Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Étude de la conception et de la réalisation d'un Environnement de Programmation Interactif COBOL à l'aide du Cornell Synthesizer Generator

Crismer, Paul-Georges; de Wergifosse, Jacques

Award date:
1988

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Jun. 2020

Etude de la conception et de la réalisation d'un Environnement de Programmation Interactif COBOL à l'aide du Cornell Synthesizer Generator.

Aufbau eines instrumentellen Compiler für COBOL mit Hilfe von Cornell Synthesizer Generatoren

Mémoire présenté par

Paul-Georges Crismer et Jacques de Wergifosse

en vue de l'obtention du titre de

Licencié et Maître en Informatique

Année académique 1987 - 1988

Promoteur : Professeur Baudouin Lecharlier

Résumé :

Chaque étape du développement d'un programme possède un outil : éditeur, compilateur et debugger, qui restent indépendants et s'ignorent. Un Environnement de Programmation Interactif regroupe des outils de développement de programmes en un seul, qui utilise la connaissance de la structure syntaxique d'un langage. Le Cornell Synthesizer Generator permet de produire de tels environnements à partir d'une spécification formelle de la syntaxe et de la sémantique d'un langage, exprimée sous forme de Grammaire Attribuée. A partir de la réalisation d'un environnement qui intègre un éditeur syntaxique et un compilateur incrémental pour le langage COBOL, ce mémoire étudie les implications provoquées par l'utilisation des grammaires attribuées et tente de dégager des éléments méthodologiques liés au Cornell Synthesizer Generator et à son langage de spécification.

Mots-clefs : Environnements de Programmation Interactifs, éditeurs syntaxiques, compilateurs incrémentaux, grammaires attribuées, Cornell Syntesizer Generator, langage de spécification, méthodologie, génie logiciel.

Abstract :

Each step of a program development has its own tool : editor, compiler and debugger ignore one another and remain unrelated. An Interactive Programming Environment integrates different program development tools into a single one, using the knowledge of a langage syntactic structure. The Cornell Synthesizer Generator generates such environments from the formal specification of a language syntax and semantics using attribute grammars. This thesis provides a study of the implications of specifying an environment integrating syntactic editor and incremental compiler for the COBOL language, with attribute grammars. An attempt is made to point out some aspects of a methodology, related to the use of the Cornell Synthesizer Generator and its specification language.

Keywords : Interactive Programming Environments, syntactic editors , incremental compilers, attribute grammars, Cornell Syntesizer Generator, specification language, methodology, software engineering.

Nous tenons tout spécialement à remercier

La société SIEMENS de Munich, Olivier Staes qui nous a proposé l'idée de ce mémoire, et les membres du service D ST SP 323 du centre de production de Perlach : ils nous offert les conditions idéales pour réaliser un bon stage, au point de vue technique, matériel et moral. Ils nous ont appris à nous dépasser, à travailler en équipe, à développer nos qualités humaines.

Le Professeur Baudouin Lecharlier sans qui ce sujet de mémoire ne nous aurait pas été confié. Il a su, malgré ses nombreuses occupations, rester disponible. Nous le remercions pour ses enseignements, pour ses remarques et ses conseils, pour nous avoir appris à porter un regard philosophe sur l'Informatique, et pour son enthousiasme communicatif.

Bruno Delcour qui nous a rendu un fier service pour installer le Cornell Synthesizer Generator sur ALMA. Nous le remercions pour sa compétence, pour le souci qu'il a eu du bon fonctionnement du CSG, pour sa patience à chaque fois que nous sommes venus frapper à sa porte.

Les professeurs de l'Institut d'Informatique qui nous ont prodigué leur science, nous ont ouvert l'esprit et qui ont oeuvré à donner toute sa signification au qualificatif "universitaire" de notre formation.

Tous ceux et celles qui ont participé de près ou de loin à la réalisation de ce mémoire, ainsi que nos parents et nos proches qui ont tout supporté : notre exubérance, nos découragements.

χαλεπα τα καλα
(Les belles choses sont difficiles)

Proverbe antique

Table des Matières

1. Introduction 1

2. Grammaires attribuées et notion d'incrémentalité 4

2.1. Introduction 4
2.2. Concepts importants 4
2.3. Concepts propres à l'évaluation des attributs 7
2.4. Evaluation proprement dite 7
2.5. Evaluation incrémentale 7
2.6. L'idée de "compilation incrémentale" avec les grammaires attribuées 8

3. De Theoria Programmorum 9

3.1. Syntaxe Abstraite - Syntaxe Concrète 9
3.1.1. Syntaxe Abstraite 9
3.1.2. Syntaxe Concrète 9
3.1.3. Exemple 9
3.2. Représentation Arborescente 10
3.2.1. Arbre concret - arbre abstrait 10
3.2.2. Exemple 11
3.3. Syntaxe Abstraite versus Syntaxe Concrète 12
3.3.1. Syntaxe abstraite 12
3.3.2. Syntaxe concrète 12
3.4. Construction d'un arbre abstrait 13
3.5. Passage d'un arbre abstrait à une représentation textuelle 14
3.6. Remarques 16

4. Modèle d'édition Syntaxique 17

4.1. Qu'est-ce qu'un Editeur Syntaxique ? 17
4.2. L'édition Textuelle 17
4.3. L'édition Structurelle 18
4.4. L'édition "Bimodale" texte - structure 19

5. Le Cornell Synthesizer Generator 20

5.1. Un précurseur : le Cornell Program Synthesizer 20
5.2. L'idée qui donna naissance au Cornell Synthesizer Generator 20
5.3. Le C.S.G. et son langage de Spécification 21
5.4. Le modèle d'édition du CSG 22
5.4.1. Modèle théorique 22
5.4.2. Modèle pratique 24
5.5. Structure d'une spécification d'éditeur syntaxique 26
5.5.1. Syntaxe abstraite du langage choisi 26
5.5.2. Attributs et équations sémantiques 26
5.5.3. Schémas de décompilation 26
5.5.4. Syntaxe d'entrée concrète 26

5.5.5. Déclaration des transformations 27
5.5.6. Conclusion 27
5.6. Structure d'un éditeur syntaxique généré par le CSG 27

6. La Spécification d'un Editeur Syntaxique avec le CSG 29

6.1. But de ce chapitre 29
6.2. Quelques aspects du langage SSL 29
6.2.1. Terminologie 29
6.2.2. Le langage fonctionnel du SSL : la term algebra 30
6.2.3. Conclusion 32
6.3. La syntaxe abstraite et les schémas de décompilation 33
6.3.1. Introduction 33
6.3.2. Définition d'un phylum dans le CSG 33
6.3.3. Completing Term et Placeholder Term d'un Phylum 34
6.3.4. Schémas de décompilation 37
6.3.5. Eléments méthodologiques 40
6.4. Les unités lexicales du langage 44
6.4.1. Introduction 44
6.4.2. Description de token en SSL 49
6.4.3. Lex : un générateur d'analyseurs lexicaux 51
6.4.4. Eléments méthodologiques 54
6.5. L'analyse syntaxique 58
6.5.1. Introduction 58
6.5.2. Rappels 58
6.5.3. Principe d'édition textuelle du CSG 64
6.5.4. Spécification d'une syntaxe d'entrée concrète en SSL 66
6.5.5. Lien entre syntaxe concrète SSL et YACC 71
6.5.6. Eléments méthodologiques 71
6.6. Déclarations de transformation 81
6.6.1. Spécification de transformations 81
6.6.2. Edition structurelle avec gabarits 82
6.6.3. Restructurations complexes 82
6.7. Vérification de la sémantique d'un programme 83
6.7.1. Vérification de la sémantique statique 83
6.7.2. Détection d'anomalies 85
6.7.3. Vérification de la correction 86
6.7.4. Conclusion - Performances 86

7. Conception du générateur de code C 88

7.1. Introduction 88
7.2. Conventions 88
7.2.1. Grammaire BNF 88
7.2.2. Règles de traduction 89
7.3. Grammaire BNF concrète du subset COBOL 91
7.4. Structure globale du programme C généré 99
7.4.1. Structure du code généré pour un programme COBOL 101
7.4.2. Fonctions permettant de réaliser l'appel d'un sous-programme COBOL 102
7.4.3. Conclusion 106
7.5. Représentation de la Data Division 107
7.5.1. Introduction 107
7.5.2. Principe de mémorisation 107
7.5.3. Description de l'environnement des données 116
7.5.4. Calcul d'adresse à l'exécution 122

7.5.5. Représentation des opérations sur les fichiers COBOL............126
7.6. Spécifications des fonctions C utilisées..130
7.6.1. Fonctions d'assignation d'une valeur à une donnée...............130
7.6.2. Fonctions de comparaison..133
7.6.3. Fonctions utilisées pour la réalisation des instructions ACCEPT et DISPLAY..136
7.6.4. Fonctions de gestion des fichiers COBOL 136
7.7. Représentation de la procedure division 139
7.7.1. Principe de représentation des sections et paragraphes 139
7.7.2. Fonctions de déclaration de variables C et de représentation d'instructions COBOL...143
7.7.3. Principe de numérotation des variables générées...................144
7.7.4. Représentation de la structure COBOL IF..........................144
7.7.6. Représentation des instructions ACCEPT et DISPLAY 155
7.7.7. Représentation des instructions CLOSE, OPEN, READ et WRITE...163
7.7.8. Représentation de l'instruction CALL..............................169
7.7.9. Représentation des instructions EXIT, EXIT PROGRAM et STOP RUN 169
7.7.10. Représentation de l'instruction PERFORM.......................170
7.7.11. Représentation de l'instruction MOVE et de la clause VALUE des données de la working-storage section..................................173
7.7.12. Représentation des conditions COBOL 178
7.8. Conclusion 183

8. Implémentation du générateur de code C..184

8.1. Introduction 184
8.2. Attributs et équations d'attributs..184
8.2.1. Formalisme du SSL..184
8.2.2. Comment passer de nos règles de traduction au formalisme SSL 185
8.2.3. Elements méthodologiques...186
8.3. Implémentation des environnements de compilation 187
8.3.1. Représentation d'un environnement...............................187
8.3.2. Construction d'un environnement...................................188
8.3.3. La consultation d'un environnement.................................190
8.4. Les attributs à la demande 191
8.5. Les attributs "remote" 192
8.6. Conclusion 193

9. Conclusion 194

Annexes[1]

Bibliographie

[1] la spécification SSL de l'environnement logiciel interactif COBOL que nous avons réalisé se trouve dans un appendice séparé.

Introduction

1. Introduction

Le cycle de développement d'un programme est le même depuis plus de vingt ans :

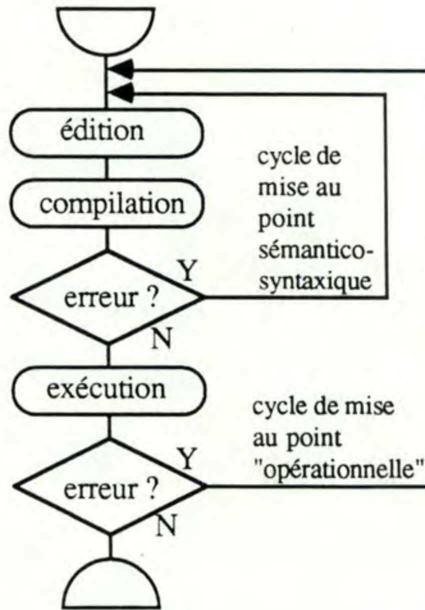


Figure 1.1 : cycle de développement traditionnel d'un programme

A l'heure des stations de travail où l'univers informatique ne jure que par l'interactivité et la convivialité; au moment où bon nombre de logiciels réunissant diverses fonctionnalités se présentent comme "intégrés", il est un domaine où l'interactivité n'est pas encore reine et où l'"intégration" brille par son absence : le développement de programmes ... En effet, en ce domaine, le paradigme de la "boîte à outils" règne encore.

Chaque étape du développement d'un programme possède son outil : éditeur, compilateur et debugger restent indépendants et s'ignorent. Pourtant un programme possède une structure syntaxique propre au langage dans lequel il est rédigé, et dont ces outils pourraient tirer parti. Un compilateur reconstitue à chaque fois la structure d'un programme avant de le traduire, afin de vérifier s'il est bien formé. Ne serait-il pas intéressant de concevoir des éditeurs "syntaxiques" qui, en plus de vérifier la correction sémantico-syntaxique des programmes, permettraient de développer ceux-ci sous une forme structurelle que compilateurs et debuggers pourraient exploiter ? Il serait alors possible d'intégrer ces outils, parce qu'ils partageraient une même représentation d'un programme.

Un Environnement de Programmation Interactif regroupe des outils de développement de programmes en un seul, qui utilise la connaissance de la structure d'un langage. Cette connaissance permet à un Environnement de Programmation Interactif de fournir des

possibilités de développement beaucoup plus efficaces par une coopération des outils qui y sont intégrés. Un Environnement de Programmation Interactif stimulerait la puissance créatrice de l'utilisateur en lui procurant une approche, structurelle et unifiée, de manipulation d'objets, et tendrait à réduire le cycle de développement de programmes à ceci :

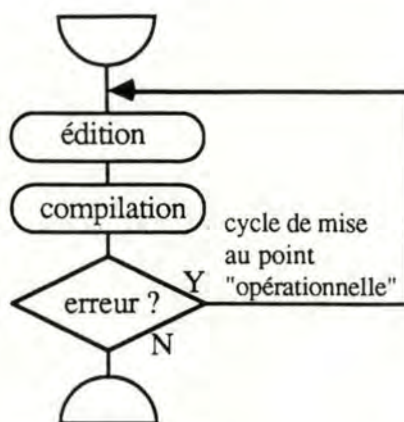


Figure 1.2 : cycle de développement d'un programme avec un Environnement de Programmation Interactif

On peut considérer que MENTOR est un des précurseurs des Environnements de Programmation Interactifs [Donzeau-Gouge & al. 1975] : c'est un outil général de manipulation d'arbres; un programme PASCAL est représenté par l'arbre de dérivation de la syntaxe abstraite de ce langage, et est édité en invoquant des primitives de manipulation d'arbres. D'autres projets, tels que GANDALF [MEDINA-MORA & FEILER 1981] et le Cornell Program Synthesizer [TEITELBAUM, REPS 1981] fournissent des environnements qui intègrent éditeur, compilateur ou interpréteur et debugger.

Reps et Teitelbaum se sont penchés sur le problème de la génération automatique d'environnements de programmation interactifs. Ils ont créé le Cornell Synthesizer Generator : à partir d'une spécification de la syntaxe, du format de représentation et de la sémantique statique d'un langage, il génère un éditeur syntaxique. Un éditeur syntaxique généré par le C.S.G. satisfait les contraintes syntaxiques et sémantiques d'un langage, lors du développement d'un programme, par un processus incrémental : c'est-à-dire que toute modification de programme provoque une analyse sémantico-syntaxique qui, pour des raisons d'efficacité, réutilise autant d'informations que possible.

La société SIEMENS A.G. de Munich nous a proposé d'utiliser le CSG afin de réaliser un Environnement de Programmation Interactif expérimental pour le langage COBOL, intégrant un éditeur syntaxique et un compilateur incrémental. Nous avons consacré à ce travail nos six mois de stage à Munich et quelques autres mois à l'institut d'informatique avant d'obtenir, enfin, un système opérationnel. Afin de nous concentrer sur les principes d'une telle réalisation, nous nous sommes défini un sous-ensemble du langage COBOL, que nous décrivons en annexe. De plus, pour en rester aux principes de la génération de code, et afin de ne pas nous lier à une machine particulière, nous avons décidé de produire un code objet en langage C.

Nous voulons présenter dans ce mémoire ce qui ressort de l'expérience que nous avons acquise par cette réalisation, mettre en exergue les différents aspects qui nous ont paru importants et qui ne sont, à notre connaissance, pas abordés dans la littérature. Nous nous sommes attachés à montrer comment il est possible de générer un Environnement de Programmation Interactif pour COBOL, à partir d'une spécification de sa syntaxe, de sa sémantique statique et d'une sémantique translationnelle; et à montrer quelles sont les difficultés qui peuvent survenir.

Comme le langage de spécification du CSG représente des grammaires attribuées, nous en présentons les principes généraux au chapitre 2.

La structure abstraite d'un programme décrit celui-ci en termes des concepts fondamentaux du langage utilisé, tandis qu'une structure concrète décrit un programme en termes des règles d'écriture du langage : c'est le sujet du chapitre 3.

Le chapitre 4 présente les principes généraux de l'édition syntaxique, tandis que le chapitre 5 décrit le CSG et son modèle d'édition syntaxique propre.

Le chapitre 6 étudie comment concevoir et réaliser un éditeur syntaxique avec le CSG.

La conception d'un générateur de code intégré à un éditeur syntaxique est décrite dans le chapitre 7, en se basant sur l'environnement de programmation que nous avons réalisé pour COBOL.

Le chapitre 8 explique comment passer à la réalisation d'un générateur de code avec le CSG.

Ce mémoire s'articule donc autour de deux grandes parties, consacrées, d'une part, aux concepts et méthodes mis en oeuvre pour spécifier un éditeur syntaxique, et, d'autre part, à la conception et à la réalisation d'un générateur de code. Enfin, la conclusion propose de nouveaux champs d'investigation à partir d'une réflexion sur les difficultés que nous avons rencontrées et le fruit de notre expérience.

2. Grammaires attribuées et notion d'incrémentalité

2.1. Introduction

Les Grammaires Attribuées sont un moyen idéal pour définir complètement un langage, y compris toutes les propriétés de celui-ci déterminables de manière statique.

Une Grammaire Attribuée est une généralisation d'une grammaire context-free, aux symboles de laquelle on associe un ensemble d'attributs. Chaque attribut décrit une propriété du langage.

Dans les sections qui suivent, nous nous attacherons à présenter au lecteur les concepts importants des grammaires attribuées, les concepts propres à l'évaluation d'arbres attribués par des ordinateurs, l'évaluation proprement dite et, enfin, la notion d'évaluation incrémentale.

2.2. Concepts importants

Une **Grammaire Attribuée** est une grammaire context-free étendue en attachant des attributs aux symboles de celle-ci.

L'ensemble des attributs attachés à un symbole se divise en deux sous-ensembles disjoints : celui des attributs synthétisés et celui des attributs hérités.

Chaque attribut possède une **équation sémantique** qui en définit la valeur.
On associe à une production X --> ß un ensemble d'équations sémantiques dont la forme est

$$\mathbf{a = f\,(c_1, ..., c_k)}$$

Deux cas sont possibles :
* a est un **attribut synthétisé** de x et $c_1, ..., c_k$ sont des attributs appartenant aux symboles du membre droit de la production , ou bien
* a est un **attribut hérité** d'un des symboles du membre droit de la production et $c_1, ..., c_k$ sont des attributs appartenant à X ou à n'importe quel symbole grammatical du membre droit de la production. [AHO & ULLMANN 1986, pp. 280].
On dira que les attributs $c_1, ..., c_k$ sont **arguments** de l'attribut a, ou encore que a **dépend** des $c_1, ..., c_k$; f est la **fonction sémantique** de a.

Un **graphe de dépendance** représente les dépendances fonctionnelles entre toutes les instances d'attribut d'une production ou d'un arbre. Chaque instance d'attribut est un sommet du graphe. Un arc partant d'un sommet b vers un sommet c signifie que l'instance

d'attribut b est utilisée pour déterminer la valeur de l'instance d'attribut c, ou encore que c **dépend** de b.

Une Grammaire Attribuée est **circulaire** si et seulement si son graphe de dépendance contient au moins un cycle.

Une Grammaire Attribuée est **bien-définie** si et seulement si le graphe de dépendance de tous les arbres de dérivation possibles est acyclique.

Une Grammaire Attribuée est **bien-formée** lorsque les symboles terminaux de la grammaire n'ont aucun attribut synthétisé, que le symbole racine n'a pas d'attribut hérité, et que chaque production inclut une équation sémantique pour tous les attributs synthétisés du non-terminal du membre gauche et pour tous les attributs hérités des symboles du membre droit.

Une Grammaire Attribuée est **ordonnée** si, pour tout symbole de la grammaire, on peut trouver un ordre partiel sur les attributs qui y sont associés tel que, dans n'importe quel contexte que se trouve le symbole, ses attributs peuvent être évalués dans un ordre qui inclut cet ordre partiel.

