* ;

type *Ref-var* : *adresse* "contient l'adresse d'une variable simple ou d'un élément de tableau passé par référence".
fin *Réf-var* ;
type *Modif* : *modif*
fin *Modif* ;
repr op *(Id-val i , Appel a)* → *Variable : Désigne* =
index (cr (registre BASE *(i))*, repr *i)*
"BASE est une fonction LISP qui retourne le nom du registre de base correspondant au niveau du *Père* d'un identificateur ; on a, de façon informelle :
~ *(id ∈ Id-proc)* ⊃ *registre* BASE *(id) = registre* BASE *(*PERE *(id))*
(idp∈Id-proc) ⊃ *registre* BASE *(idp) = succ (registre* BASE *(*PERE *(idp)))"*
repr op *(Id-var x, Appel a)* → *Ref-var : Ref-Dés* =
index (cr (registre BASE *(x))*, repr *x)*
repr op *(Ref-var r)* → *Variable : Référence = cm (*repr *r)*
repr op *(Id-tab t, Appel a, Entier i)* → *Variable : Elément* =
index (cr (registre BASE *(t)*, *plus (*repr *t*, repr *i))*
repr op *(Variable)* → *Entier : Valeur = cm (*repr *v)*

La spécification donnée ci-dessus. permet de traduire les expressions de SIMPROC. Nous allons maintenant nous intéresser aux intructions, simples ou composées, autres que la déclaration et l'appel de procédure.

repr op *(Id-val i, Id-proc p)* → *Modif : Déclarer-ent* =
charger (registre 'sommet', plus (cr (regitre'sommet'), valeur '1'))
repr op *(Id-tab t, Id-proc p, Entier e)* → *Modif : Déclarer-tab* =
charger (registre 'sommet', plus (cr (registre 'sommet'), repr *e))*
repr op *(Variable v, Entier e)* → *Modif : Affecter* =
*ranger (*repr *v,* repr *e)*
repr op *(Modif* m_1, *Modif* m_2*)* → *Modif : Seq* =
seq (repr m_1, repr m_2*)*

```
repr op (Entier i, Entier j, Modif m1, Modif m2) → Modif : Cond =
    (# local (EGAL, SUITE)
       (SETQ  EGAL  (GENETI))
       (SETQ  SUITE (GENETI))#)

  SEQ (comparer (repr i, repr j),
       aller-si (code-cond 'eg', étiquette EGAL),
       repr m2,
       aller-à (étiquette SUITE),
       étiqueter (étiquette EGAL, repr m1),
       étiqueter (étiquette SUITE, nop ()))

repr op ( ) → Modif : Continuer = nop ()
```

Pour terminer cette spécification de la représentation de SIMPROC, nous donnons maintenant la représentation des modifications qui ont directement trait aux procédures, c'est-à-dire la déclaration et l'appel de procédure.

Rappelons qu'un identificateur de procédure est représenté par une étiquette, qui va repérer le corps de cette procédure ; en outre le premier mot de la zone allouée pour une activation de procédure contient l'étiquette de retour.

```
repr op (Id-proc P, Id-proc Q, Id-val I, Id-var X, Modif corps)
                                                → Modif : Déclarer-proc =
    (# local (SUITE)
       (SETQ  SUITE  (GENETI))#)

                "SUITE nous permet de ne pas exécuter le corps de la
                 procédure lors de l'exécution de sa déclaration".

  SEQ (aller-à (étiquette SUITE),
       étiqueter (repr P, charger (registre BASE (p), cr (registre 'sommet'))),
       charger (registre 'sommet', plus (cr (registre 'sommet'), valeur'3))

                "on vient d'effectuer le changement d'environnement"
       repr corps,
       aller-à (cm (cr (registre BASE (P)))),
       étiqueter (étiquette SUITE, nop ()))
```


Voici enfin la représentation de l'appel de procédure : on a successivement à sauvegarder les registres de base qui vont être modifiés, passer les paramètres - qui sont bien évalués dans l'"ancien environnement" puisque les registres de base sont inchangés -, sauvegarder l'étiquette de retour, se brancher au corps de la procédure appelée - qui, comme on vient de le voir, se charge du changement d'environnement nécessaire -, et enfin, lors du retour, à restaurer les registres de base et le sommet.

```
repr op (Id-proc P, Entier E, Variable V, Id-proc Q) → Modif : Exécuter =
    (# local (N              "différence de niveau entre l'appelant et l'appelé"
              , RETOUR)
       (SETQ   N   (1+ (- (NIVEAU Q) (NIVEAU P))))
       (SETQ   RETOUR  (GENETI)) #)

                    "NIVEAU est une fonction LISP qui calcule le
                     niveau d'une procédure tel qu'il a été défini
                     plus haut : pour le programme, NIVEAU retourne 0
                     et l'on a, de façon informelle :

                          (NIVEAU p) = (1+ (NIVEAU PERE (p)))"

  SEQ (                 "on sauvegarde les registres de base : "
       RANGER-MULT (cr (registre 'sommet'), registre BASE (P), N ),
       charger (registre 'sommet', plus (cr (regsitre ' sommet'), valeur N)),

                    "on passe les paramètres : "

       ranger (index (cr (registre 'sommet'), valeur '1'), repr E),
       ranger (index (cr (registre 'sommet'), valeur '2'), repr V),

                    "on se branche à la procédure après sauvegarde
                     de l'étiquette de retour :"

       ranger (cr (registre 'sommet'), étiquette RETOUR),
       aller-à (repr P),

                    "on restaure, lors du retour, les registres de
                     base précédemment sauvegardés et le sommet :"
```

charger (registre 'sommet', index (cr (registre BASE (p)),
moins (valeur '0', valeur N))),
CHARGER-MULT (registre BASE (p), N, *cr (registre 'sommet')))*

Pour spécifier complètement un compilateur de SIMPROC, il nous suffirait de décrire comment produire le code correspondant aux termes objets engendrés par la représentation que nous venons de décrire ; comme nous l'avons vu dans l'exemple donné au chapitre II de cette thèse, cette phase de production peut se charger de certaines optimisations telles que la suppression des instructions vides (*nop*) ou le remplacement des successions d'incrémentations du registre *sommet*, engendrées par les déclarations, par une seule incrémentation de ce registre pour chaque procédure.

IV.4. Conclusion

Nous espérons que cette spécification d'un compilateur du langage complexe et réaliste qu'est SIMPROC aura convaincu le lecteur de l'intérêt et de l'efficacité du système Perluette. La méthode de description sémantique des langages de programmation implantée par Perluette, basée sur les schémas fonctionnels, permet d'obtenir des spécifications aisément lisibles, comme en témoigne cet exemple ; de plus des concepts présents dans d'autres langages de programmation, comme les fonctions, peuvent être facilement introduits dans une telle spécification : ceci est réalisé dans [Gau 80], ainsi que dans [Pai 80] , où il est montré en outre comment on peut se ramener à une sémantique plus classique, moins orientée vers la compilation, en incorporant les états dans les profils des opérations. Un autre intérêt, et non des moindres, de la sémantique utilisée ici est la possibilité de preuve de la représentation choisie, comme nous l'avons vu au chapitre I.

Enfin l'exemple que nous venons de développer montre bien comment on peut utiliser le métalangage présenté au chapitre II pour décrire des traitements à effectuer lors de la traduction, ces traitements étant similaires à ceux qu'effectue un compilateur produit de manière classique.

Cette seconde phase, conçue comme la réécriture d'un terme par son interprétation directe, justifie l'emploi de LISP comme langage d'implantation des compilateurs produits ; ce choix ne doit d'ailleurs pas être un obstacle à une utilisation "semi-industrielle" de Perluette, puisque les fonctions qui réalisent cette interprétation peuvent être compilées.

En fait les problèmes posés par la réalisation d'un compilateur au moyen de Perluette résident surtout dans la mise en oeuvre de la méthode de description des langages de programmation : établir le type abstrait associé à un langage n'est pas toujours chose facile pour un programmeur moyen, et notamment la distinction entre les aspects statiques et dynamiques du langage source est souvent délicate à faire.

Cette méthode a néanmoins le mérite de faciliter la conception d'un traducteur et de permettre la preuve des choix d'implantation effectués, preuve qui doit d'ailleurs être automatisable, du moins en partie. Elle permet en outre, tant par l'utilisation de descriptions séparées des langages source et objet que par les différents niveaux introduits dans ces descriptions, d'améliorer l'architecture des compilateurs et donc leur fiabilité.

ANNEXES

Les descriptions données ici utilisent un métalangage proche de celui qui a été introduit par Backus et Naur. Par convention, les non-terminaux y apparaissent entre chevrons (< et >), les terminaux génériques sont précédés du caractère "pour-cent" (%) et les autres terminaux sont désignés par leur texte encadré par des symboles "double-apostrophe" ("). Nous utilisons en outre les symboles suivants :

=	est défini par
|	ou
;	fin de définition
[x]	éventuellement "x" ("x" peut apparaître à cet endroit)
{x}	un nombre quelconque de fois "x" (éventuellement nul)
(x)	parenthèsage (mise en facteur)

Les terminaux génériques sont par ailleurs décrits au moyen d'expressions régulières utilisant les mêmes symboles conventionnels.

Sommaire :

A :	syntaxe des types abstraits
B :	syntaxe de la représentation
C :	les formes LISP acceptées
D :	exemple d'utilisation de DELTA
E :	description de SIMPROC
F :	métalangage accepté par &I.I.

A - ANNEXE

SYNTAXE DES TYPES ABSTRAITS

La grammaire donnée ici est une forme simplifiée de celle qui a servi à implanter les modules traitant les types abstraits source et objet. Le terminal %id (qui représente les identificateurs) est décrit au niveau lexical par l'expression régulière :

```
%id =     LETTRE  {[SOULIGNE] (LETTRE | CHIFFRE)} ;

<description d'un type abstrait>  =  <type abstrait>
                                        {";" <type abstrait>} "." ;
<type abstrait>     =  "type"  %id  [<union>]  ":"
                            <opérations>
                            { ";" <opérations>}
                       "fin"  %id ;
<union>             =     "=" "union" "(" <liste d'id.>  ")"  ;
<liste d'id.>       =     %id {"," %id} ;
<opérations>        =     <profil> ":" <liste d'id.> .
<profil>            =     "op" "(" [< liste d'id.>] ")" "→" %id ;
```

La grammaire réelle qui a servi à implanter Perluette diffère de celle-ci par une plus grande souplesse dans l'utilisation du symbole ";" ; l'utilisateur peut en effet le considérer soit comme un séparateur soit comme un terminateur.

B - ANNEXE

SYNTAXE DE LA REPRESENTATION

La grammaire donnée ci-dessous ne décrit que partiellement la syntaxe utilisée pour un traducteur donné, puisque les types abstraits source et objet interviennent dans cette syntaxe. En outre les formes LISP sont décrites ailleurs - voir l'annexe C.

Les conventions utilisées sont celles qui ont été décrites plus haut. Les non-terminaux <type source> , <opération à représenter> et <opération source> sont décrits dans TAB 1, engendré par Perluette 1, alors que Perluette 3 produit de son côté TAB 3, dans lequel se trouvent les définitions des non-terminaux <type objet> et <opération objet>. Les terminaux génériques utilisés ici sont %id et %nombre ; ils sont décrits au niveau lexical de la manière suivante :

```
%id =    LETTRE {[SOULIGNE] (LETTRE | CHIFFRE)} ;
%nombre =     CHIFFRE {CHIFFRE}

<description de la représentation> =
            <représentation d'un type>
                 {";" <représentation d'un type>} ";"
            <représentation d'une opération>
                 {";" <représentation d'une opération>} "." ;
<représentation d'un type>          =
            "type" <type source> ":"
                 <spécification du type>
            "fin"  <type source> ;
<spécification du type>             =
            <spécification unique>
                 [<représentation unique des constantes>]
            | <spécification multiple>
                 [<représentation multiple des constantes>] ;
```

```
<spécification unique> =
        <type objet> ;
<spécification multiple> =
        "(" <type objet> "," <type objet> {"," <type objet>} ")" ;
<représentation unique des constantes> =
        ";" "repr" <type source> "'" %id "'" "=" <constante objet> ;
<constante objet> =
        <type objet> "'" ( %id | <appel de fonction>) "'" ;
<appel de fonction> =
        %id "(" [<arguments>] ")" ;
<arguments> =
        <argument> {"," <argument>} ;
<argument> =
        %id | %nombre | <appel de fonction> ;
<représentation multiple des constantes> =
        ";" "repr" <type source> "'" %id "'"
                "=" "(" <constante objet>
                        "," <constante objet>
                        {"," <constante objet>} ")" ;
<représentation d'une opération> =
        "repr" <opération à représenter> "="
               <spécification de l'opération>
        "fin" <opération source> ;
<spécification de l'opération> =
        [ "(# " <méta-instructions> "#)"] <spécification> ;
<méta-instructions> =
        [<méta-déclaration>] {<méta-instruction>} ;
<méta-déclaration> =
        "local" "(" %id {"," %id} ")" ;
<méta-instruction> =
        <forme LISP>
<spécification> =
        <spécification unique> | <spécification multiple> ;
```



```
<spécification unique> =
        <terme objet>
        | <test>
        | "repr" ["." %nombre] <terme source>
        | <méta-appel de procédure> ;
<spécification multiple> =
        "(" <spécification unique>
            "," <spécification unique>
            {"," <spécification unique>} ")"
        | "si" <forme LISP>
          "alors" <spécification multiple>
          "sinon" <spécification multiple> ;
<terme objet> =
        <opération objet> "(" [<sous-termes objets>] ")" ;
<sous-termes objets> =
        <sous-termes objet> {"," <sous-terme objet>} ;
<sous-terme objet> =
        <spécification unique> | <constante objet> ;
<test> =
        "si" <forme LISP>
        "alors" <spécification unique>
        "sinon" <spécification unique> ;
<terme source> =
        <opération source> "(" [<sous-termes sources>] ")" ;

<sous-termes sources> =
        <sous-terme source> {"," <sous-terme source>} ;
<sous-terme source> =
        <terme-source> | <constante source> ;
<méta-appel de procédure> =
        <forme LISP> ;
```