Exemple : La sémantique des nombres binaires, extrait de [REPS 1983].
v est un attribut synthétisé des non-terminaux Number, Integer et Bit qui décrit leur valeur.
s est un attribut hérité des non-terminaux Integer et Bit qui décrit le numéro d'ordre, à l'intérieur de Number, de leur bit le plus à droite, considérant qu'on les numérote de droite à gauche en commençant par zéro. Ainsi, si le bit numéro trois d'un entier est à un, sa valeur à l'intérieur de cet entier est 2^3.

Notation :

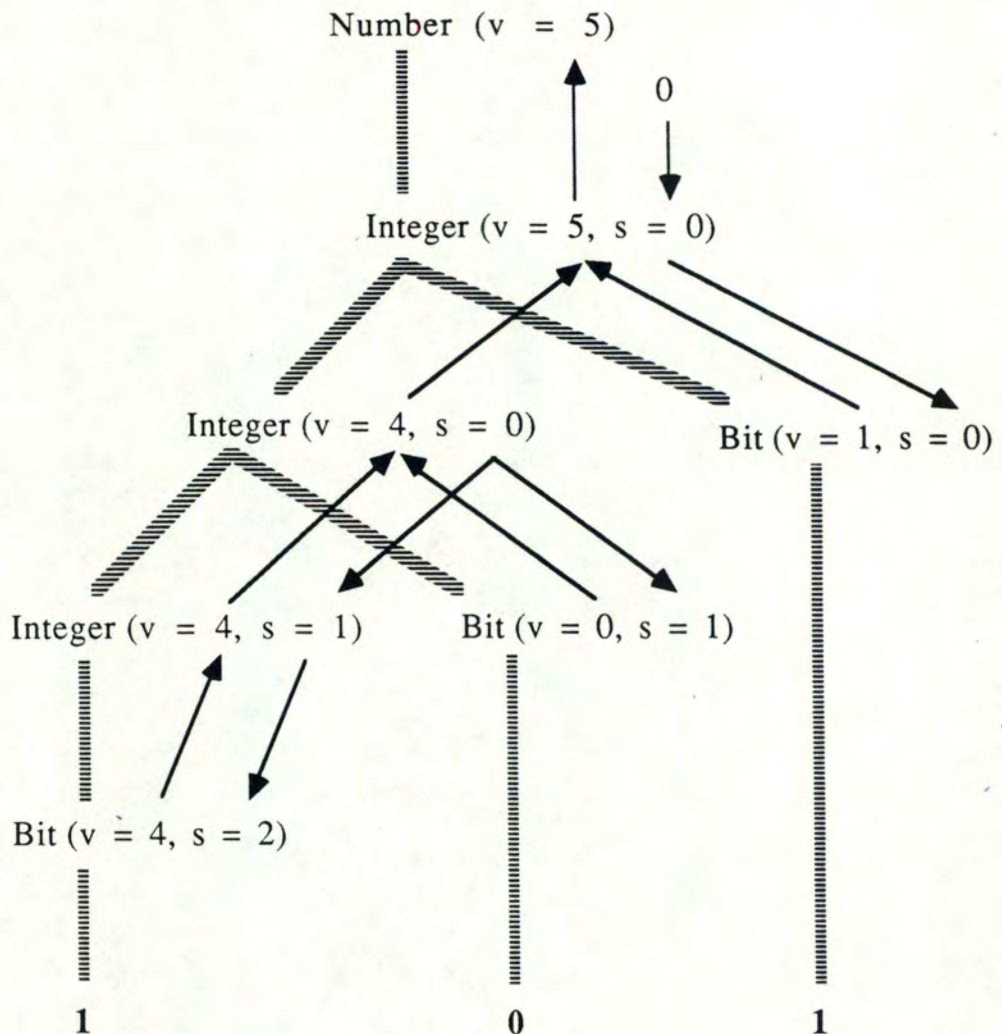
Bit.v est l'attribut v du non-terminal Bit.
Integer2 est la deuxième occurence du non-terminal Integer dans la production, en la considérant de gauche à droite ...
La notation v(Bit) a la même signification que Bit.v.

```
Number  --->    Integer
        Number.v = Integer.v
        Integer.s = 0

Integer --->    Integer  Bit
        Integer1.v = Integer2.v + Bit.v
        Integer2.s = Integer1.s + 1
        Bit.s = Integer1.s

Integer --->    Bit
        Integer.v = Bit.v
        Bit.s = Integer.s
```

Bit ---> **0**
Bit.v = 0

Bit ---> **1**
Bit.v = $2^{Bit.s}$

La figure suivante montre l'arbre attribué correspondant au nombre binaire 101.



Les Grammaires Attribuées sont utilisées comme langage de spécification pour des systèmes de génération de compilateurs, de générateurs et optimiseurs de code, et pour des générateurs d'environnements de programmation interactifs et d'éditeurs syntaxiques.

2.3. Concepts propres à l'évaluation des attributs

Un **arbre sémantique** est un arbre de dérivation avec, pour chaque instance d'attribut de cet arbre, l'assignation d'une valeur, ou d'un symbole spécial NULL qui indique que cette instance d'attribut n'a pas encore de valeur.

Un arbre sémantique est **entièrement attribué** si chacune de ses instances d'attribut sont **disponibles** c'est-à-dire que leur valeur n'est pas NULL.

Une instance d'attribut b est **cohérente** si :
1) les arguments de b sont *disponibles* et
2) la valeur de b est égale à sa fonction sémantique appliquée à ses arguments.
Dans tous les autres cas b est dit *incohérent*..

Un arbre sémantique ou un graphe de dépendance est cohérent si toutes ses instances d'attribut sont cohérentes.

Nous ne considérerons que des Grammaires Attribuées bien-définies (non-circulaires) et bien-formées.

2.4. Evaluation proprement dite

Pour analyser un programme conformément à la spécification de sa grammaire attribuée, il faut d'abord construire son arbre sémantique dont toutes les instances d'attribut sont initialisées à la valeur NULL. Ensuite, **l'évaluation des attributs** consiste à calculer autant d'instances d'attribut qu'il est possible conformément à leur équation sémantique respective.

L'ordre d'évaluation des instances d'attribut est arbitraire, mais une fonction sémantique ne peut être évaluée que lorsque tous ses attributs arguments sont *disponibles*. Une instance d'attribut non disponible est *prête pour évaluation* lorsque tous ses arguments sont disponibles.

Une stratégie d'évaluation consiste à respecter l'ordre partiel, sur les instances d'attribut, donné par leur graphe de dépendance [KNUTH 1968], [REPS 1983, pp. 27-28].

2.5. Evaluation incrémentale.

Dans le cadre d'applications qui construisent et modifient des arbres attribués, le mode d'évaluation décrit plus haut peut s'avérer fort peu efficace. En effet, il n'est pas toujours nécessaire de ré-évaluer toutes les instances d'attribut d'un arbre, qui vient d'être modifié, pour le garder cohérent.

Souvent c'est seulement un sous-ensemble des instances d'attribut qui sont devenues incohérentes. C'est pourquoi des algorithmes d'**évaluation incrémentale** ont été développés : ils déterminent l'ensemble des instances d'attributs affectés par une modification d'arbre et ré-évaluent uniquement ces instances-là.

Chaque modification d'un arbre attribué suscite l'application d'un algorithme d'évaluation incrémentale. Par ce moyen, un arbre attribué reste toujours cohérent.

2.6. L'idée de "compilation incrémentale" avec les grammaires attribuées.

L'idée d'une compilation incrémentale consiste à compiler chaque instruction du langage source pendant l'édition du programme. Puisqu'une compilation n'a de sens qu'avec un programme syntaxiquement correct et que la compilation agit durant l'édition du programme, l'idée d'associer étroitement un compilateur incrémental et un éditeur syntaxique surgit naturellement à l'esprit.

Une grammaire attribuée permet d'exprimer la sémantique translationnelle d'un langage au moyen, par exemple, d'attributs "code" associés à chaque non-terminal de la grammaire. Spécifier un langage et sa sémantique translationnelle[1] permet au Cornell Synthesizer Generator (CSG) de générer un éditeur syntaxique-compilateur incrémental. Editer un programme reviendrait à le compiler instantanément, modifier un programme reviendrait idéalement à ne recompiler que sa partie modifiée puisque le CSG utilise un évaluateur incrémental pour maintenir cohérent un arbre attribué.

Le CSG fournit un outil qui se prête idéalement à la compilation incrémentale car les attributs qui participent à la réalisation de la génération de code sont continuellement maintenus cohérents durant l'édition d'un programme. Une fois que l'édition d'un programme est terminée, celui-ci est prêt à être exécuté.

[1] La sémantique "translationnelle" d'un langage est définie en termes d'instructions d'un autre langage, elle revient à définir des règles de traduction pour un compilateur.

Première partie :

Conception et réalisation d'un Editeur Syntaxique avec le CSG.

Paul-G. Crismer

3. De Theoria Programmorum

3.1. Syntaxe Abstraite - Syntaxe Concrète

3.1.1. Syntaxe Abstraite

L'idée de la Syntaxe Abstraite est de décrire la structure d'un langage en n'y introduisant aucun détail arbitraire. Avec pour conséquence qu'une grammaire abstraite est adaptée pour décrire la sémantique d'un langage.

3.1.2. Syntaxe Concrète

La Syntaxe Concrète d'un langage est un ensemble de règles de grammaire dans lesquelles apparaissent les symboles terminaux du langage. La Syntaxe Concrète décrit ce qu'on *peut* écrire, sans vouloir en exprimer la sémantique. Définir ainsi un langage revient à :

* donner son alphabet, et
* décrire les suites de caractères qui appartiennent au langage
(par exemple avec une notation BNF)

De telles règles permettent de spécifier un analyseur syntaxique qui détermine si une suite de caractères appartient ou non au langage.

3.1.3. Exemple

L'exemple que nous utiliserons tout au long de ce chapitre est la définition d'expressions arithmétiques comprenant les opérations d'addition, soustraction, multiplication, division, les valeurs entières et une expression conditionnelle :

Syntaxe Concrète en notation[1] BNF :

expression	::=	expression '+' expression
	|	expression '-' expression
	|	expression '*' expression
	|	expression '/' expression
	|	**if** expression **then** expression **else** expression
	|	integer

Note : la valeur de l'expression "if $expression_1$ then $expression_2$ else $expression_3$" est $expression_2$ si $expression_1$ est positive, $expression_3$ sinon.

[1] **Terminologie :** les symboles terminaux sont en caractères gras; les symboles non-terminaux sont en caractères normaux.

integer ::= number
| integer number

number ::= **1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0**

Syntaxe abstraite[1] :

expression ::= Plus(expression, expression)
| Minus(expression, expression)
| Multiply(expression, expression)
| Divide(expression, expression)
| ConditionalExp(expression, expression, expression)
| Const(INT)

3.2. Représentation Arborescente

3.2.1. Arbre concret - arbre abstrait

Un **arbre de dérivation** décrit l'ordre dans lequel les règles de grammaire ont été appliquées pour dériver un programme.

Considérant un sous-arbre de dérivation, la racine de celui-ci est étiquetée par le nom du symbole non-terminal situé dans le membre gauche de la règle de grammaire qui s'applique; les feuilles (branches) du sous-arbre sont étiquetées par les noms des symboles terminaux (non-terminaux) situés dans le membre droit de cette règle.

On pourra donc refléter la structure syntaxique d'une phrase du langage par son arbre de dérivation. Il s'agira d'un arbre de dérivation abstrait ou concret selon que la dérivation est réalisée à partir des règles de syntaxe abstraite ou concrète. On parlera plus communément d'**arbre concret** ou d'**arbre abstrait**; un "arbre syntaxique" désignera aussi un arbre abstrait.

[1] Notation :
Chaque alternative d'une même règle de grammaire porte un nom que nous appellerons "**opérateur**"; celui-ci permet de dissocier les différentes variantes d'une même règle.

INT désigne le domaine des valeurs entières; c'est un symbole non-terminal.

3.2.2. Exemple

L'expression

if 3 - 9 **then** 3 * 5 **else** 3 * 9

a pour arbre de dérivation concret :

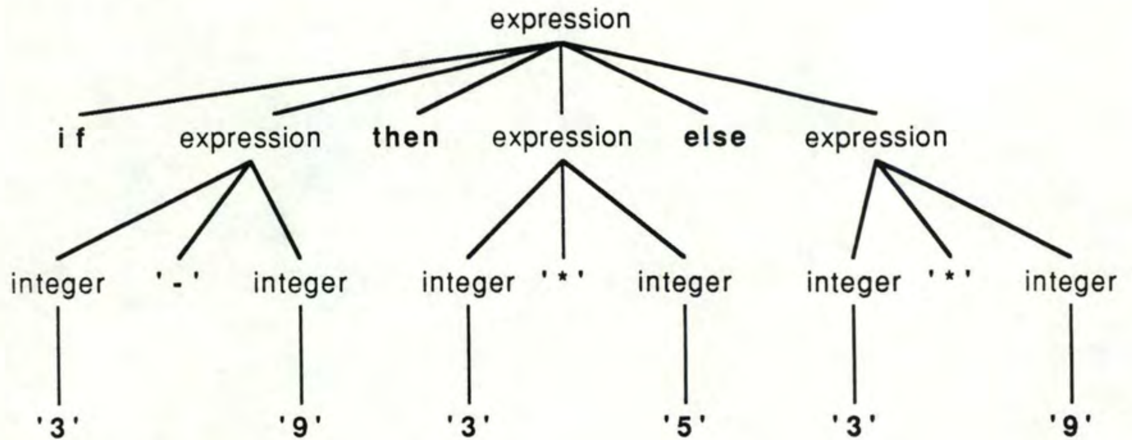


figure 1.1 : arbre de dérivation concret

ou pour arbre de dérivation abstrait :

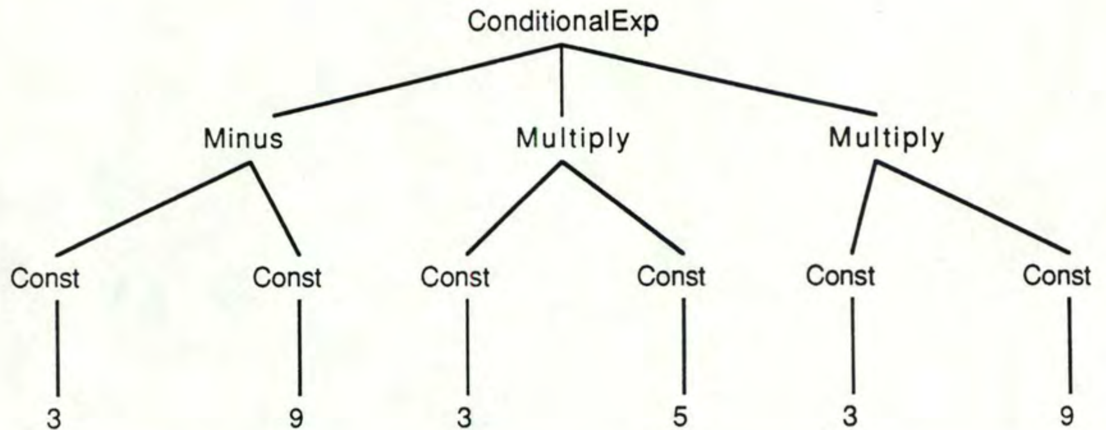


figure 1.1 : arbre de dérivation abstrait

Note : les chiffres 3, 5 et 9 désignent leur valeur et non les caractères '3', '5' et '9'. Pour un arbre abstrait on a substitué les noms de non-terminaux par les noms d'opérateurs qui s'appliquent .

3.3. Syntaxe Abstraite versus Syntaxe Concrète

3.3.1. Syntaxe abstraite

La syntaxe abstraite laisse tomber ce qui n'est pas essentiel dans un langage. Elle est adaptée pour en décrire la sémantique, mais est fort ardue à lire. Une règle de grammaire abstraite considère des n-uplets, objets composés, représentables sous forme arborescente.

Une syntaxe abstraite n'est pas ambiguë : représentant un arbre abstrait sous forme textuelle plutôt que graphique, les arbres
* Plus(Const(3), Plus(Const(4), Const(5))) ou
* Plus(Plus(Const(3), Const(4)), Const(5))
décrivent deux sémantiques différentes de 3 + 4 + 5.

3.3.2. Syntaxe concrète

La syntaxe concrète permet, quant à elle, de décrire un analyseur syntaxique et est agréable à lire. Une règle de grammaire concrète considère une concaténation de suites de caractères représentable sous forme de structure linéaire, un texte, ou sous la forme d'un arbre concret.

Une syntaxe concrète a le désavantage de pouvoir devenir ambiguë. Comment savoir si la suite de caractères 3 + 4 + 5 dérive d'un des deux arbres de dérivation suivants ?...

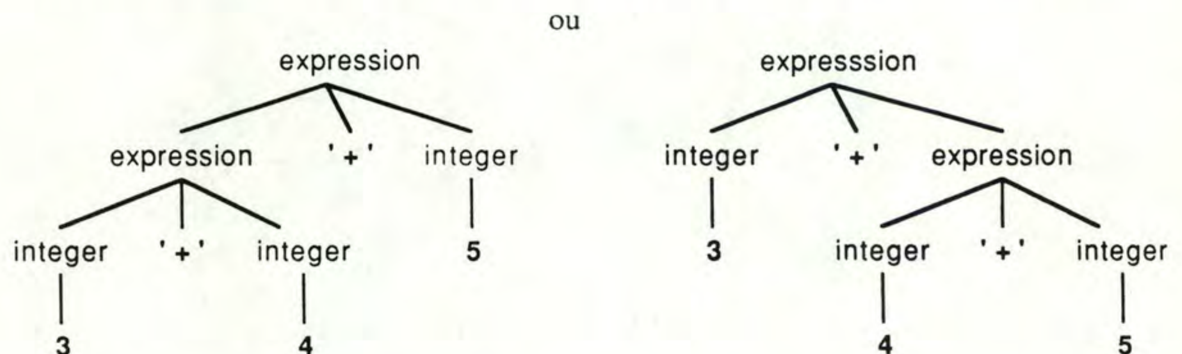


figure 1.3 : Arbres de dérivation de 3+4+5.

3.4. Construction d'un arbre abstrait

Une syntaxe abstraite et sa représentation arborescente sont utiles pour décrire la sémantique d'un langage. Encore faut-il pouvoir établir la correspondance entre une suite de caractères et sa sémantique dans un langage donné.

Les Grammaires Attribuées offrent un moyen d'exprimer la sémantique d'un langage de façon déclarative et dénotationnelle[1]. Associons à chaque règle de syntaxe concrète un attribut synthétisé Ω qui en représente l'arbre abstrait. En ajoutant à celui-ci un moyen d'éliminer les ambiguïtés[2], il est possible de construire l'arbre abstrait d'un programme à partir de son texte.

Peaufinons notre exemple : Pour éliminer toute ambiguïté, nous nous sommes défini une nouvelle syntaxe concrète dans laquelle toute expression doit être parenthésée, sauf les entiers et expressions conditionnelles. Ainsi, il n'y a aucune priorité d'opérateur implicite :

Jexpression$_1$ ::= (expression$_2$ '+' expression$_3$)
$\Omega(expression_1) = Plus(\Omega(expression_2), \Omega(expression_3))$

expression$_1$ ::= (expression$_2$ '-' expression$_3$)
$\Omega(expression_1) = Minus(\Omega(expression_2), \Omega(expression_3))$

expression$_1$ ::= (expression$_2$ '*' expression$_3$)
$\Omega(expression_1) = Multiply(\Omega(expression_2), \Omega(expression_3))$

expression$_1$ ::= (expression$_2$ '/' expression$_3$)
$\Omega(expression_1) = Divide(\Omega(expression_2), \Omega(expression_3))$

expression$_1$ ::= **if** expression$_2$ **then** expression$_3$ **else** expression$_4$
$\Omega(expression_1) = ConditionalExp(\Omega(expression_2),\Omega(expression_3), \Omega(expression_4))$

expression$_1$ ::= integer
$\Omega(expression_1)$ = Const(value[3](integer))

---

[1] Une sémantique dénotationnelle décrit le "tout" en fonction de la sémantique de chacune de ses parties. Stricto sensu, une grammaire attribuée n'est dénotationnelle que si l'on utilise **uniquement** des attributs synthétisés. En effet, avec une grammaire attribuée, un objet décrit dans une règle est rattaché aux composants de la règle; on essaye de tout calculer à partir de la règle. Une spécification d'attribut synthétisé est décrite à partir de l'objet seul, alors qu'une spécification d'attribut hérité est décrite par rapport à l'objet dans son contexte.

[2] soit en ré-écrivant des règles de syntaxe concrète non-ambiguës, ou en se servant d'attributs synthétisés ou hérités supplémentaires pour pouvoir tenir compte du contexte, ou en profitant de mécanismes propres à l'analyseur syntaxique qu'on utilise.