C - ANNEXE

LES FORMES LISP ACCEPTEES

La grammaire suivante décrit les formes LISP analysées par Perluette ; elles diffèrent d'un LISP "standard" sur deux points : d'une part un atome peut être, outre un identificateur, un nom d'opération du type abstrait ou un attribut destiné à DELTA, et d'autre part l'utilisation de deux types de parenthèses permet d'éviter les listes de parenthèses fermantes usuelles dans les programmes LISP et génératrices d'erreurs. Pour ce deuxième point en effet les formes considérées sont encadrées soit par des parenthèses (et) soit par des chevrons < et >, une parenthèse fermante permettant donc de clore un nombre quelconque de chevrons ouvrants et réciproquement.

Pour cette grammaire, le terminal %atome est décrit au niveau lexical par l'expression régulière :

```
%atome = LETTRE {[SOULIGNE | MOINS] (LETTRE | CHIFFRE)}
```

De plus les non-terminaux <operation source> et <type source>, comme dans l'annexe précédente, sont décrits pour chaque langage dans la partie de grammaire engendrée par Perluette 1.

```
<forme LISP>  =  <atome>
                 | "("   {<forme LISP>}  {"<" {<forme LISP>}} ")"
                 | "<"   {<forme LISP>}  {"(" {<forme LISP>}} ">" ;

<atome>       = %atome
                 | <attribut>
                 | <operation source>
                 | <type source> ;
```


D - ANNEXE

EXEMPLE D'UTILISATION DE DELTA

Cet exemple est présenté dans l'introduction, au paragraphe 3.

```
V0501 UUUP00 MM/CCC/QQQ
EXECUTION F4LIB VERSION V10-05 14*08*09*

TABULATION SYNTAXE ET SEMANTIQUE

   VERSION DU 24/03/80

1     D=
2     (PRETTY
3        (PROG (
4     $
5     I=                )
6
7     $
8     F=
9                  )
10    )
11    $
12                 S#RETOUR(<AXIOME>);
13                 S#RESULTAT(<S-EXPR>, <OPERANDES>)=
14                        #
15                 H#OPERATION(<OPERANDES>)=
16                        #
17                 $

18    1 <AXIOME>    =    <S-EXPR>  ;
19                #RETOUR(<AXIOME>)
20                   (RETURN RESULTAT(<S-EXPR>))

21    2 <S-EXPR>    =    (  %Operateur  <S-EXPR>  <OPERANDES>  )  ;
22                #RESULTAT(<S-EXPR>)
23                   (SETQ RESULTAT(<S-EXPR>)
24                      (APPEND
25                         (LIST RESULTAT(<S-EXPR>')  (QUOTE PTEXT( %Operateur )) )
26                         RESULTAT(<OPERANDES>)) )
27                #OPERATION(<OPERANDES>)
28                   (SETQ OPERATION(<OPERANDES>) (QUOTE PTEXT( %Operateur )))
```

```
 29        3 <S-EXPR>     =      %Atome  ;
 30                #RESULTAT(<S-EXPR>)
 31                      (SETQ RESULTAT(<S-EXPR>) (QUOTE PTEXT( %Atome )))

 32        4 <OPERANDES> =      <S-EXPR>  ;
 33                #RESULTAT(<OPERANDES>)
 34                      (SETQ RESULTAT(<OPERANDES>) (LIST RESULTAT(<S-EXPR>)) )

 35        5 <OPERANDES> =      <OPERANDES>  <S-EXPR>  ;
 36                #RESULTAT(<OPERANDES>)
 37                      (SETQ RESULTAT(<OPERANDES>)
 38                            (APPEND RESULTAT(<OPERANDES>)
 39                                (LIST OPERATION(<OPERANDES>) RESULTAT(<S-EXPR>)) ))



 40                $
 41                $

LE CONSTRUCTEUR A TRAITE     7 EQUATIONS SEMANTIQUES
DONT  14% D'IDENTITES ET  14% DE DEFAUTS

TAUX DE REMPLISSAGE DES TABLES

 NO      XMAX      LMAX       %

  1         5       150       3
  2         4       100       4
  3        61      1000       6
  4        18      1000       1
  5         3       100       3
  6        53       500      10
  7         7       600       1
  8         6       150       4
  9        18      1000       1
 10        11      1016       1
 11         2        50       4
 12         2       100       2
*STOP* 0
```



```
 V1502 U62P02 05/023/171
 FILE IS NOT FOUND
  STEP 01 TERMINATED
 V1502 U62P02 05/023/171
 FILE IS NOT FOUND
  STEP 02 TERMINATED
 V0501 UUUP00 MM/CCC/QQQ
 EXECUTION F4LIB VERSION V10-05 14*14*42*



 ANALYSEUR AVEC SEMANTIQUE

    VERSION DU 15/11/79



 TEXTE SOURCE :

    1 : (+ (* A B C) (- D E (/ F G H)) (* I J) K)

  ANALYSE TERMINEE


 NOMBRE DE RELATIONS DE DEPENDANCE =     52
 FIN ANALYSE : RECAPITULATION                ATTRIBUTS GAIN
 TOTAL THEORIQUE :                              37
 APRES ELIMINATION PARTIELLE IDENTITES :        33     10%
 APRES ELIMINATION TOTALE IDENTITES :           32     13%
 *STOP* 0
  IN STEP 03. COMMAND  TERMINATED
```

```
V1502 U62P02 05/023/171
FILE IS NOT FOUND
 STEP 01 TERMINATED
V0501 UUUP00 MM/CCC/QQQ
EXECUTION F4LIB VERSION V10-05 14*17*50*



TRI DESCENDANT A ALLOCATION DIFFEREE

         VERSION DU 19/11/79



FIN TRI : RECAPITULATION                      ATTRIBUTS GAIN
TOTAL THEORIQUE :                                    37
APRES ELIMINATION PARTIELLE IDENTITES :              33    10%
APRES ELIMINATION TOTALE IDENTITES :                 32    13%
APRES ELIMINATION IMPASSES :                         32    13%
APRES MINIMISATION MEMOIRE :                         10    72%
*STOP* 0
 IN STEP 02. COMMAND  TERMINATED
!
```



```
(PRETTY
      (PROG (
       I001
       I002
       I003
       I004
       I005
       I006
       I007
       I008
       I009
            )
      (SETQ I001 (QUOTE A))
      (SETQ I002 (QUOTE B))
      (SETQ I003 (LIST I002) )
      (SETQ I002 (QUOTE *))
      (SETQ I004 (QUOTE C))
      (SETQ I005
            (APPEND I003
                (LIST I002 I004) ))
      (SETQ I002
            (APPEND
               (LIST I001  (QUOTE *) )
               I005) )
      (SETQ I001 (QUOTE D))
      (SETQ I003 (QUOTE E))
      (SETQ I004 (LIST I003) )
      (SETQ I003 (QUOTE -))
      (SETQ I005 (QUOTE F))
      (SETQ I006 (QUOTE G))
      (SETQ I007 (LIST I006) )
      (SETQ I006 (QUOTE /))
      (SETQ I008 (QUOTE H))
      (SETQ I009
            (APPEND I007
                (LIST I006 I008) ))
      (SETQ I006
            (APPEND
               (LIST I005  (QUOTE /) )
               I009) )
      (SETQ I005
            (APPEND I004
                (LIST I003 I006) ))
```

```
        (SETQ I003
              (APPEND
                 (LIST I001  (QUOTE -) )
                 I005) )
        (SETQ I001 (LIST I003) )
        (SETQ I003 (QUOTE +))
        (SETQ I004 (QUOTE I))
        (SETQ I005 (QUOTE J))
        (SETQ I006 (LIST I005) )
        (SETQ I005
              (APPEND
                 (LIST I004  (QUOTE *) )
                 I006) )
        (SETQ I004
              (APPEND I001
                   (LIST I003 I005) ))
        (SETQ I001 (QUOTE K))
        (SETQ I005
              (APPEND I004
                   (LIST I003 I001) ))
        (SETQ I001
              (APPEND
                 (LIST I002  (QUOTE +) )
                 I005) )
        (RETURN I001)
        )
)


!LISP

*** LISP/IRIS80 ***
?(STATUS 4 80)
71
?(LIRE 'PHSEXP1-OUT T)
((A * B * C) + (D - E - (F / G / H)) + (I * J) + K)
```



```
V0501 UUUP00 MM/CCC/Q0Q
EXECUTION F4LIB VERSION V10-05 13*48*53*


TABULATION SYNTAXE ET SEMANTIQUE

      VERSION DU 24/03/80
```



```
     1      D=
     2             (EVAL (PROG (
     3      $
     4      I=
     5                          )
     6      $
     7      F=
     8                    )
     9                    (QUOTE ((TYPE (LAMBDA (ID TABLE)
    10                                         (CADDR (ASSOC ID TABLE)) ))
    11                            (INTERSECTION (LAMBDA (L1 L2)
    12                                          (COND (L1 (APPEND
    13                                                      (COND ((ASSOC (CAR L1) L2)
    14                                                             (LIST (CAR L1)) ))
    15                                                      (INTERSECTION (CDR L1) L2) ))) ))
    16                                 )
    17                    )
    18             )
    19      $
    20                 *
    21                 * CET ATTRIBUT LANCE LES SECONDE ET TROISIEME PHASES.
    22                 *
    23                 S#     TRADUIRE(
    24                               <PROGRAMME>
    25                               )      =
    26                                      #
```

```
27              *
28              * CET ATTRIBUT PERMET DE PROPAGER LA VALIDITE DU PROGRAMME VERS LES MODELES
29              * SEMANTIQUES, DE FACON A POUVOIR ARRETER LA PREMIERE PHASE SI L'UNE
30              * QUELCONQUE DES VERIFICATIONS CONTEXTUELLES ECHOUE.
31              *
32              G#     COMPILER(
33                            <DECLARATION*>
34                            , <DECLARATION+>
35                            , <DECLARATION>
36                            , <INSTRUCTION+>
37                            , <INSTRUCTION>
38                            , <VARIABLE>
39                            , <EXPRESSION>
40                            , <TERME>
41                            , <FACTEUR>