[3] value : integer --> INTEGER est une fonction qui, à une suite de caractères décrivant un entier, fait correspondre la valeur de cet entier.

L'arbre concret attribué de notre exemple devient :

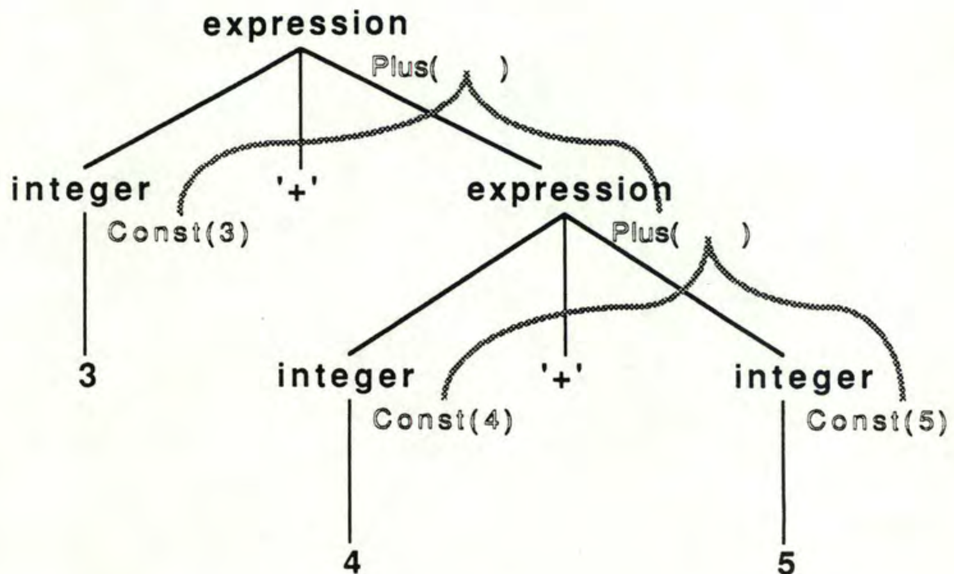


figure 1.4 : construction d'un arbre abstrait

L'attribut Ω de la racine de l'arbre concret a pour valeur :
Plus(const(3), Plus(Const(4), Const(5)).

3.5. Passage d'un arbre abstrait à une représentation textuelle

Nous pouvons obtenir, à partir d'un texte, l'arbre de dérivation abstrait correspondant en en "attribuant" l'arbre concret. Il est également intéressant de pouvoir visualiser un arbre abstrait par une représentation textuelle conforme à une syntaxe concrète donnée. Pour ce faire, on peut associer à chaque règle de grammaire abstraite un **schéma de décompilation** qui assurera le passage d'une structure arborescente à une représentation textuelle.

Définissons l'attribut synthétisé **v** comme une chaîne de caractères associée à toute règle de grammaire abstraite du langage. L'équation sémantique de **v** pour une règle décrit le schéma de décompilation de cette règle.

<u>Nous obtenons pour notre exemple</u> :

$$\text{expression}_1 \quad ::= \quad \text{Plus}(\text{expression}_2, \text{expression}_3)$$
$$v\ (\text{expression}_1) = v\ (\text{expression}_2)\$^{1}\text{"+"}\$\ v\ (\text{expression}_3)$$

$$\text{expression}_1 \quad ::= \quad \text{Minus}(\text{expression}_2, \text{expression}_3)$$
$$v\ (\text{expression}_1) = v\ (\text{expression}_2)\$\text{"-"}\$\ v\ (\text{expression}_3)$$

$$\text{expression}_1 \quad ::= \quad \text{Multiply}(\text{expression}_2, \text{expression}_3)$$
$$v\ (\text{expression}_1) = v\ (\text{expression}_2)\$\text{"*"}\$\ v\ (\text{expression}_3)$$

$$\text{expression}_1 \quad ::= \quad \text{Divide}(\text{expression}_2, \text{expression}_3)$$
$$v\ (\text{expression}_1) = v\ (\text{expression}_2)\$\text{"/"}\$\ v\ (\text{expression}_3)$$

$$\text{expression}_1 \quad ::= \quad \text{ConditionalExp}(\text{expression}_2, \text{expression}_3, \text{expression}_4)$$
$$v\ (\text{expression}_1) = \text{"if"}\$\ v\ (\text{expression}_2)\$\text{" then "}\$\ v\ (\text{expression}_3)\$\text{"else "}\$\ v\ (\text{expression}_4)$$

$$\text{expression}_1 \quad ::= \quad \text{Const(INT)}$$
$$v\ (\text{expression}_1) = \text{i_to_s}^{2}(\text{INT})$$

<u>L'arbre abstrait attribué de notre exemple devient</u> :

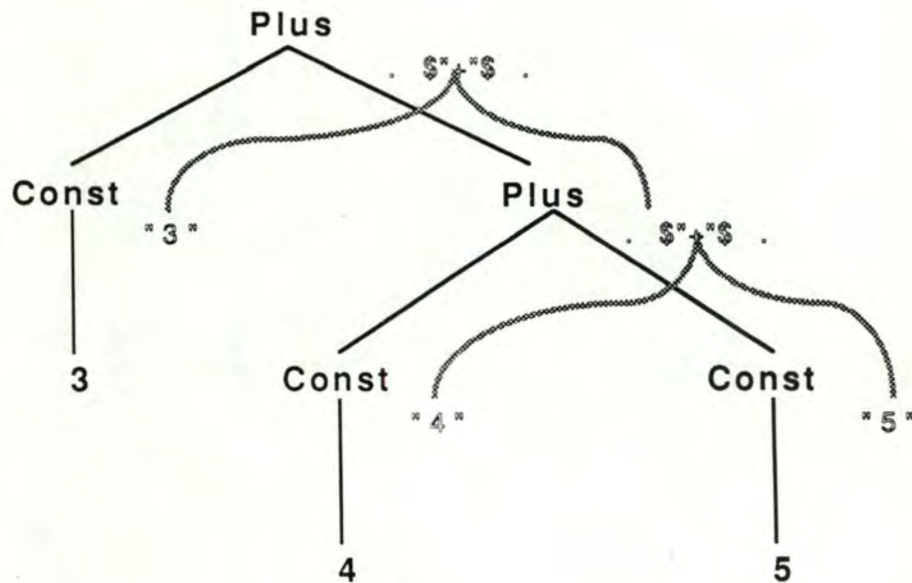


figure 1.5 : construction de la représentation visuelle d'un arbre abstrait

L'attribut **v** de la racine a pour valeur : "3 + 4 + 5"

---

[1] $: string x string --> string   fonction qui concatène deux chaînes de caractères.

[2] i_to_s : INT --> string   fonction qui convertit une valeur entière en sa représentation textuelle.

3.6. Remarques

Plusieurs langages à syntaxes concrètes différentes, mais qui partagent les mêmes concepts peuvent exprimer leur sémantique via une seule et même syntaxe abstraite. Par exemple : les dialectes d'un même langage, les langages d'interrogation de bases de données basés sur l'algèbre relationnelle, etc ...

La syntaxe abstraite d'un langage n'est pas unique mais il y a de fortes chances pour que les syntaxes abstraites d'un même langage spécifiées par des personnes différentes aient un squelette commun.

C'est à dessein que nous avons parlé de la syntaxe abstraite d'**un** langage et non **des** langages : avec des concepts et des sémantiques aussi différents, PASCAL et PROLOG, par exemple, ne peuvent partager une même grammaire abstraite.

4. Modèle d'édition Syntaxique

4.1. Qu'est-ce qu'un Editeur Syntaxique ?

Un éditeur syntaxique est dédié à un langage particulier. Il permet d'écrire un programme syntaxiquement correct en avertissant l'utilisateur des fautes que celui-ci vient de commettre, voire en les corrigeant automatiquement. Il y a des éditeurs purement syntaxiques qui n'ont aucune connaissance de la sémantique du langage, d'autres procèdent à des vérifications de sémantique statique.

Dans le cadre des Environnements de Programmation Intégrés (combinant éditeur syntaxique et compilateur incrémental, voire également debugger et interpréteur) un éditeur syntaxique doit posséder les caractéristiques suivantes :

* réaction immédiate en ce qui concerne les erreurs de syntaxe et de sémantique statique **pendant que** l'utilisateur est en train d'éditer son programme,

* disponibilité de gabarits (templates) structurés, à partir de commandes d'édition, pour construire un programme correct,

* possibilité d'introduire du texte, à tout moment, plutôt que d'utiliser des gabarits,

* des menus permettent de sélectionner la plupart des commandes.

La connaissance de la grammaire d'un langage permet à un éditeur syntaxique de renforcer la correction (syntaxique) d'un programme, à tous les moments de son développement, en n'en permettant que les modifications "légitimes".

4.2. L'édition Textuelle

L'édition en mode textuel consiste à considérer le programme comme un texte ordinaire. Les éditeurs syntaxiques basés sur ce modèle comme POE [FISCHER & al. 1984], COPE [ARCHER & CONWAY 1981] et MAGPIE [DELISLE & al. 1984] utilisent un analyseur syntaxique "intelligent", capable d'insérer des mots-clefs manquants là où c'est nécessaire afin de rendre syntaxiquement correct ce que l'utilisateur vient d'introduire. Lorsqu'une erreur de sémantique statique est détectée, la portion de texte incriminée est affichée en surintensité.

Un programme pourra être parcouru comme avec un éditeur de texte conventionnel : ligne par ligne, caractère par caractère; la manipulation du programme se fait par les commandes habituelles de recherche, d'insertion, de suppression, de substitution et de copie de texte.

4.3. L'édition Structurelle

L'édition syntaxique en mode structurel est une idée "neuve". On considère avant tout la structure syntaxique du programme sous forme d'arbre de dérivation (arbre concret comme dans PECAN [REISS 1984] ou arbre abstrait comme dans PASES [SHAPIRO & al. 1980]).

L'édition textuelle, avec un écran rectangulaire, nous donne une impression semblable à celle que nous ressentons devant une feuille de papier... Mais l'édition structurelle d'un langage syntaxiquement riche a quelque chose de déroutant. En effet, à quoi peut donc ressembler le mouvement du curseur sur une structure arborescente plutôt que sur un texte ? Il y a quelque chose d'inhabituel à éditer un programme lorsque les commandes d'édition étendent leur action au sous-arbre du noeud sur lequel le curseur clignote plutôt qu'à un caractère sur une ligne.

Le moyen le plus naturel de visualiser un texte de programme sur écran est de s'imaginer que l'arbre syntaxique est couché sur le côté, avec la racine dans le coin supérieur gauche de l'écran et ses fils décalés d'une indentation vers la droite avec assez d'espace entre ceux-ci pour permettre à leurs propres fils d'être affichés, et ainsi de suite...

Exemple :

```
père
        fils1
                fils1.1
                ...
                fils1.n
        fils2
        ...
        filsn
```

Les opérations d'édition de base sont identiques pour le texte et la structure : remplacer, insérer, copier, supprimer. La différence réside dans les arguments de ces opérations : les arguments sont des structures arborescentes, alors que dans l'édition textuelle ce sont un caractère, un mot, une ligne... D'autre part, le parcours du texte d'un programme revient à "escalader" son arbre de dérivation : remonter au noeud parent, descendre au noeud fils, passer au noeud frère, etc...

Le mode d'édition structurel *pur* consiste à introduire un programme à l'aide de **gabarits** qui donnent les mots-clefs nécessaires, la ponctuation et des "emplacements" (placehoders) pour les non-terminaux et terminaux à définir par l'utilisateur. Une fois positionné à une entité non-terminale, un nouveau jeu de gabarits est disponible pour procéder à son expansion. Positionné à une entité terminale, il faut en introduire le symbole textuellement.

Exemple : en PASCAL un gabarit pour l'instruction "if" pourrait être

if <condition> **then** <statement> **else** <statement> ;

où <condition> et <statement> sont des "emplacements".

Editer un programme avec ce type d'éditeur est donc une succession de commandes d'insertion de gabarits pour l'expansion d'entités non-terminales, et d'introductions textuelles de symboles terminaux.

4.4. L'édition "Bimodale" texte - structure

Le besoin s'est fait sentir de dépasser la rigidité perçue avec un éditeur structurel à gabarits. En effet, lorsque beaucoup de corrections sont à faire dans un programme, l'utilisateur est susceptible de vouloir travailler avec un objet textuel pour réaliser ses corrections et ne pas vouloir s'inquiéter de la structure arborescente de son programme.

Un éditeur "bimodal" permet à tout moment l'expansion des non-terminaux par gabarits ou textuellement. Dans ce dernier cas le texte introduit est analysé, un arbre est construit et inséré dans la représentation interne du programme. Un analyseur syntaxique **incrémental**[1] permet de ne pas toujours reconstruire tout l'arbre syntaxique du programme, mais une partie seulement.

Etant donné cette bimodalité, l'utilisateur peut manipuler des sous-arbres de programme ou des blocs de texte... et ensuite ré-analyser ces blocs pour les insérer dans la représentation interne du programme [SYNED]. On peut mouvoir le curseur structurellement le long des noeuds de l'arbre ou textuellement ligne par ligne, caractère par caractère.

Chaque façon d'introduire un programme a des avantages[2] aux yeux des utilisateurs :

* les gabarits et les menus sont fort utiles si l'on code un programme dans un langage qu'on utilise rarement. De même, cette approche permet d'éviter la dactylographie intensive que provoquent des langages verbeux.

* l'introduction textuelle est utile lorsqu'un utilisateur sait exactement ce qu'il veut faire et comment exprimer sa pensée correctement.

Le moment où l'utilisateur passera d'un mode à l'autre dépendra du langage utilisé, de l'utilisateur lui-même, et de son niveau d'expérience.

---

[1] Le problème est traité dans [GHEZZI & MANDRIOLI 1979].

[2] Le lecteur peut trouver des comparaisons dans [SEBESTA 1985], et dans [MORRIS-SCHWARTZ 1981].

5. Le Cornell Synthesizer Generator

5.1. Un précurseur : le Cornell Program Synthesizer

Le système de programmation interactif le plus connu est le Cornell Program Synthesizer (CPS). CPS est le premier environnement de programmation complet pour des langages structurés, comprenant un ensemble d'outils pour créer, éditer, exécuter et mettre au point ("debug") des programmes et dont la pierre angulaire est son éditeur syntaxique. A l'origine, il fut dédié au langage PL/CS. Les différentes facilités du CPS sont unies par une base commune : la grammaire du langage de programmation.

Voici comment Tom Reps et Tim Teitelbaum, auteurs du CPS, introduisent celui-ci : "Les programmes ne sont pas des textes; ils sont des compositions hiérarchiques de structures 'computationnelles' et devraient être édités, exécutés et mis au point dans un environnement qui, avec cohérence, reconnaît et renforce ce point de vue. Avec le CPS, le développement d'un programme, à toutes ses étapes, doit être vu dans une perspective structurelle" [TEITELBAUM & REPS 1981].

5.2. L'idée qui donna naissance au Cornell Synthesizer Generator

Avec le CPS, Reps et Teitelbaum ont démontré la puissance d'un éditeur syntaxique plein-écran pour langages structurés, spécialement lorsqu'il est combiné avec compilateur incrémental, interpréteur et debugger. Le succès de cet outil écrit dans un langage procédural donna à ses auteurs l'idée de créer le Cornell Synthesizer Generator (CSG) pour construire de tels environnements **automatiquement** à partir d'une spécification, et pour y intégrer des outils supplémentaires d'analyse de programmes et de compilation. Pour ce faire, il leur fallait un moyen pour incorporer dans une spécification la connaissance des relations sémantiques d'un langage : le formalisme des grammaires attribuées [DEMERS & al. 1981].

Ainsi naquit le CSG, un outil pour spécifier comment des objets structurés peuvent être manipulés en présence de relations dépendantes d'un contexte. Le concepteur d'un éditeur décrit une spécification grâce à une grammaire attribuée; cette spécification comprend des règles exprimant la syntaxe abstraite, les attributs et leurs équations sémantiques, les formats de représentation visuelle et une syntaxe d'entrée concrète[1]. A partir de cette

[1] cfr infra, le paragraphe consacré à la "structure d'une spécification d'éditeur syntaxique".

spécification, le CSG génère un éditeur plein-écran bimodal (textuel - structurel) qui manipule des objets conformément à ces règles [REPS & TEITELBAUM 1984].

Un objet en cours d'édition est représenté comme un arbre de dérivation abstrait, attribué, dont les valeurs d'attribut respectent à tout moment leur équation sémantique : un arbre attribué *cohérent*.. Modifier un programme revient à restructurer son arbre par des opérations d'élagage, de greffage et de dérivation. Cette restructuration affecte directement les valeurs d'attribut attachées à l'endroit de la modification : certains attributs peuvent ne plus avoir de valeur cohérente. Une analyse **incrémentale** - se limitant strictement à la partie de l'arbre de dérivation devenue non-cohérente - est réalisée en évaluant, à travers l'arbre, une nouvelle valeur pour chaque attribut touché par la modification.

5.3. Le C.S.G. et son langage de Spécification

La spécification d'un éditeur syntaxique doit être écrite dans le **S**ynthesizer **S**pecification **L**anguage (SSL). Il s'agit d'un formalisme qui permet de décrire des grammaires attribuées; les équations sémantiques d'attributs sont exprimées dans une "**term algebra**", un langage fonctionnel qui utilise un mécanisme de "pattern-matching". Le SSL possède également des caractéristiques spécifiques au domaine d'application de l'édition syntaxique.

En tant qu'outil disponible sous Berkeley UNIX, le CSG s'articule autour de deux composants majeurs :

* un compilateur SSL qui traduit une spécification SSL en un certain nombre de tables, et,

* un noyau d'édition qui exécute les opérations appropriées sur l'arbre de dérivation courant en réponse aux sollicitations du clavier ou d'une souris (pointing device).

5.4. Le modèle d'édition du CSG

5.4.1. Modèle théorique

5.4.1.1. Rappel

Une **grammaire attribuée** est une grammaire context-free étendue en associant des attributs aux symboles de la grammaire. Toute production d'une grammaire attribuée dispose d'un ensemble d'équations sémantiques pour ses attributs.

Un **arbre sémantique** est un arbre de dérivation avec, pour chaque instrance d'attribut de l'arbre, l'assignation d'une valeur ou d'un symbole spécial NULL dont la valeur se situe en dehors du domaine de tout attribut (afin d'exprimer qu'un attribut n'a pas encore été évalué par l'ordinateur). Un arbre sémantique est **entièrement attribué** si toutes les valeurs de ses attributs sont différentes de NULL.

Une valeur d'**attribut synthétisé** d'un objet est calculée à partir des valeurs d'attribut des composants de cet objet[1]. Une valeur **d'attribut hérité** est calculée à partir de valeurs d'attribut d'objets de son contexte[2].

5.4.1.2. Un programme est un arbre attribué cohérent

Un programe est représenté comme un arbre abstrait attribué cohérent [REPS & al. 1983] (c'est-à-dire que toutes les valeurs de ses instances d'attribut respectent leur équation sémantique). Toute opération d'édition est une manipulation de cet arbre. Pour réaliser pratiquement le mécanisme d'attribution, il faudrait évaluer tous les attributs à chaque fois que l'arbre est modifié... Cela pourrait s'avérer fort peu efficace, c'est pourquoi des algorithmes d'évaluation incrémentale furent développés : ils déterminent l'ensemble des attributs affectés par une modification et ré-évaluent uniquement ceux-là.

La création d'un programme peut être considérée comme la croissance d'un arbre sémantique, ou encore comme le développement d'un arbre de dérivation **partiel** , c'est-à-dire qu'il contient des symboles non-terminaux non-développés. Mais il est impossible de calculer des valeurs d'attribut synthétisé pour un symbole non-terminal non-développé ... Il est donc impossible de satisfaire notre désir de maintenir des valeurs cohérentes pour chaque attribut de l'arbre : lorsqu'on décrit une grammaire abstraite destinée à un éditeur

---

[1] cfr note suivante

[2] Soit $p \dashrightarrow p_1 \ldots p_m$ une production dont l'équation sémantique d'un attribut est $a = f(c1, \ldots ck)$, on a

* a est un attribut **synthétisé** de p et les c1, ... ck sont des attributs appartenant aux symboles du membre droit de la production [les p1, ... pm], ou bien

* a est un attribut **hérité** d'un des symboles du membre droit de la production et les c1, ... ck sont des attributs appartenant à p ou à n'importe quel symbole du membre droit de la production.

syntaxique, il faut donner une sémantique non seulement aux programmes bien formés mais aussi aux programmes **partiellement développés**.

Voilà pourquoi le concepteur d'un éditeur syntaxique doit compléter la grammaire abstraite de son langage en y ajoutant, pour chaque non-terminal, une production spéciale qui indique que celui-ci est non-développé : nous l'appellerons **production complétive**. Donc, tout non-terminal X aura une production complétive X --> ⊥.

"⊥" signifie "non-développé"; les équations sémantiques de cette production complétive permettent ainsi de définir des valeurs pour les attributs synthétisés de toute occurrence de symbole non-terminal non-développé.

Par convention, nous dirons qu'une occurrence d'un non-terminal non-développé a dérivé "⊥". Tous les arbres de dérivation, *partiels* du point de vue de l'utilisateur, sont considérés comme *complets* du point de vue du système. Donc, puisqu'un programme peut être *complètement dérivé* grâce à ces production complétives, il peut être *entièrement attribué* .

5.4.1.3. Que représente l'édition d'un programme ?

En termes d'opérations de manipulation d'arbre :
modifier un programme revient à restructurer son arbre de dérivation par des opérations d'élagage et de greffage de sous-arbre.

Soient **T** un arbre sémantique,
U un sous-arbre de **T**, de racine **r** et qui dérive un non-terminal X,
U' un arbre sémantique de racine **r'**, dérivant également un non-terminal X :

élaguer le sous-arbre **U** de l'arbre **T** consiste à enlever ce sous-arbre enraciné en **r**,

greffer **U'** sur **T** à la feuille **r** consiste à assigner les valeurs des attributs synthétisés de **r** aux instances des attributs synthétisés de **r'**, et à remplacer ensuite **r** par **U'** dans **T**,

remplacer un sous-arbre **U** par un autre **U'** consiste à élaguer **U** et puis greffer **U'** à la place.

Un évaluateur incrémental d'attributs rendra l'arbre cohérent et entièrement attribué après chaque opération de remplacement de sous-arbre.

Le système doit tenir compte d'**arbres autonomes** dérivés de n'importe quel non-terminal de la grammaire et pas seulement de ceux qui sont dérivés du symbole racine, parce que les modifications peuvent avoir lieu à n'importe quel endroit dans un programme. En d'autres termes, un sous-arbre dérivant X, une fois enlevé, devient un arbre autonome dont la racine dérive X. On peut le mémoriser afin que, plus tard, il puisse être inséré ailleurs dans le programme.

Chaque **insertion** à un non-terminal X non-développé est considérée comme le remplacement de la production complétive de X par un arbre autonome U' dont la racine dérive X. Chaque **suppression** est considérée comme le remplacement du sous-arbre U dérivant X par une instance de la production complétive de X.

Une session d'édition est, en résumé, une succession d'opérations de remplacement de sous-arbre et de mouvements de sélection, en partant initialement d'un arbre sémantique complet et totalement attribué

avec la sélection positionnée à ROOT.

5.4.2. Modèle pratique

Un objet en cours d'édition est mémorisé dans un **buffer**. Il peut y avoir plusieurs buffers qui ont chacun un nom. On édite un objet à travers une **fenêtre** qui en donne une représentation visuelle. Chaque fenêtre présente le contenu d'un seul buffer mais plusieurs fenêtres peuvent être ouvertes sur un même buffer.

L'objet contenu dans un buffer est un arbre de dérivation soumis aux règles de la syntaxe abstraite incorporée à l'éditeur. Cependant, on ne peut voir qu'une section rectangulaire de la représentation visuelle d'un arbre, mais on peut déplacer la fenêtre afin d'y voir défiler l'objet dans sa totalité.

Dans un éditeur de texte, un curseur désigne le caractère sur lequel va porter l'opération d'édition à venir. Dans notre modèle la **sélection** est une sorte de "curseur structurel" qui désigne le sous-arbre sur lequel portera l'opération d'édition suivante. On peut changer de sélection par des opérations de parcours d'arbre ou à l'aide du curseur : en sélectionnant un caractère à l'écran, on pose la sélection sur le plus petit sous-arbre dont la représentation visuelle comprend ce caractère. Il est possible d'étendre la sélection en pointant sur un deuxième caractère : la sélection se pose alors sur le plus petit sous-arbre dont la représentation visuelle comprend ces deux caractères. Puisque la sélection est un sous-arbre et non une chaîne de caractères, elle désigne toujours une portion de programme syntaxiquement bien formée.

La figure 3.1 qui suit représente un modèle d'écran d'éditeur syntaxique : il est divisé en quatre parties : une barre de titre, une ligne de commande, une partie "objets" et une partie "aide". La barre de titre donne des informations sur l'état de la fenêtre visualisée et du buffer associé; la ligne de commande montre l'écho des commandes introduites et les messages d'erreur; la partie "objets" contient les fenêtres ouvertes sur les objets en cours d'édition (maximum deux fenêtres pour un terminal conventionnel); la partie "aide" affiche

d'une part le nom de l'objet contenu dans la sélection et, d'autre part, les différents gabarits disponibles et les différentes commandes permises pour restructurer la sélection.

```
buffer : exemple.pas
command : writeln
program <name> ;
var <name> : <type denoter> ;
begin
    writeln("ceci est un exemple") ;
    a {<--- undeclared variable } := 3 ;
    <statement>
end.

positioned at : statement          insert_before   insert_after   if
while  forto  fordown  assign  write  writeln  case  repeat
```

figure 3.1 : écran type d'un éditeur syntaxique conforme au modèle du CSG

Chaque opération d'édition remplace le sous-arbre de la sélection par un autre. Ces opérations peuvent être textuelles (comme avec un éditeur de texte habituel) ou structurelles (parcours d'arbre, élagage, greffage).

Une fois le processus d'édition terminé, le contenu des buffers est écrit dans des fichiers selon un format choisi : textuel ou structurel. Le format structurel est illisible pour le commun des mortels mais peut être relu par l'éditeur sans problèmes. Le format textuel pourra servir pour imprimer des listings mais ne pourra être relu par l'éditeur que si celui-ci dispose d'une syntaxe d'entrée concrète complète dans sa spécification ...

5.5. Structure d'une spécification d'éditeur syntaxique

Les différents éléments d'une spécification d'éditeur syntaxique sont étroitement liés aux fonctions remplies par un tel éditeur ...

5.5.1. Syntaxe abstraite du langage choisi

La syntaxe abstraite est le noyau de toute spécification puisque la structure de tout programme, interne à l'éditeur, est son arbre de dérivation abstrait.

5.5.2. Attributs et équations sémantiques

C'est grâce au maintien d'un arbre attribué cohérent, par un évaluateur incrémental d'attributs, qu'une réaction immédiate peut être donnée à l'utilisateur en réponse à la présence d'erreurs. Ce sont les attributs qui permettent d'exprimer les relations sémantiques entre les différents composants d'un langage. Il faudra déclarer les attributs de chaque règle de grammaire et en décrire les équations sémantiques dans un langage fonctionnel, la "term algebra", qui dispose d'un mécanisme de pattern-matching.

5.5.3. Schémas de décompilation

Ce sont les schémas de décompilation qui permettent d'avoir une représentation visuelle de l'arbre abstrait d'un programme. Des attributs peuvent entrer dans leur composition afin de pouvoir annoter l'écran avec des messages de diagnostic ... ou provoquer d'autres effets encore.

5.5.4. Syntaxe d'entrée concrète

La syntaxe d'entrée concrète est nécessaire si l'on veut pouvoir introduire un programme textuellement. Que se passe-t-il dans ce cas ? Un analyseur syntaxique utilise les services d'un analyseur lexical et construit un arbre de dérivation concret attribué, dont un des attributs est l'arbre abstrait correspondant. La spécification comportera donc :

* pour l'analyseur lexical, la définition de lexèmes avec expressions régulières,

* pour l'analyseur syntaxique, la description d'une syntaxe d'entrée concrète,

* pour la construction de l'arbre abstrait, la déclaration des attributs de chaque règle de grammaire concrète avec leurs équations sémantiques respectives, ainsi que des "déclarations d'entrée" qui permettent d'établir la correspondance entre l'arbre abstrait d'un objet et un attribut de l'arbre concret de cet objet.

5.5.5. Déclaration des transformations

L'édition structurelle par gabarits est réalisée avec ces transformations : elles désignent les commandes qui fourniront des gabarits pour chaque symbole non-terminal non-développé. Une transformation spécifie comment restructurer un objet situé à l'emplacement de la sélection lorsque cet objet coïncide avec un modèle structurel donné.

5.5.6. Conclusion

Le lecteur aura remarqué que ces spécifications décriront un éditeur bimodal textuel - structurel. Si l'on désire un éditeur syntaxique uniquement textuel, il ne faudra pas spécifier de transformations; si l'on ne désire qu'un éditeur syntaxique purement structurel, la syntaxe d'entrée concrète et ses déclarations associées ne doivent pas être spécifiées.

5.6. Structure d'un éditeur syntaxique généré par le CSG

Tout éditeur syntaxique généré par le CSG comporte deux modules standards : un noyau d'édition et un évaluateur incrémental d'attributs. Le noyau d'édition est "paramétré" par les schémas de décompilation, l'analyseur lexical, l'analyseur syntaxique et le jeu de transformations, qui sont décrits dans une spécification. L'évaluateur incrémental d'attributs comporte un interpréteur de pseudo-code. Ce pseudo-code est généré à partir des équations sémantiques des attributs.

Ces deux modules communiquent entre eux et maintiennent ensemble une sorte de base de données de faits, dérivés à partir des règles de grammaire abstraite : des arbres abstraits attribués. La figure 3.2 , page suivante, décrit la structure d'un éditeur syntaxique généré par le CSG.

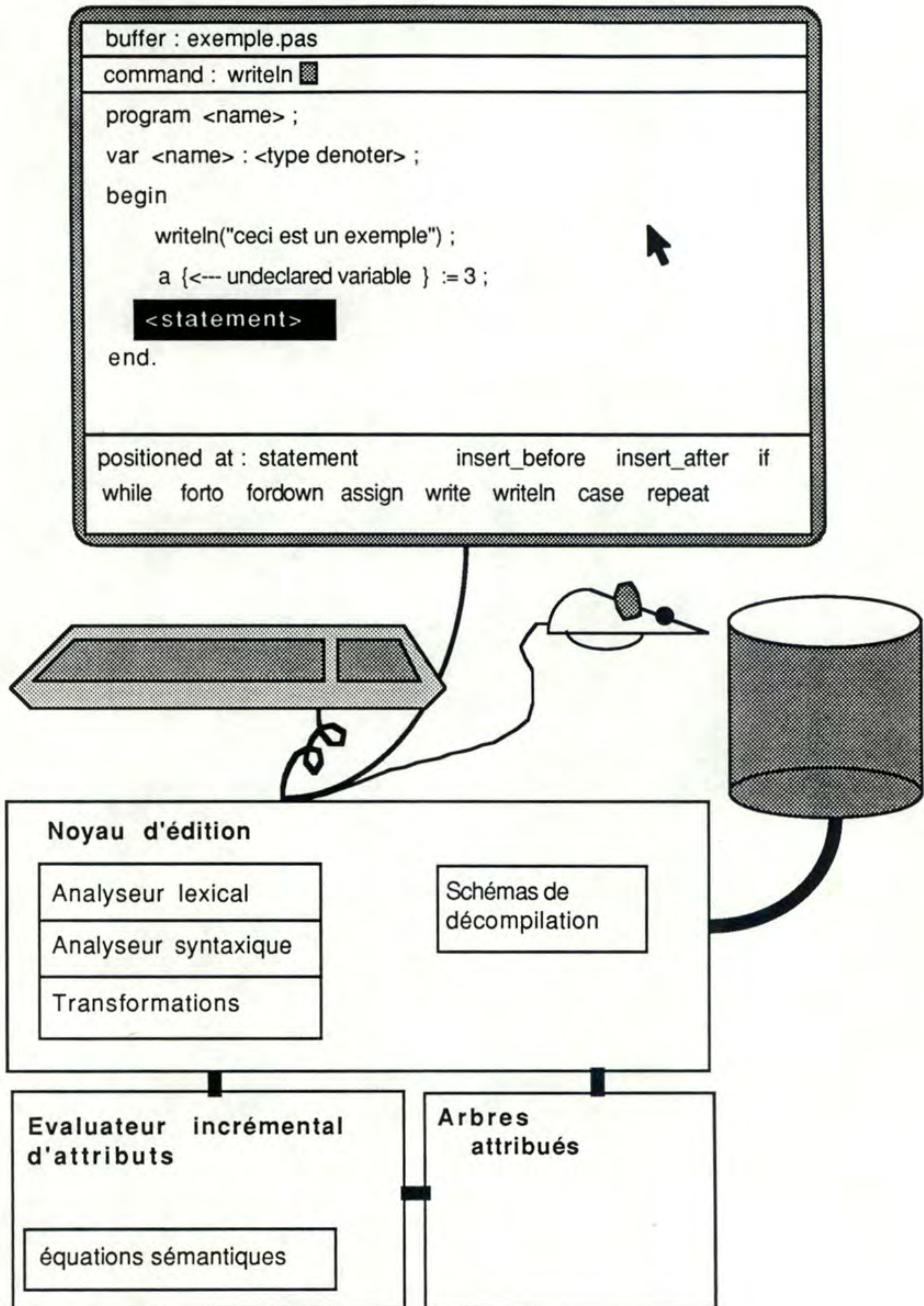


figure 3.2 : structure d'un éditeur généré par le CSG

6. La Spécification d'un Editeur Syntaxique avec le CSG

6.1. But de ce chapitre

Le but de ce chapitre est de sensibiliser le lecteur, peut-être futur utilisateur du CSG, à des problèmes que nous avons rencontrés. Nous voulons lui présenter les aspects du CSG qui nous ont semblé importants et lui proposer, en guise de réflexion, des éléments méthodologiques que nous avons dégagés par notre expérience. Il ne s'agit donc pas d'un manuel d'utilisation du CSG, mais plutôt d'un complément à celui-ci.

Ce chapitre est la conséquence d'une réalisation prévue au départ : un environnement de programmation interactif pour le langage COBOL. Plusieurs des exemples qui seront présentés décrivent des aspects du sous-système expérimental que nous avons réalisé pour la société SIEMENS à Münich.

6.2. Quelques aspects du langage SSL

Le Synthesizer Specification Language est un langage qui permet de décrire tous les éléments d'une spécification : syntaxe abstraite, attributs et équations sémantiques, syntaxe d'entrée concrète et transformations. Ce paragraphe a pour but de familiariser le lecteur avec des concepts du SSL afin qu'il puisse suivre aisément la suite du chapitre.

6.2.1. Terminologie

Un symbole non terminal définit un domaine syntaxique. Un élément de domaine syntaxique peut être identifié à son arbre de dérivation car, dans une grammaire abstraite, l'élément c'est la structure et vice-versa.

En SSL,
* un **terme** est un élément d'un domaine syntaxique,
* un **phylum**[1] est le nom donné au concept de domaine syntaxique; un phylum correspond à un symbole non terminal d'une grammaire.
* un **opérateur** est un nom permettant de distinguer les termes d'un phylum ayant des structures ou des fonctions (du point de vue sémantique) différentes; comme un terme est un n-uplet on pourra dire qu'un opérateur est "appliqué" à ses n *arguments*.

Une grammaire est l'ensemble des phyla de la spécification.

[1] Le mot anglais **phylum** (pluriel : phyla) désigne, en Histoire Naturelle, un "embranchement" qui relie différentes familles à une même espèce.

6.2.2. Le langage fonctionnel du SSL : la term algebra

6.2.2.1. Domaines de ce langage

La term algebra a pour domaines les domaines syntaxiques d'une grammaire : les phyla.

Un terme est donc à la fois un élément de domaine syntaxique, objet à éditer, et une valeur de la term algebra.

Un opérateur désigne un terme de la grammaire, et est aussi utilisé dans la term algebra soit pour dissocier les différents termes d'un phylum soit comme "constructeurt" de terme en appliquant à n arguments un opérateur n-aire.

Il existe quleques phyla "primitifs" que nous reconnaissons facilement :

BOOL	désigne les valeurs de vérité *true* et *false*,
INT, REAL	désignent respectivement les nombres signés entiers ou réels,
CHAR	désigne l'ensemble des caractères ASCII,
STR	désigne l'ensemble des chaînes de caractères.

Les valeurs primitives de ces phyla sont les arbres de dérivation dérivés d'un ensemble infini de productions de grammaire associées aux différents non terminaux (phyla) BOOL, INT, etc ... Les opérateurs définis par ces productions décrivent en fait les valeurss constantes d'un phylum donné : l'opérateur 3(), ou 3, appartient à INT et décrit la valeur *trois* ; l'opérateur true(), ou true, appartient à BOOL et décrit la valeur *vrai* .

Les autres phyla, non primitifs, sont à définir par l'utilisateur. Le lecteur remarquera l'analogie entre définir un phylum en SSL et définir un type de donnée par énumération, chaque élément de l'énumération étant un opérateur suivi de ses n arguments (ceux-ci étant eux même des phyla, primitifs ou non).

Le tableau suivant résume ce qui précède :

OBJET	point de vue GRAMMAIRE	point de vue TERM ALGEBRA
phylum	non terminal; domaine syntaxique	type de donnée; domaine syntaxique
terme	élément du domaine syntaxique arbre de dérivation; n-uplet	valeur d'une donnée stucturée n-uplet
opérateur	nom de terme	constructeur de terme; dissocie les différents termes d'un phylum

6.2.2.2. Expressions

Il est possible de définir des fonctions ayant pour arguments des termes, pour résultat un terme, et dont le corps est une **expression.**

Une variable est un nom lié à un terme. Le terme lié à une variable est la valeur de celle-ci. Différentes instances dynamiques d'une variable donnée peuvent être liées à différentes valeurs mais, une fois liée, une instance de variable donnée ne peut plus être re-liée. En d'autres termes, il n'existe pas d'instruction d'assignation.

En lieu et place de l'assignation, il y a un mécanisme de pattern matching : il s'agit de faire coïncider deux expressions, l'une ayant des variables et l'autre pas. Pour ce faire, on lie aux variables de la première expression des valeurs telles que les deux expressions ont une valeur identique.

Exemple : soit un phylum b_tree : EmptyNode() | BTreeNode(b_tree b _tree);
soit la fonction NumberOfNodes dont la valeur est le nombre de noeuds d'un b_tree, définie comme suit ...

```
INT NumberOfNodes( b_tree b ) (
        with(b) (
            EmptyNode : 0,
            BTreeNode( left, right ) : 1 + NumberOfNodes(left) + NumberOfNodes(right)
        )
    );
```

Cette fonction est composée d'une expression "with" dont la forme est :

with($expression_0$) (
$pattern_1$: $expression_1$,
...
$pattern_n$: $expresion_n$)

dont la sémantique est de faire coïncider à $expression_0$ un des pattern, en commençant par le premier. La valeur de l'expression "with" est la valeur de l'$expression_i$ associée au premier $pattern_i$ pour lequel réussit le pattern-matching.

-Une même variable v, apparaissant dans $pattern_i$, est "libre" et, apparaissant dans $expression_i$ est "liée".

- Le compilateur SSL doit pouvoir déterminer statiquement qu'il existera toujours un pattern qui coïncidera; c'est pourquoi le mot-clef "default" ou le symbole "*", qui sont des patterns qui coïncident toujours, permettent de satisfaire à cette condition. Par exemple, l'expression qui permet de déterminer si un *b_tree* est vide ou pas pourrait être :

```
with(b) (
        EmptyTree : true,
        default : false )
```

6.2.2.3. Les attributs et expressions attribuées

Les attributs peuvent être considérés comme des valeurs attachées à chaque noeud d'un arbre de dérivation c'est-à-dire à chaque terme.

Un attribut a pour valeur un **terme**. Lorsqu'on déclare un attribut, il faut spécifier le phylum auquel ses valeurs appartiendront.

Dans la "term algebra", un terme est un arbre de dérivation ... il peut donc être attribué ! Une donnée structurée est définie de façon uniforme. Une valeur de la "term algebra" est un terme, attribuable, dont les attributs sont des termes, eux-mêmes attribuables, ... à l'infini.

Une *expression attribuée* a la forme suivante :

expression{ équations }.attribut

dont la valeur est calculée comme suit :

a) l'**expression** est évluée. Sa valeur est un terme attribuable, mais non-attribué, T pour lequel **attribut** est un attribut ,
b) les valeurs des attributs hérités de T sont définies par les équations données,
c) la valeur de **T.attribut** est calculée "à la demande[1] et devient la valeur de l'expression attribuée.