42                            )      =
43                                    #
44              *
45              * CET ATTRIBUT PERMET D'EFFECTUER LES VERIFICATIONS CONTEXTUELLES.
46              *
47              S#     VALIDE(
48                            <PROGRAMME>
49                            , <DECLARATION*>
50                            , <DECLARATION+>
51                            , <DECLARATION>
52                            , <LISTE D'IDENT>
53                            , <INSTRUCTION+>
54                            , <INSTRUCTION>
55                            , <VARIABLE>
56                            , <EXPRESSION>
57                            , <TERME>
58                            , <FACTEUR>
59                            )      =
60                                    #
```



```
61              *
62              * "ATTRIBUT D'INTERET" :
63              * CONTIENT LE TEXTE INTERMEDIAIRE ENGENDRE (T1) QUI SERA LA DONNEE DE LA
64              * SECONDE PHASE.
65              *
66              S#     MODELE(
67                            <PROGRAMME>
68                            , <DECLARATION*>
69                            , <DECLARATION+>
70                            , <DECLARATION>
71                            , <INSTRUCTION+>
72                            , <INSTRUCTION>
73                            )      =      MODIF
74                                    #
75              *
76              * CONTIENT UN TERME DE TYPE ENTIER (VALEUR D'UNE EXPRESSION).
77              *
78              S#     TERME_ENTIER(
79                            <EXPRESSION>
80                            , <TERME>
81                            , <FACTEUR>
82                            )      =      ENTIER
83                                    #
84              *
85              * TERMES DU TYPE VAR.
86              *
87              S#     TERME_VAR(
88                            <VARIABLE>
89                            )      =      VAR
90                                    #
91              *
92              * OPERATION ASSOCIEE A UN OPERATEUR.
93              *
94              S#     OPERATION(
95                            <PLUS_MOINS>
96                            , <MULT_DIV>
97                            )      =      OP (ENTIER, ENTIER) -> ENTIER
98                                    #
```

```
 99              *
100              *
101              * GESTION DE LA TABLE DES SYMBOLES
102              * ------------------------------
103              *
104              *
105              * AJOUTS FAITS PAR LES DECLARATIONS INTERNES A UNE PROCEDURE A LA TABLE DES
106              * SYMBOLES QU'ELLE A HERITEE.
107              *
108              S#     DECLARATIONS(
109                           <PROGRAMME>
110                           , <DECLARATION*>
111                           , <DECLARATION+>
112                           , <DECLARATION>
113                           )      =
114                                  #
115              *
116              * TABLE DES SYMBOLES UTILISABLES (HERITEE).
117              *
118              H#     VISIBLES(
119                           <DECLARATION*>
120                           , <DECLARATION+>
121                           , <DECLARATION>
122                           , <INSTRUCTION+>
123                           , <INSTRUCTION>
124                           , <VARIABLE>
125                           , <EXPRESSION>
126                           , <TERME>
127                           , <FACTEUR>
128                           )      =
129                                  #
```



```
130              *
131              * SYMBOLES DEJA DECLARES DEPUIS LE DEBUT TEXTUEL DU PROGRAMME (AFIN DE
132              * VERIFIER QUE TOUS LES IDENTIFICATEURS SONT DIFFERENTS).
133              *
134              M#     DEJA_DECLARES(
135                           <PROGRAMME>
136                           , <DECLARATION*>
137                           , <DECLARATION+>
138                           , <DECLARATION>
139                           )      =
140                                  #
141              *
142              * CONTIENT LE TEXTE DE L'IDENTIFICATEUR DE PROCEDURE ENGLOBANT LE NON-TERMINAL.
143              * POUR LES DECLARATIONS, PERE(ID) = IDPROC 'ENGLOBANT(<DECLARATION>)'
144              *
145              H#     ENGLOBANT(
146                           <DECLARATION*>
147                           , <DECLARATION+>
148                           , <DECLARATION>
149                           , <LISTE D'IDENT>
150                           , <INSTRUCTION+>
151                           , <INSTRUCTION>
152                           , <VARIABLE>
153                           , <EXPRESSION>
154                           , <TERME>
155                           , <FACTEUR>
156                           )      =
157                                  #
158              *
159              * LISTE DES IDENTIFICATEURS APPARAISSANT DANS UNE DECLARATION
160              * D'ENTIER OU DE TABLEAU.
161              *
162              S#     LISTE(
163                           <LISTE D'IDENT>
164                           )      =
165                                  #
166              $
```

```
1 <PROGRAMME>        ==     PROGRAMME  %IDENT  :
                                  <DECLARATION+>
                                  <INSTRUCTION+>
        #DECLARATIONS(<PROGRAMME>)
              =      (QUOTE (TEXT(%IDENT )))
        #VISIBLES(<DECLARATION+>) = DECLARATIONS(<DECLARATION+>)
        #VISIBLES(<INSTRUCTION+>) = DECLARATIONS(<DECLARATION+>)
        #H.DEJA_DECLARES(<DECLARATION+>) = DECLARATIONS(<PROGRAMME>)
        #S.DEJA_DECLARES(<PROGRAMME>) = S.DEJA_DECLARES(<DECLARATION+>)
        #ENGLOBANT(<DECLARATION+>)
              =      (QUOTE TEXT(%IDENT ))
        #ENGLOBANT(<INSTRUCTION+>) = ENGLOBANT(<DECLARATION+>)
        #VALIDE(<PROGRAMME>)
              =      (COND ((AND VALIDE(<DECLARATION+>) VALIDE(<INSTRUCTION+>)) T)
                           (T (STOP (QUOTE (ERREURS EN PREMIERE PHASE:
                                                ABANDON DE LA COMPILATION )))) )
        #COMPILER(<DECLARATION+>) = VALIDE(<PROGRAMME>)
        #COMPILER(<INSTRUCTION+>) = VALIDE(<PROGRAMME>)
        #TRADUIRE(<PROGRAMME>)
              =      (RETURN (PHASE3 (PHASE2 MODELE(<PROGRAMME>) )))
        #MODELE(<PROGRAMME>)
              =      CONCAT (INIT,
                            CONCAT (MODELE(<DECLARATION+>), MODELE(<INSTRUCTION+>) ))
```



```
2 <DECLARATION*>     =
        #DECLARATIONS(<DECLARATION*>)
              =      NIL
        #VALIDE(<DECLARATION*>)
              =      T
        #MODELE(<DECLARATION*>)
              =      VIDE

3 <DECLARATION*>     =      <DECLARATION+>

4 <DECLARATION+>     =      <DECLARATION>

5 <DECLARATION+>     =      <DECLARATION+>  <DECLARATION>
        #DECLARATIONS(<DECLARATION+>)
              =      (APPEND  DECLARATIONS(<DECLARATION>)  DECLARATIONS(<DECLARATION+>'))
        #VALIDE(<DECLARATION+>)
              =      (AND  VALIDE(<DECLARATION+>')  VALIDE(<DECLARATION>))
        #MODELE(<DECLARATION+>)
              =      CONCAT (MODELE(<DECLARATION+>'), MODELE(<DECLARATION>))
```

```
206     6 <DECLARATION>     =     ENTIER  <LISTE D'IDENT>  #;                    ;
207           #DECLARATIONS(<DECLARATION>)
208                 =     (PROG (L DECL TDS)
209                             (SETQ L  LISTE(<LISTE D'IDENT>) )
210                             (SETQ DECL (LIST  ENGLOBANT(<DECLARATION>) (QUOTE TVAL)))