Il est donc possible, par le biais d'une expression attribuée, de construire un terme et puis d'en extraire une valeur d'attribut synthétisé en lui fournissant des valeurs pour ses attributs hérités.

6.2.3. Conclusion

Dans ce paragraphe destiné à présenter quelques aspects du SSL, nous avons voulu familiariser le lecteur à la terminologie qui sera utilisée ultérieurement, et à la philosophie du langage fonctionnel du SSL. Nous n'avons donc pas été exhaustifs. D'autres aspects du SSL seront abordés dans le paragraphe suivant, au fur et à mesure des besoins.

La section trois présente la spécification d'une syntaxe abstraite et des schémas de décompilation, le section quatre traite des unités lexicales d'un langage, le section cinq étudie l'analyse syntaxique, le section six aborde la déclaration des *transformations* et le section sept propose des idées pour la vérification de la sémantique statique d'un langage.

---

[1] alors qu'un terme de l'arbre abstrait d'un programme est un arbre **attribué**, la valeur d'un attribut est un terme qui est un arbre **attribuable**; c'est-à-dire qu'on ne peut connaître la valeur d'un des attributs de ce terme qu'en **demandant** son évaluation explicite grâce à une "expression attribuée".

6.3. La syntaxe abstraite et les schémas de décompilation

6.3.1. Introduction

La *syntaxe abstraite* d'un langage joue un rôle majeur dans la spécification d'un Editeur Syntaxique :

* elle exprime la grammaire abstraite, context-free, du langage

* elle conditionne le processus d'édition des termes d'un phylum en donnant à chaque phylum une "propriété".

Les *schémas de décompilation* sont intimement liés aux phyla : ils déterminent comment chaque phylum sera représenté à l'écran. Voilà pourquoi ces deux aspects d'une spécification SSL sont présentés ensemble ... et voilà peut être aussi pourquoi, inconsciemment, nous avons spécifié ces deux aspects ensemble, lors de la réalisation de notre éditeur syntaxique COBOL.

Après avoir présenté comment un phylum est défini dans le CSG, nous nous attacherons à développer de concepts, spécifiques à l'édition syntaxique, ajoutés à une syntaxe abstraite : les productions complétive et d'emplacement, définissant le completing term et le placeholder term d'un phylum; ensuite nous évoquerons la façon de donner une représentation textuelle à un terme grâce aux schémas de décompilation; enfin nous essayerons de tirer des conclusions de notre expérience en proposant quelques éléments méthodologiques.

6.3.2. Définition d'un phylum dans le CSG

Puisqu'un phylum est un domaine syntaxique, ou encore un symbole non-terminal, définir un phylum revient à définir des productions de grammaire context-free sous la forme suivante :

$$\textbf{phylum}_0 \textbf{ : operator(phylum}_1 \textbf{ phylum}_2 \textbf{... phylum}_n \textbf{) ;}$$

Différentes productions définissant un même phylum peuvent être regroupées, factorisées; et des opérateurs d'un phylum ayant la même arité et les même phyla comme arguments peuvent être regroupés également.

Voici la grammaire abstraite, de notre petit exemple d'expressions arithmétiques, exprimée en SSL :

```
expression    : ExpNull()
              | Plus, Minus, Multiply, Divide(expression expression)
              | ConditionalExp(expression expression expression)
              | Const(INT) ;
```

Expnull() est l'opérateur de la production complétive du phylum *expression* .

6.3.3. Completing Term et Placeholder Term d'un Phylum

Pour satisfaire au modèle dédition présenté plus haut, toute spécification de phylum doit avoir une *production complétive* décrivant le **completing term** du phylum. D'autre part, dans le paradigme d'édition d'un éditeur généré par le CSG, on ne peut faire d'insertion que lorsque la sélection est un **placeholder term**, ou *emplacement* : généralement sa représentation visuelle est une phrase ou un métasymbole qui décrit la classe des objets qui peuvent être insérés à cet endroit-là.

6.3.3.1. Phylum simple

Le **completing term** d'un phylum "simple" (sans déclaration de propriété) est défini par la *production complétive* de celui-ci. Le completing term d'un phylum est le terme construit en appliquant le premier opérateur, apparaissant dans la spécification du phylum, aux completing terms de ses arguments.

Le completing term d'un phylum ne peut pas être défini de manière circulaire.

Le **placeholder term** d'un phylum "simple" est son completing term; ils sont assimilés l'un à l'autre.

<u>Exemple</u> : soit les phyla

p : P_Null(g) | ... ;

g : G_Null() | ... ;

Le terme P_Null(G_Null()) est à la fois completing term et placeholder term de p.

6.3.3.2. Phylum avec déclaration de propriété

Un phylum peut être déclaré avec une de ces trois propriétés : **optional, list** ou **optional list**. Une propriété influence la façon dont on peut éditer ou visualiser celui-ci.

6.3.3.2.1. Phylum "optional"

Un phylum déclaré *optional* est un phylum dont le completing term et le placeholder term sont *dissociés*. On ne donne pas de représentation visuelle à son completing term : ainsi rien n'apparaît à l'écran lorsqu'un tel phylum est "non-développé" (c'est à dire qu'il dérive sa production complétive) et qu'il n'est pas *sélectionné*. Le placeholder term d'un phylum optionnel a une représentation visuelle qui désigne l'*emplacement* du phylum et indique que celui-ci peut être développé.

Lorsque la sélection se pose sur un phylum optionnel non-développé, cet événement provoque le remplacement du completing term (sans représentation visuelle) par le placeholder term (un emplacement apparaît alors à l'écran). Si le phylum reste non développé, une fois que la sélection se pose ailleurs le placeholder term est remplacé par le completing term avec pour résultat apparent que l'*emplacement* disparaît de l'écran.

Le **completing term** d'un phylum optionnel est le premier opérateur de sa spécification qui a une arité nulle (sans arguments). Un phylum optionnel **doit** avoir au moins un terme dont l'opérateur est d'arité nulle.

Le **placeholder term** d'un phylum optionnel est le terme construit en "complétant" la première production du phylum différente de son completing term.

Exemple : soient les phyla

optional p;
p : P_No(g) | P_Null() | ... ;
g : G_Null() | ... ;
optional h;
h : H0() | H1() | ... ;

P_Null() est le completing term de p, P_No(G_Null()) est son placeholder term .
H0() est le completing term de h et H1() est son placeholder term.

La dissociation entre completing term et placeholder term d'un phylum n'est utile qu'à des fins d'édition. Elle n'ajoute rien à la syntaxe abstraite du langage. L'intérêt est de n'avoir représentés à l'écran que les emplacements qui *doivent* être obligatoirement développés, et de voir apparaître les emplacements facultatifs lorsque la sélection passe à l'endroit où ceux-ci *peuvent* être développés.

6.3.3.2.2. Phylum "list" ou "optional list"

La convention selon laquelle toute insertion a lieu à un emplacement (placeholder term) implique que, dans un liste, un emplacement peut apparaître avant et après chaque élément de la liste. Tous ces emplacements entre éléments d'une liste sont implicitement optionnels : lorsque nous avons placé la sélection sur un élément d'une liste et que nous la déplaçons vers l'élément suivant, un emplacement apparaît alors pour signaler qu'une insertion est possible entre les deux éléments et puis disparaît s'il n'est pas développé.

Pour avoir cet effet à l'édition, un phylum doit :
a) avoir la propriété **list** ou **optional list,**
b) être décrit par deux productions dont une est d'arité nulle et l'autre est binaire avec récursivité à droite.

Exemple : la spécification

```
list p;
p : P_Nil()  |  P_Pair( item  p ) ;
```

La différence entre les propriétés **list** ou **optional list** réside dans le comportement visuel de l'*emplacement* pour des listes vides : un phylum "optional list" est considéré comme une liste de zéro éléments ou plus, tandis qu'un phylum "list" est considéré comme une liste d'*un élément au moins*. Dans ce dernier cas, le placeholder term d'une liste vide reste affiché tant qu'au moins un élément n'a pas été inséré dans la liste.

Exemple : soient les phyla

```
list p;
p : P_Nil() | P_Pair(item  p) ;
item : ItemNull() | ... ;
optional list q;
q : Q_Nil() | Q_Pair(item  q) ;
```

Les completing term et placeholder term de p sont identiques : c'est le terme P_Pair(ItemNull(), P_Nil()).
Il n'en est pas de même pour q dont le completing term est Q_Nil(), mais dont le placeholder term est Q_Pair(ItemNull(), Q_Nil()).

6.3.3.2.3. Phylum "root"

Toute spécification SSL doit avoir une déclaration de racine qui définit le phylum (domaine) de tous les objets éditables :

root *phylum* **;**

C'est au phylum racine que se pose toujours la sélection lorsqu'on débute une séance d'édition.

6.3.4. Schémas de décompilation

Un schéma de décompilation détermine :

* la représentation textuelle d'un terme,
* les composants d'un terme que l'on peut sélectionner,
* le mode d'édition par défaut de ces composants sélectionnables.

On déclare un schéma de décompilation de la manière suivante :

phylum : opérateur [schéma de décompilation] ;

(On peut aussi faire suivre chaque description de terme d'un phylum par son schéma de décompilation entre crochets).

Un schéma de décompilation ressemble à une production : un membre gauche et un membre droit séparés par le symbole "::=" ou ":".

6.3.4.1. Mode d'Edition par défaut d'un terme

La ré-édition *textuelle* d'un terme est permise s'il existe une syntaxe d'entrée concrète pour le phylum de ce terme. En supposant que celle-ci existe, il est possible d'éditer un terme textuellement après avoir introduit la commande d'édition "**text-capture**".

Le mode "*édition textuelle par défaut* " signifie que toute opération d'édition textuelle (insertion, suppression de caractères) est précédée **implicitement** d'une commande "text-capture". Ceci évite donc à l'utilisateur le désagrément de devoir constamment introduire cette commande avant chaque opération d'édition textuelle.

Lorsque les membres gauche et droit du schéma de décompilation d'un terme sont séparés par un symbole :

"::=", il y a édition textuelle par défaut;

":" toute édition textuelle devra être précédée explicitement de la commande précitée.

6.3.4.2. Composants d'un terme qu'on peut sélectionner

Un schéma de décompilation comporte des "resting-place denoters" (indicateurs de gîte). Ce sont des symboles qui correspondent aux occurrences des phyla qui apparaissent dans la production et dont le format de représentation est défini. Les symboles "@", " ^ " et ".." sont des indicateurs de gîte.

Le membre gauche du schéma de décompilation est un indicateur de gîte unique (exclusivement "@" ou " ^ ") et correspond au membre de gauche de la production qui décrit le terme auquel le schéma de décompilation est associé. Le membre droit du schéma de décompilation comporte n indicateurs de gîte qui correspondent aux n phyla du membre droit la production associée.

Un terme est un gîte si le sommet de son arbre de dérivation peut devenir le sommet de la sélection :

".." indique que le sous-terme correspondant n'est *pas visualisable* et qu'*il ne s'agit pas* d'un gîte.

"@" indique que le sous-terme correspondant est à visualiser et qu'il s'agit d'un gîte.

" ^ " indique que le sous-terme correspondant est à visualiser mais que celui-ci n'est un gîte que si le membre droit du schéma de décompilation de ce sous terme est le symbole "@".

Exemple : soient les phyla

```
p : P( b )   [ @ ::= ^ ] ;
b : B( c )   [ ^ : ^ ] ;        /* ce terme n'est pas un gîte ! */
c : C( d )   [ @ ::= @ ] ;
```

Supposons que la sélection est sur le terme P(B(C(...))).
Lorsqu'on avance la sélection au plus proche noeud fils de ce terme qui est un gîte, celle-ci devient C(...). Car la sélection se déplace toujours de gîte en gîte : on peut ainsi cacher la structure "fine" d'un arbre de dérivation en ne permettant à la sélection de se reposer que sur les noeuds "intéressants", les gîtes.

6.3.4.3. Représentation textuelle d'un terme

Le membre droit d'un schéma de décompilation peut comporter des éléments de décompilation (unparsing items) qui peuvent être :

* des chaînes de caractères entre guillemets (c'est à dire, dans la terminologie du CSG : des constantes du phylum STR entre guillemets).

* des occurrences de phyla de la production à décompiler.

* des attributs de ces mêmes occurrences de phyla de la production à décompiler.

* des éléments de décompilation conditionnels, spécifiques aux phyla *list* ou *optional list*

.

Un élément de décompilation conditionnel permet d'afficher cet élément uniquement lorsque la production binaire associée à son schéma de décompilation n'est pas la dernière d'une liste.

Exemple :

```
list p;
p : P_Nil()             [ ^ : ]
  | P_Pair( STR  p )  [ @ : ^ [", "] @ ] ;
```

Le terme P_Pair("**a**", P_Pair("**b**", P_Nil()))
sera décompilé comme **a, b** (on remarque que la virgule n'apparaît pas après b)

Reprenons notre exemple préféré : on peut écrire

```
expression : ...
           |  Divide(expression expression)
                  { local STR error;
                        error = with(expression$3)[1] (
                                    Const(0) : "<-- division by zero ",
                                    default  : " " );
                  }
                  [ @ ::= @ "/" @ error ]
           |  ConditionalExp(expression  expression  expression)
                  [ @ ::= "if" @ "then" @ "else" @ ]
           |  ... ;
```

avec cette spécification, le terme Division(Const(4), Const(0)) sera décompilé comme : "**4 / 0 <-- division by zero**".

Tandis que le terme Division(Const(4), Const(2)) sera décompilé comme : "**4 / 2**".

Provoquer la visualisation d'attributs lors de la décompilation d'un terme est un moyen très commode d'annoter l'écran pour prévenir l'utilisateur d'erreurs.

[1] expression$3 signifie "la troisième occurrence du phylum *expression* dans la production"; le membre gauche de la production étant ici considéré comme expression$1; le membre gauche d'une production est aussi noté $$.

6.3.5. Eléments méthodologiques

6.3.5.1. Un même élément de spécification couvre plusieurs aspects

Le lecteur aura remarqué que, en SSL, définir un phylum c'est plus que définir une règle de production et définir un schéma de décompilation c'est plus que déterminer la représentation visuelle d'un terme; ces spécifications sont chargées d'un contenu supplémentaire, ont une toile de fond dont il faut tenir compte : le "but du jeu" est de réaliser un éditeur syntaxique.

Voilà pourquoi il y a ces fameuses **propriétés**, ces **placeholder term** et **completing term** dont il faut tenir compte lorsqu'on définit les productions d'un phylum. Voilà pourquoi en décrivant un schéma de décompilation on définit trois choses à la fois : la représentation textuelle du terme, son mode d'édition par défaut et les sous-termes sélectionnables.

Un concepteur doit tenir compte des différentes facettes que revêt un seul élément de spécification. Lorsque nous avons commencé la réalisation de notre éditeur syntaxique COBOL l'édition syntaxique elle-même était pour nous quelque chose de tout à fait nouveau : nous soupçonnions comment nous voulions que cet éditeur se comporte mais nous n'avions pas déterminé de modèle de dialogue entre l'éditeur et l'utilisateur.

6.3.5.2. Considérer un seul aspect à la fois

Il faut, nous semble-t-il, considérer **un** aspect des spécifications à la fois, ou peut-être quelques aspects complémentaires :

* d'abord la grammaire abstraite, proprement dite, du langage,

* et, peut-être parallèlement, les schémas de décompilation en n'en considérant *que* l'aspect "représentation textuelle",

* ensuite, seulement, considérer les aspects restants des schémas de décompilation, une fois qu'il est possible d'avoir une vision globale de la syntaxe abstraite, une fois qu'il est possible de s'imaginer comment doit se dérouler le processus d'édition.

6.3.5.3. Deux erreurs à ne pas commettre

En ce qui concerne la syntaxe abstraite proprement dite, il y a deux erreurs à ne pas faire lorsqu'on la spécifie :

* écrire une grammaire trop générale (au point qu'elle devienne "générique" ...),

* écrire une grammaire trop chargée de la sémantique du langage.

La première erreur se commet lorsqu'on n'a pas compris qu'une syntaxe "abstraite" n'a d'abstrait que le nom. Elle a quelque chose de concret dès qu'elle est exprimée dans un

formalisme particulier (le SSL dans ce cas) et qu'elle décrit la grammaire d'**un** langage particulier. Elle déshabille le langage pour mettre à nu les concepts qui le composent.

La deuxième erreur se commet lorsqu'on se dit "je vais essayer de donner à la grammaire abstraite un contenu sémantique, le plus riche possible".

Exemple : Nous avions dissocié, en COBOL, nom de section, nom de paragraphe et nom de procédure ... Ce ne sont pas des concepts différents car un nom de section ou de paragraphe est un nom de procédure (au sens COBOL du terme).

La spécification qui suit laisse à désirer :

```
section_name : SecNameNo()
        | SecName( procedure_name ) ;

paragraph_name : ParaNameNo()
        | ParaName( procedure_name ) ;

procedure_name : ProcNameNo()
        | ProcName( STR ) ;
...

section : SectionNo()
        | Section( section_name  paragraph_list ) ;

paragraph : ParagraphNo()
        | Paragraph( paragraph_name  sentence_list );
...
```

Une meilleure façon de spécifier tout cela est de supprimer les phyla "paragraph_name" et "section_name" et redéfinir les phyla "section" et "paragraph" comme suit :

```
section : SectionNo()
        | Section( procedure_name paragraph_list );

paragraph : ParagraphNo()
        | Paragraph( procedure_name sentence_list);
```

Pour notre part, il nous a paru difficile de spécifier une "bonne" syntaxe abstraite du premier coup. Nous avons remis plusieurs fois en question l'existence de certains phyla. Nous avons également cherché à exprimer, dans un arbre abstrait de programme, "assez" de sémantique pour pouvoir réaliser un compilateur incrémental. Nous avons par exemple spécifié, pour le COBOL, une chaîne picture de description de donnée comme un phylum composé de termes structurés et non comme une simple chaîne de caractères[1].

---

[1] Nous renvoyons le lecteur intéressé à l'annexe qui présente nos spécifications du langage COBOL, à la description que nous donnons d'une chaîne picture.

6.3.5.4. Adapter une spécification au comportement qu'on veut donner à un éditeur syntaxique

Pour un langage syntaxiquement riche il est difficile de trouver **la** syntaxe abstraite, d'autant plus qu'en toile de fond doit rester l'idée que le "but du jeu" est de réaliser un éditeur syntaxique.
Alors, faudra-t-il spécifier :

```
ifstmt : IfThen( condition statement )
        | IfThenElse( condition statement statement ) ;
```

ou bien :

```
ifstmt2 : IfThenElse2( condition statement else_statement ) ;
optional else_statement;
else_statement : ElseNo()
        | ElseYes( statement );           ?
```

Ces deux spécifications décrivent le même concept, mais provoquent un comportement différent de l'éditeur ... c'est le comportement que l'on veut donner à l'éditeur qui détermine la réponse à la question. Dans le premier cas il faudra, si l'on veut ajouter une partie "else" à un "if" représenté par le terme *IfThen(x , y)* :

1) sélectionner et couper la condition *x* , et la mémoriser dans un buffer a,
2) sélectionner et couper le statement *y* , et le mémoriser dans un buffer b,
3) sélectionner le terme *IfThen* et le couper,
4) greffer un terme *IfThenElse* ,
5) coller le contenu des buffers a et b à leur place; on a le terme *IfThenElse(x, y, ?)*
6) enfin (!) développer la partie "else" ...

Dans le second cas :

1) après avoir posé la sélection sur la partie "then"
2) avancer la sélection au noeud suivant; la sélection est maintenant sur la partie "else", non développée, dont le placeholder term apparaît ...
3) Il suffit maintenant de développer la partie "else" .

6.3.5.5. Des conventions d'écriture

La lecture et la mise au point d'une syntaxe abstraite peut devenir une tâche fort ardue une fois que les spécifications prennent une certaine ampleur; c'est pourquoi nous avons adopté la convention d'écriture proposée par le manuel du CSG :

- les noms de phyla en lettres minuscules, par exemple : ifstmt

- les noms d'opérateurs sont la concaténation de mots qui commencent par une lettre majuscule et continuent en lettres minuscules, par exemple : IfThenElse.

Cette convention rend une spécification désagréable à dactylographier, mais nous avons pu

nous rendre compte par expérience qu'elle rend une telle spécification beaucoup plus lisible.