211                             (WHILE L
212                                   (SETQ TDS (CONS (CONS (CAR L) DECL) TDS))
213                                   (SETQ L (CDR L)) )
214                             (RETURN TDS) )
215           #S.DEJA_DECLARES(<DECLARATION>)
216                 =     (APPEND  LISTE(<LISTE D'IDENT>)  H.DEJA_DECLARES(<DECLARATION>) )
217           #VALIDE(<DECLARATION>)
218                 =     (AND (NULL
219                             (PROG (L)
220                                   (RETURN
221                                    (COND ((SETQ L (INTERSECTION
222                                                     LISTE(<LISTE D'IDENT>)
223                                                     H.DEJA_DECLARES(<DECLARATION>) ))
224                                           (PRINT (CONS ENGLOBANT(<DECLARATION>)
225                                                        (APPEND
226                                                         (QUOTE (: REDECLARATION DE : ))
227                                                         L) )) )) )))
228                             VALIDE(<LISTE D'IDENT>) )
229           #MODELE(<DECLARATION>): COMPILER(<DECLARATION>)
230                 =     ACCES CALCULE
231                 (PROG (L M)
232                       (SETQ L  LISTE(<LISTE D'IDENT>) )
233                       (SETQ M  DECL_VAR (ID_VAL (CAR L),
234                                          ENGLOBANT(<DECLARATION>) ))
235                       (WHILE (SETQ L (CDR L))
236                             (SETQ M  CONCAT (M,
237                                              DECL_VAR (ID_VAL (CAR L),
238                                                        ENGLOBANT(<DECLARATION>) ))))
239                       (RETURN M) )
```



```
240     7 <DECLARATION>     =     TABLEAU  <LISTE D'IDENT>  [  %NOMBRE  ]  #;          ;
241           #DECLARATIONS(<DECLARATION>)
242                 =     (PROG (L DECL TDS)
243                             (SETQ L  LISTE(<LISTE D'IDENT>) )
244                             (SETQ DECL (LIST  ENGLOBANT(<DECLARATION>)
245                                               (QUOTE TTAB)
246                                               (QUOTE TEXT(%NOMBRE )) ))
247                             (WHILE L
248                                   (SETQ TDS (CONS (CONS (CAR L) DECL) TDS) )
249                                   (SETQ L (CDR L)) )
250                             (RETURN TDS) )
251           #S.DEJA_DECLARES(<DECLARATION>)
252                 =     (APPEND  LISTE(<LISTE D'IDENT>)  H.DEJA_DECLARES(<DECLARATION>) )
253           #VALIDE(<DECLARATION>)
254                 =     (AND (NULL
255                             (PROG (L)
256                                   (RETURN
257                                    (COND ((SETQ L (INTERSECTION
258                                                     LISTE(<LISTE D'IDENT>)
259                                                     H.DEJA_DECLARES(<DECLARATION>) ))
260                                           (PRINT (CONS ENGLOBANT(<DECLARATION>)
261                                                        (APPEND
262                                                         (QUOTE (: REDECLARATION DE : ))
263                                                         L) )) )) )))
264                             VALIDE(<LISTE D'IDENT>) )
265           #MODELE(<DECLARATION>): COMPILER(<DECLARATION>)
266                 =     ACCES CALCULE
267                 (PROG (L M)
268                       (SETQ L  LISTE(<LISTE D'IDENT>) )
269                       (SETQ M  DECL_TAB (ID_TAB (CAR L),
270                                          ENTIER 'TEXT(%NOMBRE )',
271                                          ENGLOBANT(<DECLARATION>) )
272                       (WHILE (SETQ L (CDR L))
273                             (SETQ M  CONCAT (M,
274                                              DECL_TAB (ID_TAB (CAR L),
275                                                        ENTIER 'TEXT(%NOMBRE )',
276                                                        ENGLOBANT(<DECLARATION>) ))))
277                       (RETURN M) )
```

```
 8 <DECLARATION>     ==    PROC  %IDENT  (  VAL  %IDENT  ,  VAR  %IDENT  )  :
                           <DECLARATION*>
                           <INSTRUCTION+>
                           FIN  PROC  #;                                              ;
       #DECLARATIONS(<DECLARATION>)
             =     (LIST (CONS (QUOTE TEXT(%IDENT ))
                           (CONS  ENGLOBANT(<DECLARATION>)
                                  (QUOTE (TPROC TEXT(%IDENT ´) TEXT(%IDENT ´´) ))))
       #VISIBLES(<DECLARATION*>)
             =     (APPEND (APPEND (QUOTE ((TEXT(%IDENT ´)  TEXT(%IDENT ) TVAL)
                                           (TEXT(%IDENT ´´) TEXT(%IDENT ) TVAR) ))
                                DECLARATIONS(<DECLARATION*>)
                           VISIBLES(<DECLARATION>) )
       #VISIBLES(<INSTRUCTION+>) = VISIBLES(<DECLARATION*>)
       #H.DEJA_DECLARES(<DECLARATION*>)
             =     (APPEND (QUOTE (TEXT(%IDENT ) TEXT(%IDENT ´) TEXT(%IDENT ´´) ))
                           H.DEJA_DECLARES(<DECLARATION>) )
       #ENGLOBANT(<DECLARATION*>)
             =     (QUOTE  TEXT(%IDENT ))
       #ENGLOBANT(<INSTRUCTION+>) = ENGLOBANT(<DECLARATION*>)
       #VALIDE(<DECLARATION>)
             =     (AND (NULL
                         (PROG (L)
                               (RETURN
                                (COND ((SETQ L (INTERSECTION
                                                (QUOTE (TEXT(%IDENT )
                                                        TEXT(%IDENT ´)
                                                        TEXT(%IDENT ´´) ))
                                                H.DEJA_DECLARES(<DECLARATION>) ))
                                       (PRINT (APPEND
                                                (QUOTE (TEXT(%IDENT ) :
                                                        REDECLARATION DE :))
                                                L) ) )))))
                         VALIDE(<DECLARATION*>)
                         VALIDE(<INSTRUCTION+>) )
       #MODELE(<DECLARATION>)
             =     DECL_PROC (ID_PROC ´TEXT(%IDENT )´,
                              ID_VAL ´TEXT(%IDENT ´)´,
                              ID_VAR´TEXT(%IDENT ´´)´,
                              ID_PROC  ENGLOBANT(<DECLARATION>),
                              CONCAT (MODELE(<DECLARATION*>), MODELE(<INSTRUCTION+>) ))
```



```
 9 <LISTE D´IDENT>   =     %IDENT                                                   ;
       #VALIDE(<LISTE D´IDENT>)
             =     T
       #LISTE(<LISTE D´IDENT>)
             =     (QUOTE (TEXT(%IDENT ) ))

10 <LISTE D´IDENT>   =     <LISTE D´IDENT>  ,  %IDENT                               ;
       #VALIDE(<LISTE D´IDENT>)
             =     (AND (NULL
                         (COND ((INTERSECTION
                                 (QUOTE (TEXT(%IDENT )))
                                 LISTE(<LISTE D´IDENT>´) )
                                (PRINT (CONS  ENGLOBANT(<LISTE D´IDENT>)
                                              (QUOTE (: DOUBLE DECLARATION DE
                                                        TEXT(%IDENT )       )) ))) ))
                         VALIDE(<LISTE D´IDENT>´) )
       #LISTE(<LISTE D´IDENT>)
             =     (CONS (QUOTE TEXT(%IDENT ))  LISTE(<LISTE D´IDENT>´) )

11 <INSTRUCTION+>    =     <INSTRUCTION>                                            ;


12 <INSTRUCTION+>    =     <INSTRUCTION+>  #;  <INSTRUCTION>                        ;
       #VALIDE(<INSTRUCTION+>)
             =     (AND  VALIDE(<INSTRUCTION+>´)  VALIDE(<INSTRUCTION>) )
       #MODELE(<INSTRUCTION+>)
             =     CONCAT (MODELE(<INSTRUCTION+>´), MODELE(<INSTRUCTION>) )
```

```
13 <INSTRUCTION>     =     <VARIABLE>  :=  <EXPRESSION>                                    ;
        #VALIDE(<INSTRUCTION>)
              =     (AND  VALIDE(<VARIABLE>)  VALIDE(<EXPRESSION>) )
        #MODELE(<INSTRUCTION>)
              =     AFFECTER_VAR (TERME_VAR(<VARIABLE>),  TERME_ENTIER(<EXPRESSION>) )
```

```
14 <INSTRUCTION>     =     EXECUTER  %IDENT  (  <EXPRESSION>  ,  <VARIABLE>  )   ;
        #VALIDE(<INSTRUCTION>)
              =     (AND (NULL
                         (COND ((NULL (INTERSECTION
                                        (QUOTE (TEXT(%IDENT )))
                                        VISIBLES(<INSTRUCTION>) ))
                                (PRINT (CONS ENGLOBANT(<INSTRUCTION>)
                                        (QUOTE (: TEXT(%IDENT ) EST
                                                  INACCESSIBLE)))))
                               ((NULL (EQ (TYPE (QUOTE TEXT(%IDENT ))
                                             VISIBLES(<INSTRUCTION>) )
                                        (QUOTE TPROC) ))
                                (PRINT (CONS ENGLOBANT(<INSTRUCTION>)
                                        (QUOTE (: TEXT(%IDENT ) N'EST PAS UNE
                                                  PROCEDURE)) ))) ))
                        VALIDE(<EXPRESSION>)
                        VALIDE(<VARIABLE>)      )
        #MODELE(<INSTRUCTION>)
              =     APPEL_PROC (ID_PROC 'TEXT(%IDENT )',
                                TERME_ENTIER(<EXPRESSION>),
                                TERME_VAR(<VARIABLE>),
                                ID_PROC  ENGLOBANT(<INSTRUCTION>)  )
```



```
15 <INSTRUCTION>     ==    SI  <EXPRESSION>  EGALE  <EXPRESSION>
                             ALORS  <INSTRUCTION+>
                             SINON  <INSTRUCTION+>
                             FIN  SI                                                    ;
        #VALIDE(<INSTRUCTION>)
              =     (AND  VALIDE(<EXPRESSION>)   VALIDE(<EXPRESSION>')
                          VALIDE(<INSTRUCTION+>) VALIDE(<INSTRUCTION+>') )
        #MODELE(<INSTRUCTION>)
              =     SI_EGAL_ALORS_SINON
                          (TERME_ENTIER(<EXPRESSION>)
                          , TERME_ENTIER(<EXPRESSION>')
                          , MODELE(<INSTRUCTION+>)
                          , MODELE(<INSTRUCTION+>')     )
```

```
16 <INSTRUCTION>     =                                                                  ;
        #VALIDE(<INSTRUCTION>)
              =     T
        #MODELE(<INSTRUCTION>)
              =     VIDE
```

```
17 <VARIABLE>          =      %IDENT                                                    ;
        #VALIDE(<VARIABLE>)
              =      (NULL (COND
                     ((NULL (INTERSECTION (QUOTE (TEXT(%IDENT ))) VISIBLES(<VARIABLE>) ))
                      (PRINT (CONS  ENGLOBANT(<VARIABLE>)
                                  (QUOTE (: TEXT(%IDENT ) EST INACCESSIBLE)) )) )
                     ((NULL (MEMBER (TYPE (QUOTE TEXT(%IDENT ))  VISIBLES(<VARIABLE>) )
                                    (QUOTE   (TVAL  TVAR) ) ))
                      (PRINT (CONS  ENGLOBANT(<VARIABLE>)
                                  (QUOTE (: TEXT(%IDENT ) EST UNE VARIABLE
                                                INCORRECTE)) )) ) )))
        #TERME_VAR(<VARIABLE>): COMPILER(<VARIABLE>)
              =      ACCES  CALCULE
              (COND ((EQ (TYPE (QUOTE TEXT(%IDENT ))  VISIBLES(<VARIABLE>) )
                          (QUOTE  TVAL) )
                      REPERE
                          (ID_VAL 'TEXT(%IDENT )'
                          , APPEL_EN_COURS        )      )
                     (T
                      REFERENCE
                          (REF_REPERE
                                  (ID_VAR 'TEXT(%IDENT )'
                                  , APPEL_EN_COURS        ) )    ))

18 <VARIABLE>          =      %IDENT  [  <EXPRESSION>  ]                                ;
        #VALIDE(<VARIABLE>)
             = (AND (NULL (COND
                     ((NULL (INTERSECTION (QUOTE (TEXT(%IDENT ))) VISIBLES(<VARIABLE>) ))
                      (PRINT (CONS  ENGLOBANT(<VARIABLE>)
                                  (QUOTE (: TEXT(%IDENT ) EST INACCESSIBLE)) )) )
                     ((NULL (EQ (TYPE (QUOTE TEXT(%IDENT ))  VISIBLES(<VARIABLE>) )
                                  (QUOTE  TTAB) ))
                      (PRINT (CONS  ENGLOBANT(<VARIABLE>)
                                  (QUOTE (: TEXT(%IDENT ) N'EST PAS UN TABLEAU)) )) ) ))
                     VALIDE(<EXPRESSION>)     )
        #TERME_VAR(<VARIABLE>): COMPILER(<VARIABLE>)
              =      ELEMENT
                          (ID_TAB 'TEXT(%IDENT )'
                          , APPEL_EN_COURS
                          , TERME_ENTIER(<EXPRESSION>) )
```



```
19 <EXPRESSION>        =      <TERME>                                                   ;

20 <EXPRESSION>        =      <EXPRESSION>  <PLUS_MOINS>  <TERME>                       ;
        #VALIDE(<EXPRESSION>)
              =      (AND  VALIDE(<EXPRESSION>')  VALIDE(<TERME>) )
        #TERME_ENTIER(<EXPRESSION>)
              =      ACCES  CALCULE
              OPERATION(<PLUS_MOINS>)
                     (TERME_ENTIER(<EXPRESSION>')
                     , TERME_ENTIER(<TERME>)        )

21 <TERME>             =      <FACTEUR>                                                 ;

22 <TERME>             =      <TERME>  <MULT_DIV>  <FACTEUR>                            ;
        #VALIDE(<TERME>)
              =      (AND  VALIDE(<TERME>')  VALIDE(<FACTEUR>) )
        #TERME_ENTIER(<TERME>)
              =      ACCES  CALCULE
              OPERATION(<MULT_DIV>)
                     (TERME_ENTIER(<TERME>')
                     , TERME_ENTIER(<FACTEUR>) )
```

```
23 <FACTEUR>          =      <VARIABLE>                                    ;
        #TERME_ENTIER(<FACTEUR>)
              =      VALEUR (TERME_VAR(<VARIABLE>))


24 <FACTEUR>          =      %ENTIER                                       ;
        #VALIDE(<FACTEUR>)
              =      T
        #TERME_ENTIER(<FACTEUR>)
              =      ENTIER ' TEXT(%ENTIER ) '

25 <FACTEUR>          =      (  <EXPRESSION>  )                            ;


26 <FACTEUR>          =      (  -  <FACTEUR>  )                            ;
        #TERME_ENTIER(<FACTEUR>)
              =      OPPOSE (<TERME_ENTIER(<FACTEUR>'))


27 <PLUS_MOINS>       =      +                                             ;
        #OPERATION(<PLUS_MOINS>)
              =      PLUS

28 <PLUS_MOINS>       =      -                                             ;
        #OPERATION(<PLUS_MOINS>)
              =      MOINS
```



```
29 <MULT_DIV>         =      *                                             ;
        #OPERATION(<MULT_DIV>)
              =      MULT

30 <MULT_DIV>         =      %                                             ;
        #OPERATION(<MULT_DIV>)
              =      DIV
        $
        ;
        [
        ]
        FIN
        :=
        EGALE
        ALORS
        SINON
        )
        $
```