Placer le completing term d'un phylum au début de sa spécification; dans notre appendice contenant la spécification d'une syntaxe abstraite COBOL, le lecteur remarquera qu'une majorité des spécifications de phyla ont un premier terme dont l'opérateur est d'arité zéro : c'est leur completing term.

Pour les phyla à propriété *list* ou *optional list* , nous conseillons de spécifier d'abord le terme avec opérateur d'arité zéro et puis seulement le terme avec opérateur binaire, afin que le compilateur ne donne pas une erreur pour "phylum dont le completing term est défini circulairement ou qui dérive d'un tel phylum ".

6.4. Les unités lexicales du langage

6.4.1. Introduction

Le rôle d'un analyseur lexical est de lire des caractères en entrée, et de produire une suite de symboles (typiquement les symboles terminaux d'une grammaire). L'analyseur syntaxique détermine ensuite si cette suite de symboles peut être dérivée des règles de grammaire concrète, si elle appartient au langage. Cette introduction présente quelques éléments théoriques relatifs à l'analyse lexicale inspirés de [AHO & ULLMAN 1986].

Les raisons qui poussent à séparer l'analyse syntaxique et l'analyse lexicale sont :
* une conception générale plus simple (l'analyseur lexical peut facilement supprimer des blancs et des commentaires, alors qu'il est moins facile d'incorporer des conventions pour les séparateurs et les commentaires dans un analyseur syntaxique),
* une conception plus soignée et plus efficace de chaque analyseur,
* une portabilité plus grande : l'analyseur syntaxique peut ignorer l'alphabet propre à une machine (ASCII ou EBCDIC), c'est l'analyseur lexical qui s'en occupe.

Après un rappel, dans cette introduction, des concepts de lexème, expression régulière et automate fini, nous présenterons au lecteur la façon de spécifier des lexèmes en SSL; comme la génération d'un analyseur lexical à partir de spécifications SSL utilise Lex, un outil UNIX de génération d'analyseurs lexicaux, nous présenterons celui-ci avant d'évoquer des éléments méthodologiques pour contourner les faiblesses de Lex et tirer parti de certains de ses mécanismes.

6.4.1.1. Lexèmes, modèles, tokens

Un lexème est une suite de caractères dans le programme source qui coïncide avec le modèle d'une unité lexicale ("token"). Le tableau suivant décrit la relation qui existe entre un token, son modèle et un lexème.

Token	Exemple de lexème	description informelle du modèle
literal	"toto"	tous les caractères entre " et ", excepté " lui-même
num	33	une constante numérique quelconque
identifier	Arthur23	une lettre suivie par des lettres ou des chiffres

Un token peut être considéré comme un symbole terminal de la grammaire concrète. Il faudra définir les tokens décrivant les mots-clefs du langage ainsi que les classes de lexèmes telles que les identificateurs, les nombres, ...

6.4.1.2. Spécification d'un token

6.4.1.2.1. Chaînes et langages

Un **alphabet** ou classe de caractères est n'importe quel ensemble fini de symboles; par exemple { 0, 1 } est l'alphabet binaire.

Une **chaîne** s sur un alphabet est une suite finie de symboles extraits de cet alphabet; la longueur d'une chaîne s est notée $|s|$; par exemple *toto* est une chaîne de longueur quatre (4). La chaîne vide, notée ε, est une chaîne de longueur zéro.

Un **langage** désigne n'importe quel ensemble de chaînes sur un alphabet donné. $\emptyset$, l'ensemble vide, est un langage, conformément à notre définition; $\{ \varepsilon \}$ est un langage; l'ensemble des phrases syntaxiquement correctes de COBOL ou de la langue française sont aussi des langages.

La **concaténation** des chaines x et y, notée xy est une chaîne formée en ajoutant y à x; on peut considérer la concaténation comme un "produit" dont ε est l'élément identité : $\varepsilon s = s\varepsilon = s$.

On peut définir l'exponentiation de chaînes comme suit :

$s^0 = \varepsilon$

$s^i = s^{i-1}s$ pour $i > 0$

donc $s^0 = \varepsilon$, $s^1 = s$, $s^2 = ss$, $s^3 = sss$, etc...

6.4.1.2.2. Opérations sur les langages

Si L et M sont des langages et, s et t sont des chaînes :

* l' **union** de L et M, notée $L \cup M$
est : $L \cup M = \{ s \mid s \in L \text{ ou } s \in M \}$

* la **concaténation** de L et M, notée LM
est : $LM = \{ st \mid s \in L \text{ et } t \in M \}$

* la **fermeture de Kleene**, notée L*
est : $L^* = L^0 \cup L^1 \cup \ldots \cup L^\infty$ c'est à dire zéro ou plusieurs concaténations de L.

* la **fermeture positive**, notée L+
est : $L+ = L^1 \cup \ldots \cup L^\infty$ c'est à dire une ou plusieurs concaténations de L.

* $L = \varepsilon$ et $L^i = L^{i-1}L$ pour $i > 0$ c'est à dire que L^i est concaténé i-1 fois à lui-même.

6.4.1.2.3. Expressions régulières

Une expression régulière r désigne un langage L(r). Les règles pour la définition d'expressions régulières sur un alphabet Σ sont :

1) ε est une expression régulière (e.r.) qui représente le langage { ε } (l'ensemble composé d'une chaîne vide).

2) si **a** est un symbole de Σ alors **a** désigne une e.r. qui représente { **a** } c'est à dire l'ensemble qui contient la chaîne **a**. Techniquement l'e.r. **a** est différente de la chaîne **a** ou du symbole **a**; bien que **a** désigne trois choses, le contexte rendra explicite sa signification.

3) soient r et s, des e.r. représentant les langages L(r) et L(s), alors

- (r) | (s) est une e.r. qui représente L(r) U L(s)
- (r)(s) est une e.r. qui représente L(r)L(s)
- (r)* est une e.r. qui représente (L(r))*
- (r) est une e.r. qui représente L(r) c'est-à-dire que des paires de parenthèses peuvent être placées n'importe où dans une e.r. afin de la rendre plus lisible.

Un langage représenté par une e.r. est un *ensemble régulier*[1].

Les conventions suivantes permettent d'éviter les parenthèses superflues :

- l'opérateur unaire * est le plus prioritaire et est associatif à gauche
- la concaténation a une priorité inférieure et est associative à gauche
- "|" a la priorité la plus faible et est associative à gauche.

6.4.1.2.4. Abrévation de la notation des e.r.

Les abréviations suivantes rendent plus commode l'écriture d'expressions régulières :

1) l'opérateur + signifie "une ou plusieurs instances de ..." et a la même priorité que "*"

r+ = rr* et r* = r+ | ε

2) l'opérateur ? signifie "zéro ou une instance de ... "

r? = r | ε

3) les classes de caractères sont définies comme suit :

[abc] = a | b | c

[a-z] = a | b | c | ... | x | y | z

Par exemple les nombres non signés du COBOL sont décrits par l'expression régulière suivante : [0-9]+(.[0-9]+)?

Des identificateurs, commençant par une lettre et continuant par des lettres ou des chiffres sont définis par : [A-Za-z][A-Za-z0-9]*

[1] Certains langages ne peuvent pas être décrits par des expressions régulières : ce sont des ensembles non réguliers. Les ensembles réguliers peuvent être utilisés pour décrire seulement un nombre fixé au départ ou non spécifié de répétitions d'une construction donnée. Deux nombres arbitraires ne peuvent être comparés pour voir s'ils sont égaux. Par exemple une chaîne Hollerith FORTRAN : $nHa_1...a_m$ ne peut être reconnue par une expression régulière car le nombre de caractère suivant H doit être égal au nombre n qui précède H.

6.4.1.3. Les automates finis

Un "analyseur" pour un langage est un programme qui prend en entrée une chaîne x et répond "oui" si x est une phrase du langage et non sinon. Les expressions régulières peuvent être reconnues par des analyseurs appelés automates finis.

6.4.1.3.1. Les automates finis non-déterministes

Un automate fini non-déterministe (AFN) est un modèle mathématique qui consiste en :

1) un ensemble d'états S,
2) un ensemble de symboles d'entrée Σ (l'alphabet des symboles d'entrée),
3) une fonction de transition move qui fait correspondre un ensemble d'états à une paire symbole-état (move : état x symbole --> {état, ... }),
4) un état s_0 : l'*état initial*, ou encore *état de départ,*
5) un ensemble d'états F : l'ensemble des *états finals*, ou encore *états d'acceptation.*

On peut représenter un AFN par un diagramme appelé graphe de transition, ou par une table de transition dans laquelle il y a une ligne par état et une colonne pour chaque symbole d'entrée, et ε si nécessaire; l'entrée pour l'état s_i et le symbole "a" dans la table est l'ensemble des états qui peuvent être atteints par une transition de l'état i lorsqu'un symbole "a" surgit dans la chaîne d'entrée.

Exemple : l'expression régulière (a|b)*abb peut être reconnue par l'AFN dont le graphe de transition est ...

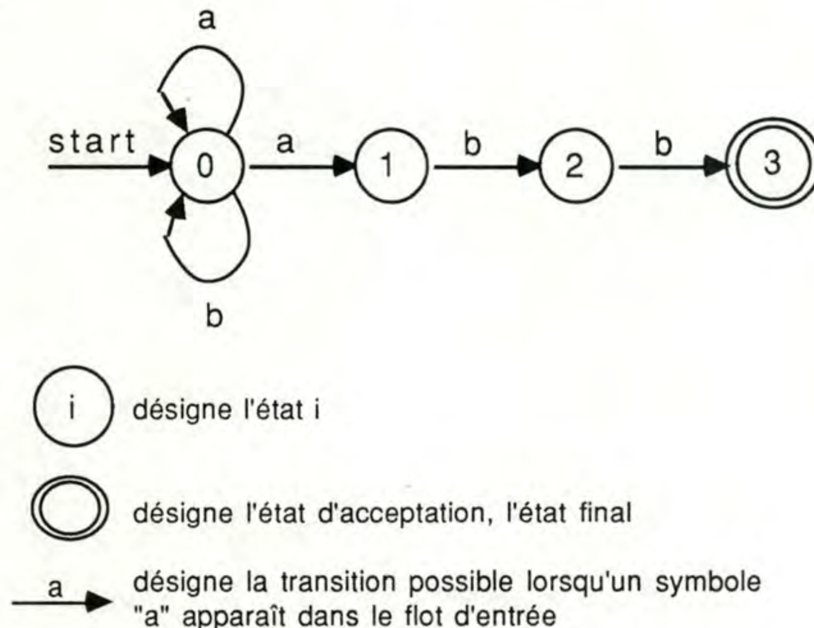


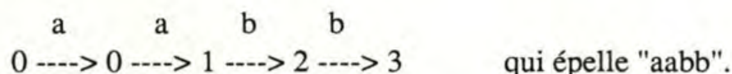
figure 4.4.1 : automate fini non-déterministe pour (a|b)*abb

La table de transition de cet AFN est :

état	symbole d'entrée a	b
0	{0,1}	{0}
1	-	{2}
2	-	{3}

Le langage défini par un AFN est l'ensemble des chaînes d'entrée qu'il "accepte". Un AFN "accepte" une chaîne d'entrée x si et seulement si un chemin existe dans son graphe de transition partant de l'état initial et aboutissant à un état final, tel que les étiquettes des arcs le long de ce chemin épellent la chaîne x.

aabb est accepté car il existe un chemin :



qui épelle "aabb".

6.4.1.3.2. Les automates finis déterministes

Un automate fini déterministe (AFD) est un cas particulier d'un automate fini non-déterministe dans lequel :

1) aucun état n'a de transition pour ε en entrée,

2) pour tout couple état-symbole (s, a), il y a *au plus un* arc étiqueté "a" partant de l'état s dans le graphe de transition de l'automate.

Un AFD possède au plus une transition à partir de chaque état pour n'importe quel symbole d'entrée : chaque entrée d'une table de transition d'un AFD est *un seul* état (et non un ensemble d'états comme avec un AFN).

Exemple : le graphe de transition d'un AFD qui accepte l'expression régulière (a|b)*abb

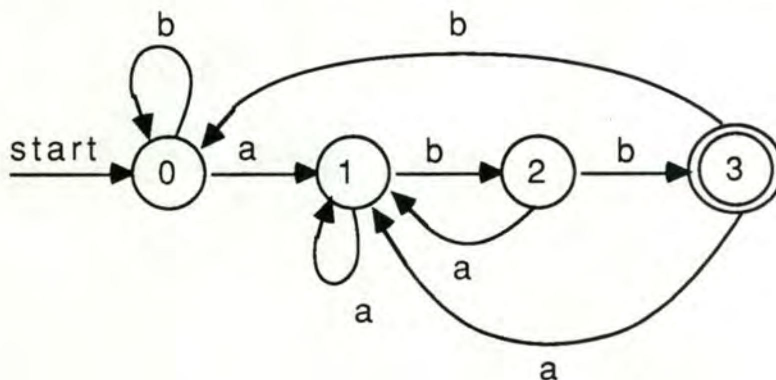


figure 4.4.2 : automate fini déterministe pour (a|b)*abb

6.4.2. Description de token en SSL

6.4.2.1. Forme générale

En SSL, un token désigne un domaine syntaxique : l'ensemble des chaînes qui coïncide avec une expression régulière. Un phylum peut donc être décrit sous forme d'un ensemble de productions ou sous forme d'un ensemble de *tokens*.

Un token peut être défini comme suit :

phylum : token_name < expression régulière > ;

Un phylum comprenant un ensemble de tokens peut être spécifié comme :

phylum : token_name < e.r. >
| token_name < e.r. >
| ...
| token_name < e.r. >
;

Nous appellerons de tels phyla des "phyla lexicaux" pour les dissocier des phyla productions d'une grammaire abstraite que nous appellerons "phyla abstraits".

L'ensemble ordonné des phyla lexicaux d'une spécification sert à définir un analyseur lexical. Lors d'une édition textuelle, à chaque fois que l'analyseur syntaxique aura besoin d'un token, il invoquera l'analyseur lexical qui lira le texte d'entrée pour y reconnaître un lexème; une fois un lexème reconnu l'analyseur lexical enverra le token correspondant pour l'analyseur syntaxique.

L'analyseur lexical essayera toujours de trouver comme lexème le plus long préfixe possible du texte d'entrée. Lorsqu'un lexème correspond à plusieurs tokens, l'analyseur lexical renverra le token dont l'expression régulière apparaît le plus tôt dans le code SSL.

Au cas où aucun lexème ne peut être reconnu dans le texte d'entrée, c'est le premier caractère de celui-ci qui est renvoyé comme token.

Une fois qu'un token est reconnu, le préfixe correspondant du texte d'entrée est "consommé" : l'analyse lexicale suivante portera sur le suffixe restant.

6.4.2.2. Définition d'une expression régulière en SSL

Le processus de génération d'un éditeur syntaxique à partir de spécifications SSL réalise un analyseur lexical en invoquant Lex, un outil UNIX de génération d'analyseurs lexicaux. Lex, qui sera décrit plus loin, utilise également des expressions régulières. Les conventions d'écriture de celles-ci sont identiques en SSL et en Lex.

Les expressions régulières en SSL peuvent être spécifiées comme suit : soient **c**, un caractère, **e** une expression régulière, et **n** un entier ...

c	le caractère "c".
"$c_1c_2c_3$"	la chaîne $c_1c_2c_3$.
\c	le caractère c (\ est un caractère d'échappement).
[$c_1c_2c_3$]	les caractères c_1 ou c_2 ou c_3.
[c1-c2]	...
[^$c_1c_2c_3$]	tout caractère sauf c1 ou c2 ou c3.
.	tout caractère sauf celui de passage à la ligne (newline).
^e	une expression régulière e en début de ligne.
e$	e en fin de ligne.
e?	zéro ou une fois e.
e*	zéro ou plusieurs fois e.
e+	une ou plusieurs fois e.
e_1e_2	e_1 suivi de e_2.
e1|e2	e_1 ou e_2.
(e)	e.
e1/e2	une expression régulière e1 seulement si elle est suivie par une expression régulière e2.
e{n1,n2}	de n_1 à n_2 occurences de e.

Les caractères
" \ [ ] - ^ . $ * + | ( ) / { } % < >
doivent être précédés du caractère d'échappement ou apparaître entre guillemets si l'on veut les utiliser en tant que caractère litéral. Sinon ils ont une signification particulière.

6.4.2.3. Un lexème particulier : WHITESPACE

Le phylum lexical dont le nom est WHITESPACE a une signification particulière : lorsqu'un lexème corespondant est reconnu dans le texte d'entrée, il est "consommé" à l'intérieur de l'analyseur lexical, sans renvoyer de token. Ce phylum n'est pas prédéfini : il faut en fournir une déclaration adéquate, propre à chaque langage.

Exemple : En cobol, un "séparateur" est :

- un ou plusieurs espaces, ou tabulations.
- un ou plusieurs passages à la ligne.
- un point-virgule ou une virgule suivi d'au moins un espace ou passage à la ligne ou tabulation.

Nous définissons donc WHITESPACE comme : **WHITESPACE : < [;,] ? [\ \t \n]+ >;**
où \t et \n désignent respectivement la tabulation et le passage à la ligne, comme en C.
"\ " désigne le caractère d'espacement; il doit être précédé d'un caractère d'échappement pour pouvoir être reconnu.

6.4.2.4. Sensibilité au contexte

Parfois il est utile de tenir compte de ce qui a été lu auparavant dans le texte d'entrée. Il est possible de spécifier qu'un lexème ne peut être reconnu que dans un cas bien particulier.

L'analyseur lexical généré sera un automate fini déterministe. Nous pouvons nommer l'état de départ et l'état d'acceptation d'une expression régulière en écrivant :

phylum : token_name < < start > e < accept > >;

Dans ce cas, l'expression régulière **e** ne pourra être reconnue que si l'analyseur lexical est préalablement dans l'état nommé **start**. Une fois un lexème de **e** reconnu, l'analyseur lexical entrera dans l'état **accept**.

<u>Exemple</u> : En COBOL, une ligne est un commentaire lorsqu'au début de cette ligne apparaît le caractère '*' et ce qui suit est le commentaire proprement dit. Nous ne permettrons donc de reconnaître le lexème d'une ligne de commentaire qu'après avoir reconnu la marque de commentaire en début de ligne :

```
COMMENT_DENOTER : < ^[\ \t\n]*"*" < comment_ok > >;
COMMENT_LINE : < < commment_ok > .* >;
```

Une expression régulière sans état initial est <u>toujours active</u>, c'est-à-dire qu'on peut toujours en reconnaître un lexème. Une expression régulière avec un état **s** n'est active que lorsque l'analyseur lexical est dans l'état **s**.

6.4.3. Lex : un générateur d'analyseurs lexicaux

6.4.3.1. Description générale

Les spécifications d'unités lexicales (token) en SSL sont traduites par le compilateur SSL en spécifications pour Lex, un utilitaire UNIX de génération d'analyseurs lexicaux. Comme le comportement d'un analyseur lexical est intéressant à connaître pour écrire de bonnes spécifications, nous nous attacherons à présenter en bref ce qu'est Lex, comment s'écrit une spécification Lex et comment le compilateur SSL traduit des spécifications de phyla lexicaux en spécifications Lex.

Les spécifications fournies à Lex constituent une table d'expressions régulières, associées à des fragments de programme.

expression régulière_1	action_1
...	...
expression régulière_n	action_n

A partir de cela, Lex génère un automate fini déterministe qui lit un texte en entrée et essaye de reconnaître des lexèmes qui coïncident avec les expressions régulières fournies en spécification. A chaque fois qu'un lexème est reconnu, le fragment de programme, associé à l'expression régulière qui y coïncide, est exécuté. On peut donc dire que les fragments de programme spécifiés par l'utilisateur sont exécutés dans l'ordre dans lequel les instances d'expressions régulières correspondantes sont reconnues dans le texte d'entrée.

Le programme d'analyse lexicale que Lex génère accepte des spécifications ambigües :

- il essaye de trouver un lexème dans le plus long préfixe possible du texte d'entrée.
- lorsqu'un lexème peut appartenir à plusieurs expressions régulières (celles-ci définissent des ensembles réguliers dont l'intersection est non-vide), c'est l'expression régulière qui apparaît le plus tôt dans la spécification qui est choisie et c'est le fragment de programme correspondant qui est exécuté. Ce comportement influencera l'ordre dans lequel le concepteur doit écrire ses spécifications.

6.4.3.2. Forme d'une spécification Lex

La structure d'une spécification Lex est :

déclarations
%%
règles
%%
sous-programmes
de l'utilisateur.

Une règle a la forme suivante :

< start_state > expression régulière { action };

La partie **action** n'est pas obligatoire et est une suite d'instructions en langage C. Lex génère un analyseur lexical sous forme d'un programme C qui peut être associé à un analyseur syntaxique. Un état de départ n'est pas obligatoire mais lorsqu'on le spécifie, il doit être déclaré préalablement dans la partie **déclarations** de la spécification :

% Start nom_1, nom_2, ... où nom_1, nom_2 sont des états.

On entre dans un état final (au sens d'un lexème SSL) en ajoutant à la partie action :

BEGIN état_final

Les conventions d'écriture d'expressions régulières du SSL sont celles de Lex; elles ont déjà été présentées plus haut.

Une règle qui ne possède pas d'état initial est toujours active, c'est-à-dire que son expression régulière est toujours susceptible d'être reconnue. Tandis qu'une règle avec état initial n'est active que dans cet état-là. Nous pouvons formaliser cette propriété de la manière suivante :

Soit P l'ensemble des règles d'une spécification Lex
$S_1, \ldots, S_n$ les ensembles des règles actives dans les états $s_1, \ldots, s_n$ respectivement
F l'ensemble des règles toujours actives,
On peut dire que
* $P = F \cup S_1 \cup S_2 \cup \ldots \cup S_n$.
* $S_1, \ldots, S_n$ sont des ensembles disjoints.
* à l'état s_i, $F \cup S_i$ est l'ensemble des règles actives.
* à l'état spécial INITIAL, seul F est l'ensemble des règles actives.

Conséquence intéressante : on peut considérer les règles d'un ensemble S_i comme spécifiant un analyseur lexical autonome propre à un contexte particulier, l'état s_i. Un analyseur lexical généré à partir de P sera donc composé d'un ensemble d'analyseurs lexicaux auxiliaires propres à chaque contexte d'état s_i, dans lesquelles les règles de F sont aussi actives.

6.4.3.3. Passage d'une spécification de lexème SSL en une règle Lex

Le lecteur aura remarqué la forte similitude entre la spécification d'un lexème en SSL et une règle de Lex. Le compilateur SSL traduit les spécifications de lexèmes SSL en une passe, "on the fly".

Mieux que de longues explications ce schéma montre le processus de traduction :

en SSL : **phylum : token_name < <start_state> e <final_state> >;**
devient ...

en Lex : **<start_state>e { BEGIN final_state; ... return (token_name_TOKEN); };**

où **e** est une expression règulière, et **token_name_TOKEN** est le nom d'une constante entière associée au token dont le modèle est l'expression régulière e.

Ces règles Lex sont générées dans un fichier appelé **scan.l**, et les déclarations d'état sont générées dans un fichier appelé **scan.h**.
Lorsqu'un nom d'état nouveau n apparaît dans la spécification SSL, le compilateur génère "%Start n " dans scan.h. La spécification complète pour Lex sera donc la concaténation des fichiers scan.h et scan.l.

6.4.4. Eléments méthodologiques

6.4.4.1. Introduction

Le lecteur aura remarqué que certaines caractéristiques du SSL sont étroitement liées aux caractéristiques de Lex : la façon d'écrire des expressions régulières, la possibilité de spécifier des états de départ et d'arrivée, la façon dont les ambiguïtés et conflits sont résolus.

Notre expérience nous a appris qu'il y a une "bonne façon" d'écrire les spécifications de lexèmes en SSL afin d'obtenir un analyseur lexical de taille minimale : en effet, Lex génère un automate fini déterministe dont la taille des tables augmente fortement en fonction du nombre et de la complexité des règles spécifiées.

Ce qui suit présente quelques idées, fruits de notre expérience (et de quelques déboires), qui aideront le concepteur à bien spécifier la partie "analyse lexicale" d'un éditeur syntaxique.

Il n'est malheureusement pas possible de tout expliquer. Le lecteur intéressé pourra trouver plus d'informations dans le manuel du CSG et dans le manuel du Lex. Nous nous sommes limités à dire ce qui, à notre connaissance, n'est dit nulle part ailleurs.

6.4.4.2. Ordre des spécifications

L'ensemble des spécifications de lexèmes doit être ordonné. Il faut que le concepteur garde à l'esprit ces deux caractéristiques de l'analyseur lexical qui est généré :
- il tente de reconnaître un lexème qui correspond au plus long préfixe possible du texte d'entrée.
- au cas où plusieurs tokens peuvent être reconnus, c'est le premier de ceux-ci qui apparaît dans la spécification qui est choisi.

Que faut-il en déduire ?

Nous pouvons considérer l'ensemble des définitions de lexèmes en SSL comme une suite d'expressions régulières $e_1, \dots, e_m$ définissant une suite d'ensembles reguliers $L(e_1), \dots, L(e_m)$. Il faut ordonner les expressions régulières d'une spécification de telle manière que $| L(e_1) | \leq | L(e_2) | \leq \dots \leq | L(e_m) |$. En d'autres termes, il faut d'abord spécifier les mots-clefs du langage et puis les tokens qui décrivent des classes de lexèmes de plus en plus grandes (c'est-à-dire la classe des identificateurs, etc...).

Exemple : Soient les spécifications

PROGRAM : ProgramLex < "PROGRAM"("ME")?> | programLex < "program"("me")?>;
IDENTIFIER : identifierLex < [A-Za-z][A-Za-z0-9]* >;

si le texte d'entrée est **PROGRAMME**
où PROGRAMME $\in$ L(PROGRAM) $\cap$ L(IDENTIFIER), c'est le token ProgramLex, qui apparaît le plus tôt dans la spécification, qui sera renvoyé.

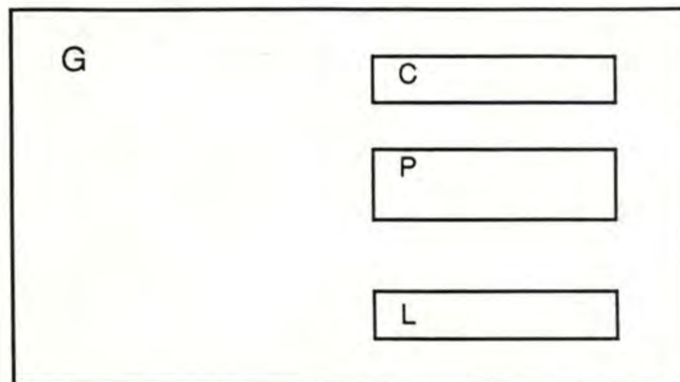
6.4.4.3. Utilité des états de départ et d'arrivée

Une expression régulière sans état de départ est toujours active, c'est-à-dire qu'elle est toujours susceptible d'être choisie si un lexème qui y coïncide est reconnu.

Une expression régulière **avec** état de départ n'est active que dans cet état-là. C'est une caractéristique très utile car elle permet de décrire des analyseurs lexicaux propres à chaque contexte. C'est ainsi qu'il est possible de spécifier plusieurs petits analyseurs lexicaux "spécialisés", auxiliaires d'un analyseur lexical plus "général".

Pour illustrer ceci, voici la structure de l'analyseur lexical COBOL que nous avons réalisé :
* Un analyseur lexical "général" reconnaît les mots-clefs, les identificateurs, les littéraux numériques.
* Un analyseur lexical "auxiliaire" reconnaît les littéraux alphanumériques avec continuation : en COBOL, un littéral alphanumérique peut être "continué" sur une ou plusieurs lignes selon des règles bien précises.
* Un analyseur lexical "auxiliaire" reconnaît les chaînes "picture" qui déterminent le format d'une donnée. Comme ces chaînes obéissent à des règles de formation particulières, nous avons écrit un mini analyseur lexical qui y est spécialement dédié. Il est activé après que l'analyseur "général" ait reconnu le lexème "PICTURE IS".
* Un analyseur auxiliaire pour les lignes de commentaires.

Le synoptique de notre analyseur lexical COBOL est :



Chaque boîte décrit un analyseur lexical.
G : analyseur "général" ; contexte G,
C : analyseur de litéraux alphanumériques avec continuation; contexte C,
P : analyseur de chaînes "picture"; contexte P,
L : analyseur pour les lignes de commentaires; contexte L.

figure 4.4.3 : synoptique de notre analyseur lexical COBOL

Lorsqu'on spécifie des analyseurs lexicaux auxiliaires, il faut porter son attention sur le nombre de règles **toujours actives** (sans états de départ). Si celles-ci sont nombreuses, nous avons remarqué que les tables générées par Lex sont très volumineuses car, dans **chaque** contexte donné, l'analyseur doit tenir compte, à la fois, des règles actives dans ce contexte **et** des règles toujours actives.

C'est pourquoi nous conseillons, après l'avoir expérimenté, de spécifier un minimum de règles toujours actives (une dizaine au plus) afin de diminuer très sensiblement la taille des tables de l'analyseur généré et de diminuer son temps de génération.

Exemple :
Avec une centaine de règles toujours actives et
- une règle active dans le contexte C,
- 4 dans le contexte P,
- 1 dans le contexte L,

il faut plus de deux heures à Lex pour générer plus ou moins 400 K de code.

Avec deux règles toujours actives
- une centaine dans le contexte G,
- 4 dans le contexte P,
- 1 dans le contexte C,
- 1 dans le contexte L,

il faut 15 minutes à Lex pour générer plus ou moins 180 K de code.

6.4.4.4. Convention d'écriture

Dans les quelques exemples qui précèdent le lecteur aura remarqué que nous utilisons une convention d'écriture particulière :
- un nom de phylum lexical est en majuscules,
- un nom de token est en un mot en lettre minuscules auquel on ajoute la chaîne "Lex".

Cette convention rend plus aisée la lecture d'une spécification, et facilite la mise au point d'un éditeur syntaxique.

6.4.4.5. Modification du compilateur SSL

Le manuel du CSG décrit une structure de spécification de lexème comme :

phylum : lexeme_name < < start_state > e.r. < final_state > >;

Dans le cas précis du phylum WHITESPACE, il est impossible de spécifier un < final_state >, le compilateur SSL répond par un laconique " > missing ".

Soit c'est un oubli dans le CSG, soit c'est un choix délibéré considérant que ce cas-là ne se présenterait pas.

Comme nous avions besoin de cette possibilité, nous avons modifié le compilateur SSL de telle manière que la description donnée dans le manuel soit valable pour tout nom de phylum, même pour WHITESPACE. Le lecteur trouvera en annexe une description de ce qu'il faut faire pour réaliser cette modification.

6.4.4.6. Conclusion

Nous avons vu dans la section précédente que le SSL se distingue par sa multiplicité d'aspects. C'est encore le cas pour la spécification des unités lexicales d'un langage : il faut connaître "plus" que le SSL en lui-même; il faut être familiarisé avec les expressions régulières et avec Lex, qu'on utilise finalement sans le savoir.

Nous conseillons donc au futur concepteur de bien lire le manuel du Lex [LESK & SCHMIDT]. Nous espérons aussi que ces quelques éléments méthodologiques lui donneront un éclairage supplémentaire pour maîtriser les subtilités de l'analyse lexicale.

6.5. L'analyse syntaxique

6.5.1. Introduction

Le CSG permet la génération d'éditeurs textuels-structurels. Quelle que soit la proportion de mode textuel que l'on désire, il est nécessaire de donner une syntaxe d'entrée concrète pour chaque terme que l'on veut pouvoir éditer textuellement.

Dans le chapitre "De theoria programmorum", le passage d'un arbre concret à un arbre abstrait est évoqué. Nous allons voir dans cette section quelques rappels sur l'analyse syntaxique, quel principe d'édition textuelle est utilisé dans le CSG, comment on spécifie une syntaxe d'entrée concrète en SSL, quel lien il y a entre une syntaxe d'entrée concrète en SSL et YACC, un utilitaire UNIX de génération d'analyseurs syntaxiques et, enfin, nous proposerons au lecteur quelques éléments méthodologiques.

6.5.2. Rappels

6.5.2.1. Généralités sur l'analyse syntaxique

En toute généralité, un analyseur syntaxique vérifie si la structure syntaxique d'un programme est bien-formée par rapport aux règles de grammaire du langage cible. Ces règles peuvent être décrites avec des grammaires context-free.

L'intérêt d'utiliser une telle grammaire est de pouvoir donner une spécification précise et facile à comprendre de la syntaxe d'un langage de programmation, de pouvoir générer automatiquement un analyseur syntaxique qui vérifie si un programme source est syntaxiquement correct, de pouvoir facilement adapter une grammaire au fur et à mesure de l'évolution d'un langage.

Le "modèle" général d'analyse syntaxique est celui-ci : l'analyseur syntaxique reçoit une chaîne de tokens de l'analyseur lexical et vérifie si cette chaîne peut être générée à partir de la grammaire du langage source.

figure 4.5.1 : Schéma de coopération entre analyseurs syntaxique et lexical.

Les deux classes d'analyseurs syntaxiques les plus utilisées sont appelées "top-down" ou "bottom-up".
Une analyse syntaxique "top-down" construit l'arbre concret à partir de la racine de l'arbre (top) jusqu'aux feuilles (down) tandis qu'une analyse syntaxique "bottom-up" construit l'arbre concret en commençant par les feuilles (bottom) et en remontant jusqu'à la racine (up). Chacune de ces méthodes analyse la chaîne de tokens d'entrée de gauche à droite, un token à la fois. Le CSG utilise YACC, un générateur d'analyseurs syntaxiques qui produit un analyseur "bottom-up".

6.5.2.2. Rappel sur les grammaires context-free

[AHO & ULLMAN 1986] définissent une grammaire context-free comme un ensemble de symboles terminaux, non-terminaux, un symbole de départ (start-symbol) et des productions où :
1) les **terminaux** sont les symboles de base à partir desquels les chaînes sont formées. On les appelle aussi "tokens" ou lexèmes.
2) les **non-terminaux** sont des variables syntaxiques qui désignent des ensembles de chaînes.
3) le **start-symbol** ou symbole de départ désigne l'ensemble des chaînes qui constituent le langage défini par la grammaire (cfr le phylum *root* d'une grammaire abstraite en SSL).
4) les productions d'une grammaire spécifient comment peuvent être combinés terminaux et non-terminaux du langage pour former des chaînes. Une production a la forme :
non-terminal ---> chaîne de non-terminaux et de terminaux
(souvent on remplace "--->" par "::=").