LE CONSTRUCTEUR A TRAITE 169 EQUATIONS SEMANTIQUES
DONT 67% D'IDENTITES ET 62% DE DEFAUTS

TAUX DE REMPLISSAGE DES TABLES

NO	XMAX	LMAX	%
1	30	150	20
2	27	100	27
3	279	1000	27
4	120	1000	12
5	13	100	13
6	172	500	34
7	169	600	28
8	55	150	36
9	374	1000	37
10	22	1016	2
11	12	50	24
12	4	100	4

STOP 0
!

F - ANNEXE

METALANGAGE ACCEPTE PAR &1.1

La grammaire ci-dessous décrit le métalangage utilisé pour les définitions d'attributs qui doivent être traduites en LISP par &1.1. Les notations utilisées sont les mêmes que précédemment, les formes LISP sont décrites dans l'annexe C et les non-terminaux <type source> et <operation source> sont introduits, dans la grammaire qui décrit &1.1 ; par &1.0.

```
<definition d'attribut>  =  "#" <attribut>  [<synchronisation>]
                            "=" <modele>  ;

<synchronisation>        =  ":" <attribut> {"," <attribut>} ;

<modele>                 =  <forme LISP>
                          | <operateur> "(" [<paramètres>]  ")"
                          | <constante source>   ;

<operateur>              =  <operation source>
                          | "opérateur calculé"  <forme LISP> ;

<paramètres>             =  (<constante source> | <attribut>)
                              { "," (<constante source> | <attribut>)}  ;

<constante source>       =  <type source> ("'" %id "'" | <forme LISP>) ;
```

Nous donnons maintenant la syntaxe de la déclaration d'un attribut associé à un type ou un profil du Type Source. Dans cette grammaire, le non-terminal <profil> est également introduit par &1.0 ; en outre, <sens> représente les types d'attributs acceptés par TABDEF et peut donc prendre

les valeurs suivantes :

S	pour synthétisé	G	pour global
H	pour hérité	M	pour mixte.

```
<déclaration d'attribut>  = <sens> "#" <attribut>  ":"  <type d'attribut> ;
<type d'attribut>         = <type source>
                          | <profil> ;
```

BIBLIOGRAPHIE

[AU77] Aho A.V., Ullmann J.D.,
Principles of compiler design,
Addison-Wesley, 1977.

[Bou77] Boullier P,
Le système SYNTAX : Manuel d'utilisation,
Groupe Langages et Traducteurs, IRIA, 1977.

[Bou80] Boullier P.,
Génération automatique d'analyseurs syntaxiques avec rattrapage d'erreurs,
Journées Francophones sur la production assistée de logiciel, Genève, 1980.

[BG77] Burstall R.M., Goguen J.A.,
Putting theories together to make specifications,
Proceedings of the 5th International Joint Conference on Artificial Intelligence, 1977.

[CN76] Courcelle B., Nivat M.,
Algebraic Families of Interpretation,
17th Symposium on Foundations of Computer Science, Houston, 1976.

[Der79] Deransart P.,
The language LISP does not exist !
SIGPLAN Notices, vol 4, n° 15, 1979.

[DM80] Deschamp Ph., Mazaud M.,
Manuel d'utilisation de l'analyseur Pascal de SYNTAX, à paraître.

[ES70] Earley, Sturgis,
A formalism for Translator Interactions,
Communications of the ACM 1970, vol 13, n° 10.

[Gau80] Gaudel M.C.,
Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation,
Thèse d'Etat, INPL, 1980.

[GDM78] Gaudel M.C., Deschamp Ph., Mazaud M.,
Semantics of Procedures as an Algebraic Abstract Data Type,
Rapport Laboria n° 334, IRIA, 1978.

[GH78] Guttag J.V., Horning J.J.,
The algebraic specification of abstract data type,
Acta Informatica 1978, vol 10, n° 1.

[GTW78] Goguen J.A., Thatcher J.W., Wagner E.G.,
Abstract data types as initial algebras and the correctness of data representations,
Current Trends in Programming Methodology 4,
Prentice Hall (édité par R. Yeh), 1978.

[Hoa69] Hoare C.A.R.,
An axiomatic basis of computer programming,
Communications of the ACM 1969, vol 12, n° 10.

[HKM77] Huet G., Kahn G., Maurice P.,
Environnement de programmation Pascal,
Rapport SESORI, IRIA, 1977.

[Lor74] Lorho B.,
De la définition à la traduction des langages de programmation,
Thèse d'Etat, Toulouse, 1974.

[Lor76] Lorho B.,
DELTA : Manuel d'utilisation,
Groupe Langages et Traducteurs, IRIA, 1976.

[Lor80] Lorho B.,
Manuel d'utilisation de SDELTA, à paraître.

[Lux75] Lux A.,
Etude d'un modèle abstrait pour une machine LISP et de son implémentation,
Thèse de 3ème cycle, USMG-INPG, Grenoble, 1975.

[Moi79] Moisant S.,
Contribution à la réalisation d'un éditeur syntaxique paramétré par le langage,
Rapport de 3ème année de l'ENSEEIHT, 1979.

[Mor73] Morris F.L.,
Advice on structuring compilers and proving them correct,
POPL, Boston, 1973.

[Pai80] Pair C.,
Types abstraits et sémantique algébrique des langages de programmation,
Rapport CRIN, Nancy, 1980.

[Ten76] Tennent R.D.,
The denotational semantics of programming languages,
Communications of the ACM, 1976, vol 19, n° 8.

Service Commun de la Documentation
INPL
Nancy-Brabois