Voici la grammaire context-free de notre exemple d'expressions arithmétiques :

```
expression   ::=   expression '+' expression
             |     expression '-' expression
             |     expression '*' expression
             |     expression '/' expression
             |     if expression then expression else expression
             |     integer

integer      ::=   number
             |     integer number

number   ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

En général, le start-symbol est le membre gauche de la première production. Plusieurs productions ayant le même membre gauche peuvent être rassemblées en une seule, qui comprend des alternatives séparées par le caractère '|'.

Nous dirons que A "dérive" ß s'il existe une production A --> ß. La relation "dérive en une étape" est notée "==>" , ainsi nous pouvons écrire que A ==> ß. Nous dirons que A dérive B en plusieurs étapes si A ==> ß ==> ... ==> B.
"==>+" signifie "dérive en au moins une étape",
"==>*" signifie "dérive en zéro ou plusieurs étapes".

Soit une grammaire IG avec un start-symbol "YYentry", nous pouvons utiliser la relation ==>+ pour définir L(IG), le **langage généré** par IG. Les chaînes de L(IG) peuvent contenir uniquement des symboles terminaux (token) de IG. Nous dirons qu'une chaîne de terminaux *s* est dans L(IG) si et seulement si YYentry ==>+ *s* . La chaîne *s* est appelée **phrase** (sentence) de IG.

Un langage qui peut être généré par une grammaire telle que définie plus haut est un langage "context-free". Si deux grammaires génèrent le même langage, elles sont dites "équivalentes".

Si YYentry ==>+* ß, où ß peut contenir des non-terminaux, nous dirons que ß est une *forme sentencielle* (sentential form) de IG. Une *phrase* (sentence) est une forme sentencielle sans aucun non-terminal.

Un **arbre de dérivation** décrit l'ordre dans lequel les règles de grammaire ont été appliquées pour dériver une phrase du langage. La forme de l'arbre dépend des choix de dérivation qu'on a effectués : une "*dérivation gauche* " consiste à dériver d'abord le non-terminal situé le plus à gauche dans une production; tandis qu'une "*dérivation droite* " consiste à dériver d'abord le non-terminal situé le plus à droite dans une production.

Exemple : dérivons "3 + 4 + 5" conformément aux règles de grammaire présentées plus haut ...

son arbre de dérivation droite est :

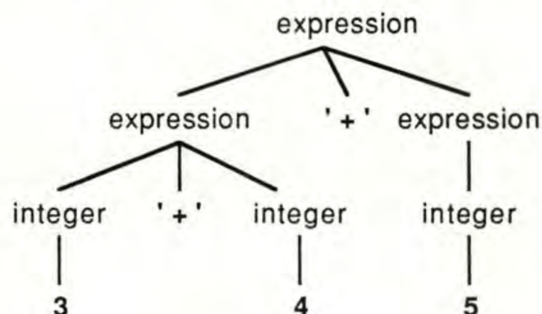


figure 4.5.2 : Arbre de dérivation droite de "3+4+5"

son arbre de dérivation gauche est :

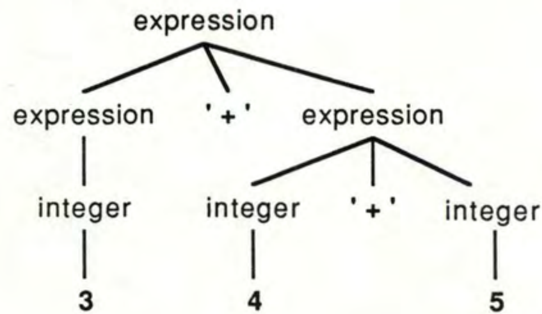


figure 4.5.3 : Arbre de dérivation gauche de "3+4+5"

Une grammaire ambiguë est une grammaire qui produit plus d'un arbre de dérivation gauche ou droite pour la même phrase.

Exemple : soit la grammaire

A ::= **if** e **then** A (1)
| **if** e **then** A **else** A (2)
| ...

Cette grammaire est ambiguë car la chaîne

if e_1 **then if** e_2 **then** a_1 **else** a_2 (s)

peut avoir deux arbres de dérivation gauche :

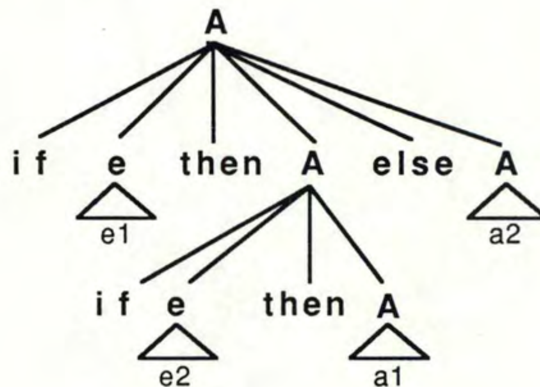


figure 4.5.4 a : Premier arbre de dérivation gauche

ou encore ... (voir page suivante)

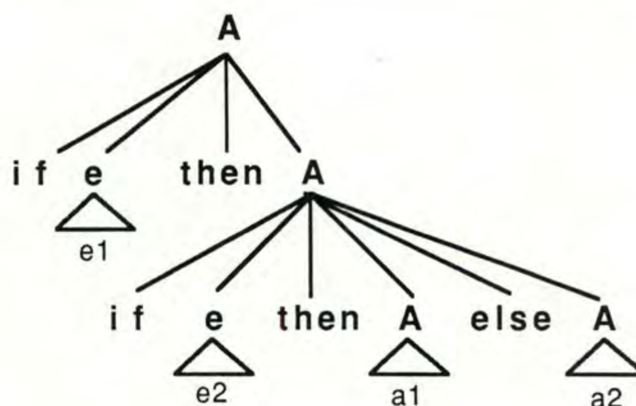


figure 4.5.4b : Second arbre de dérivation gauche

Cette ambiguïté résulte du fait qu'il est possible de choisir d'abord la production (1) ou la production (2) pour une dérivation droite de la chaîne (s).

6.5.2.3. L'analyse syntaxique "bottom-up"

Un éditeur syntaxique généré par le CSG utilise un analyseur syntaxique "bottom-up" (bottom-up parser) qui met en oeuvre une technique appelée "shift/reduce parsing". Nous pourrions dire bien des choses sur les différentes techniques d'analyse syntaxique. Nous nous limiterons au strict nécessaire en priant le lecteur intéressé par le sujet de trouver une bibliographie exhaustive dans [AHO & ULLMAN 1986]. C'est de cet ouvrage que nous nous sommes inspirés pour tout ce qui concerne l'analyse syntaxique et lexicale : plusieurs exemples en sont extraits.

Le "shift/reduce parsing" essaye de construire un arbre concret pour une chaîne d'entrée en commençant par les feuilles et continuant vers le haut jusqu'à la racine (bottom-up). On peut considérer que ce processus revient à "réduire" en symbole de départ de la grammaire (start-symbol) une chaîne s.

A chaque étape de réduction un préfixe particulier de la chaîne d'entrée, qui coïncide avec le membre droit d'une production, est remplacé par le symbole du membre gauche associé. Si le préfixe est choisi correctement à chaque étape, une "dérivation droite" est réalisée à l'envers.

Exemple : Considérons la grammaire G dont les productions sont

S --> aABe	(r1)
A --> Abc	(r2)
	b	(r3)
B --> d	(r4)

La phrase **abbcde** peut être réduite en **S** par les étapes de réduction suivantes :

abbcde	réduction de **b** en **A** par r3
aAbcde	réduction de **Abc** en **A** par r2
aAde	réduction de **d** en **B** par r3
aABe	réduction de **aABe** en **S** par r1
S	

On peut donc dire que abbcde ∈ L(G). Cette suite de réductions retrace à l'envers la "dérivation droite" suivante : S ==> aABe ==> aAde ==> aAbcde ==> abbcde

Pour réaliser un analyseur par la technique du "shift/reduce parsing" on utilise une pile pour mémoriser des symboles grammaticaux, et un tampon d'entrée qui contient une chaîne *s* à analyser. Si l'on marque par le symbole "$" le fond de la pile et la fin du tampon d'entrée, avant l'analyse syntaxique de la chaîne *s* nous avons la configuration suivante :

pile	tampon
$	s$

L'analyseur fait glisser zéro ou plusieurs symboles sur la pile (shift) jusqu'à ce que la suite de symboles empilés coïncide avec un membre droit d'une production, auquel cas on "réduit" (reduce) la pile en dépilant la suite de symboles reconnue et en empilant le symbole non-terminal du membre gauche de la production correspondante.

L'analyseur répète ce processus jusqu'à ce que la pile contienne le symbole de départ (start-symbol) et que le tampon soit vide :

pile	tampon
$S	$

Dans ce cas l'analyseur s'arrête et déclare que la chaîne *s* appartient bien au langage L(G).

Exemple : nous reprenons l'exemple précédent et présentons un tableau qui schématise le processus de "shift/reduce parsing" de la chaîne **abbcde**.

	pile	tampon d'entrée	action
1	$	abbcde$	shift
2	$a	bbcde$	shift
3	$ab	bcde$	réduire par A --> b
4	$aA	bcde$	shift
5	$aAb	cde$	shift
6	$aAbc	de$	réduire par A --> Abc
7	$aA	de$	shift
8	$aAd	e$	réduire par B --> d
9	$aAB	e$	shift
10	$aABe	$	réduire par S --> aABe
11	$S	$	accepter

Pourquoi, à l'étape 5, y a-t-il eu une action *shift* au lieu d'une réduction par A --> b ? Dans ce dernier cas le contenu de la pile serait devenu *$aAA* et il aurait été impossible par la suite d'arriver par réduction au start-symbol *S*.

Un analyseur "shift/reduce" doit donc opérer de bons choix. Comment ? Cette question sort du cadre de ce mémoire; nous ne nous y attarderons donc pas car, y répondre n'ajoute rien à la compréhension de ce qui suit.

6.5.3. Principe d'édition textuelle du CSG

Ce qui précède nous a montré qu'un terme n'est manipulable que lorsque la sélection s'est posée sur lui : lorsqu'il est sélectionné.

Une fois que la sélection s'est posée sur un terme *t* éditable textuellement, toute opération d'édition textuelle déclenche le processus que voici :
- sans que l'utilisateur s'aperçoive de quoi que ce soit, la représentation visuelle v(t) du terme *t* est transférée dans un buffer de texte[1] dont nous appellerons *s* le contenu.
- l'opération d'édition textuelle est appliquée à *s* .
- toutes les opérations d'édition texuelle suivantes seront appliquées à *s* jusqu'à ce que l'utilisateur décide de poser la sélection sur un autre terme. L'édition textuelle de *t* est terminée.
- Avant de déplacer la sélection :

1) *s* est analysé conformément aux règles de syntaxe concrète d'un phylum *pc* ; un arbre syntaxique *tc* est construit.

2) *tc* est ensuite attribué. Un de ses attributs, *abs* par exemple, représente l'arbre abstrait (terme) correspondant au texte *s* .

3) le terme *t* est remplacé par la valeur de *tc.abs* . L'arbre concret *tc* et ses attributs sont détruits.

- La sélection se pose ailleurs.

La figure 4.5.5, page suivante, retrace sous forme de schéma les différentes étapes de l'édition textuelle d'un terme abstrait :

---

[1] Nous rappelons au lecteur que chaque buffer est associé à un phylum (domaine syntaxique) particulier; car un buffer est destiné à recevoir un **arbre autonome** d'un domaine syntaxique donné. Un phylum spécial "text" est prédéfini. Un buffer de texte est donc associé au domaine syntaxique *text* sur lequel toutes les manipulations textuelles sont permises.

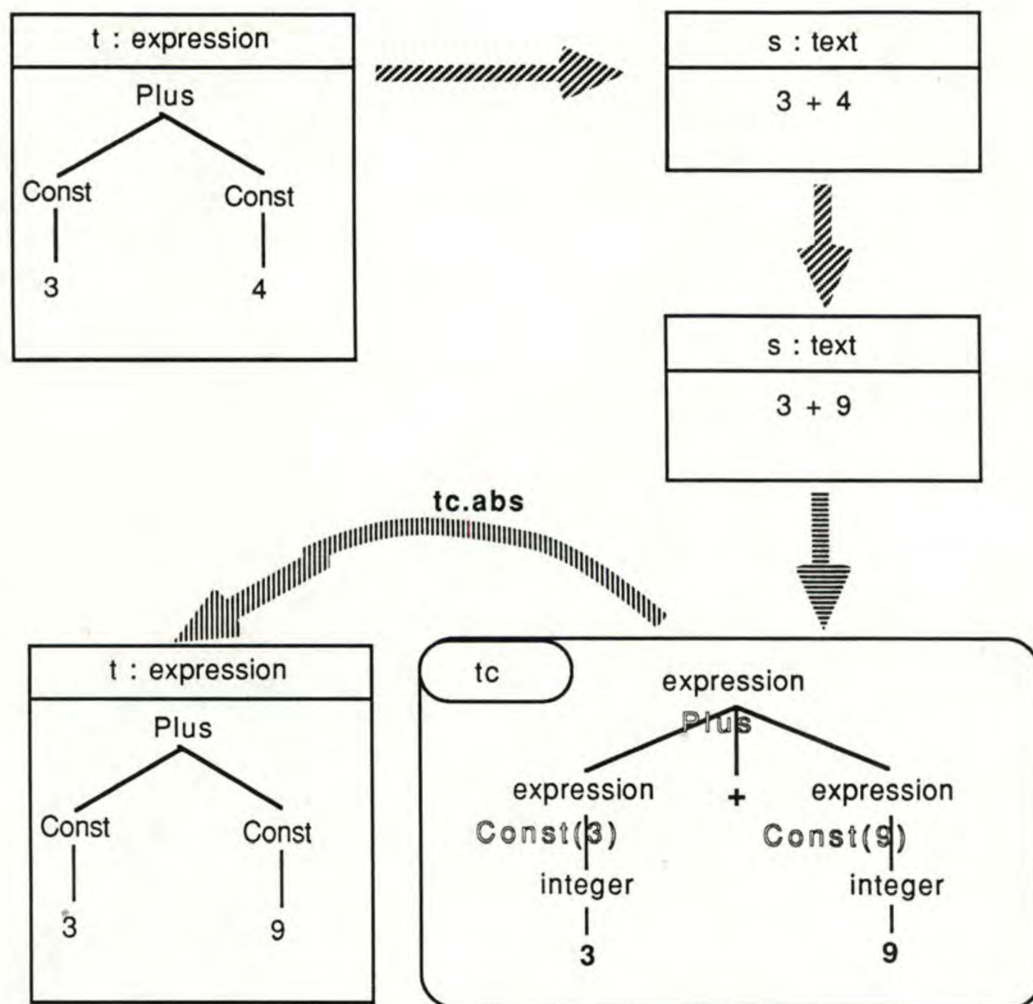


figure 4.5.5 : Schéma du processus d'édition textuelle d'un terme

Pourquoi séparer arbre abstrait et arbre concret ?

1) D'abord parce que arbre abstrait et arbre concret ont chacun un but et une structure différents. Le but d'un arbre concret est de montrer qu'une chaîne de symboles est dérivable des règles de syntaxe concrète, tandis que le but d'un arbre abstrait est d'exprimer au mieux la sémantique du langage : de ce fait la structure de chacun leur est propre.

2) Cette dissociation permet en outre l'utilisation de toute la puissance du mécanisme d'attribution pour construire un arbre abstrait. Nous verrons plus loin combien ce mécanisme est précieux.

3) Il est possible de récupérer des erreurs d'entrée fréquentes et de construire néanmoins un arbre abstrait correct.

4) Cette séparation crée la possibilité de concevoir des langages d'entrée qui ne se limitent pas à une représentation concrète du langage-cible d'un éditeur : le langage d'entrée peut contenir, par exemple, des phrases qui sont reconnues comme des commandes d'édition. D'autre part, cette séparation permet, à partir d'une même syntaxe abstraite, de construire des éditeurs syntaxiques propres à des dialectes ou à des formalismes différents d'un même langage. Seule la syntaxe d'entrée concrète différera.

Le processus d'entrée textuelle d'un terme comporte donc deux étapes :
1) analyse lexicale et construction d'un arbre concret.
2) attribution de l'arbre concret et *remplacement* de l'ancien terme de l'arbre abstrait par le nouveau terme qui vient d'être construit par attribution.

L'exposé qui suit présente au lecteur comment ces étapes sont réalisées en SSL.

6.5.4. Spécification d'une syntaxe d'entrée concrète en SSL

6.5.4.1. Déclarations d'entrée

Afin qu'un phylum *p* de la grammaire abstraite puisse être édité textuellement, il **faut** que soit spécifiée au moins une *déclaration d'entrée* pour *p* : c'est elle qui établit le lien entre un attribut d'un phylum concret et le phylum *p* abstrait (nous présenterons plus loin comment on spécifie un phylum concret).

Une *déclaration d'entrée* est spécifiée comme suit :
abstract_syntax_phylum ~ <start_state>concrete_syntax_phylum.attribut{équations};

<start_state> et *{équations}* sont facultatifs

Lorsque le terme de la sélection appartient au phylum de syntaxe abstraite, toute chaîne résultant de l'édition textuelle de celui-ci doit être analysée en utilisant les règles de dérivation du phylum de syntaxe concrète; la valeur de l'attribut du "phylum concret" doit être insérée dans l'arbre abstrait pour remplacer le terme de la sélection après édition textuelle de ce dernier. Le compilateur SSL doit pouvoir déterminer statiquement que la valeur d'*attribut* aura nécessairement pour domaine : *abstract_syntax_phylum* .

Pour attribuer l'arbre concret en tenant compte du contexte dans lequel se trouve la sélection, il est possible de déterminer des valeurs d'attribut héritées pour le *concrete_syntax_phylum* par l'intermédiaire de la partie *équations* de la déclaration d'entrée.

De plus, afin que l'analyseur lexical puisse également tenir compte du contexte, il est possible de placer celui-là dans un état de départ déterminé, par l'intermédiaire de la partie *<start_state>*.

6.5.4.2. Spécification d'une grammaire concrète

La grammaire d'entrée IG (Input Grammar) est définie comme suit :

a) le symbole de départ (start-symbol) de IG est YYentry,

b) les symboles non-terminaux de IG sont tous les *phyla concrets* de la spécification SSL ainsi que le symbole YYentry[1],

c) les symboles terminaux de IG incluent tous les caractères ASCII, tous les tokens (dont nous avons parlé à la section précédente), et un symbole spécial *p_ENTRY* pour chaque déclaration d'entrée avec *p* comme membre gauche (p ~ pc.abs, par exemple...).

d) les productions de IG sont définies dans les paragraphes qui suivent. Il y a une production par terme de phylum concret, une pour chaque *déclaration d'entrée* et une pour chaque token de phylum lexical.

La déclaration

phylum : token_name <expression régulière>;

définit *phylum* comme un ensemble régulier dont les éléments sont des chaînes qui coïncident avec l'expression régulière donnée. Cette déclaration contribue à la grammaire concrète IG par la production

phylum --> token_name

où *token_name* est considéré comme un symbole terminal de IG qui sera transmis par l'analyseur lexical à l'analyseur syntaxique.

Un *phylum concret* [2] est défini différemment d'un "phylum abstrait"

$phylum_0$ ::= operator(charconst $phylum_1$ charconst ... $phylum_n$ charconst);

charconst désigne une suite de constantes de type CHAR, c'est-à-dire une suite éventuellement vide de caractères placés chacun entre apostrophes (ex: 'a' 'b' '\'' 'w'). *opérator* est le nom d'opérateur du terme défini avec une "production concrète". En pratique il n'est pas utilisé. C'est pour rester cohérent avec la notation phylum/opérateur qu'on peut le spécifier, mais ce n'est pas obligatoire, contrairement aux noms d'opérateur des phyla abstraits qui sont, eux, obligatoires. Ce qui différencie une production de la grammaire concrète d'une production de la grammaire abstraite c'est que les deux membres d'une production concrète sont séparés par le symbole "::=" plutôt que par ":"[3]. Une production concrète contribue à IG par la production

$phylum_0$ --> charconst $phylum_1$ charconst ... $phylum_n$ charconst

[1] Nous attirons l'attention du lecteur sur le fait que **seuls** les phyla de la grammaire concrète contribuent aux productions de IG.

[2] Nous désignons un phylum de la grammaire concrète par l'appellation *phylum concret*. *Phylum abstrait* désigne un phylum de la grammaire abstraite et *phylum lexical* désigne un phylum défini par une expression régulière.

[3] Le manuel du CSG appelle *lexème* ce que nous appelons *token*, et appelle *token* ce que nous appelons *charconst*. Nous n'avons pas utilisé, pour ces concepts, la terminologie du manuel afin de rester cohérents avec la définition que nous avons donnée de *token* et de *lexème* : un *token* est le nom d'un ensemble régulier; un *lexème* est un élément d'un ensemble régulier ou encore une instance d'expression régulière.

Plusieurs termes d'un même phylum concret peuvent être regroupés comme suit :
$\textbf{phylum}_0$ ::= $\textbf{operator}_1$(charconst $\textbf{phylum}_1$ charconst ... $\textbf{phylum}_n$ charconst)

| ...

| $\textbf{operator}_m$(charconst $\textbf{phylum}_1$ charconst ... $\textbf{phylum}_n$ charconst)

;

Pour chaque déclaration d'entrée

p ~ pc.abs;

une production

YYentry --> p_ENTRY pc

est ajoutée à IG, de façon à fournir un "point d'entrée" dans l'analyseur syntaxique. Ainsi, lorsque la sélection est un terme du phylum *p*, on considère *pc* comme un symbole racine en lequel la chaîne d'entrée (le texte résultant d'une opération d'édition textuelle) doit être "réduite". Cette réduction est réalisée après avoir préalablement ajouté en début de chaîne le symbole *p_ENTRY* : l'analyseur syntaxique peut donc réduire le texte d'entrée ainsi modifié en *YYentry*, le véritable symbole racine de la grammaire concrète.

Exemple : la syntaxe d'entrée concrète de notre exemple d'expressions arithmétiques.

```
yexpression { synthesized expression abs }; /* l'attribut abs est synthétisé et a pour
                                        domaine le phylum expression */
yexpression    ::=    ( yexpression '+' yexpression )
                      { $$.abs = Plus(yexpression$2.abs, yexpression$3.abs ); }
               |      ( yexpression '-' yexpression )
                      { $$.abs = Minus(yexpression$2.abs, yexpression$3.abs ); }
               |      ( yexpression '*' yexpression )
                      { $$.abs = Multiply(yexpression$2.abs, yexpression$3.abs ); }
               |      ( yexpression '/' yexpression )
                      { $$.abs = Divide(yexpression$2.abs, yexpression$3.abs ); }
               |      ( IF yexpression THEN yexpression ELSE yexpression )
                      { $$.abs = ConditionalExp(yexpression$2.abs, yexpression$3.abs,
                                        yexpression$4.abs ); }
               |      ( INTEGER )
                      { $$.abs = Const(STRtoINT(INTEGER)); }
               ;
```

/* les équations sémantiques de *abs* pour chaque terme se trouvent entre "{" et "}";
la notation yexpression$n désigne la n[ième] yexpression qui apparaît dans une production de grammaire, en comptant de gauche à droite; le symbole $$ désigne le non-terminal du membre gauche de la production. */

```
expression ~ yexpression.abs; /* point d'entrée de expression dans IG */

INTEGER : integerLex < [0-9]+ > ;
IF : ifLex < "if" > ;
THEN : thenLex < "then" > ;
ELSE : elseLex < "else" > ;
```

Cette grammaire concrète contribue à IG de la façon suivante :
YYentry --> yexpression
YYentry --> expression_ENTRY yexpression
yexpression --> yexpression '+' yexpression
yexpression --> yexpression '-' yexpression
yexpression --> yexpression '*' yexpression
yexpression --> yexpression '/' yexpression
yexpression --> IF yexpression THEN yexpression ELSE yexpression
yexpression --> INTEGER
INTEGER --> integerLex
IF --> ifLex
THEN --> thenLex
ELSE --> elseLex

6.5.4.3. Résolution des ambiguïtés

En lisant l'exemple qui précède, le lecteur se sera demandé comment résoudre les ambiguïtés, comment spécifier la priorité des opérateurs...

6.5.4.3.1. Déclaration d'associativité et de priorité

Il est possible de déclarer la priorité des opérateurs afin que l'analyseur syntaxique réalise de "bons choix". Trois types de déclaration sont possibles, pour spécifier respectivement l'associativité à gauche, à droite ou la non-associativité :

left $phylum_or_char_1$, ... $phylum_or_char_n$;
right $phylum_or_char_1$, ... $phylum_or_char_n$;
nonassoc $phylum_or_char_1$, ... $phylum_or_char_n$;

où chaque *phylum_or_char* est soit une constante du phylum CHAR (un caractère entre apostrophes) ou un phylum lexical. Chaque instance de déclaration assigne le même niveau de priorité et la même associativité à l'ensemble des $phylum_or_char_i$ qui y sont associés. Les niveaux de priorité augmentent, déclaration après déclaration.

Lorsqu'un phylum lexical a reçu une priorité par une des déclarations susmentionnées, il faut donner **explicitement** le niveau de priorité ad hoc aux productions où ce phylum lexical apparaît :

DIFF : diffLex < "<>" > ;
nonassoc DIFF;
ycondition ::= (yexpression DIFF yexpression **prec** DIFF)

"prec phylum_or_char" en fin de membre droit d'une production assigne à la production correspondante de IG le même niveau de priorité que le caractère ou le token spécifié après "prec".

Lorsqu'une constante CHAR, dont on a spécifié la priorité, apparaît dans une production de phylum concret, cette production reçoit **implicitement** le niveau de priorité de cette constante.

Le compilateur SSL utilise YACC [JOHNSON 1978] pour générer un *shift/reduce parser* . Les règles suivantes s'appliquent en ce qui concerne les niveaux de priorité :
a) les associativités et priorités sont enregistrées pour les caractères et les tokens présents dans des déclarations *left*, *right* ou *nonassoc*.
b) les productions avec l'option *prec* ont la priorité et l'associativité du caractère ou du phylum lexical spécifié dans cette option. Les productions sans option *prec* reçoivent la priorité de la dernière constante CHAR de la production. Une production sans option *prec* ou sans constante CHAR n'ont aucune priorité ni aucune associativité.

Exemple : les ambiguïtés de l'exemple précédent sont levées par les déclarations suivantes
left '+', '-'; /* donne à l'addition et à la soustraction mêmes priorité et associativité */
left '*', '/' ; /* donne à la multiplication et à la division mêmes priorité et associativité;
leur priorité est supérieure à celle de l'addition et de la soustraction */

6.5.4.3.2. Conflits lors de l'analyse syntaxique

Malgré les déclarations d'associativité et de priorité que nous venons d'étudier, il arrive que se présentent des ambiguïtés (que le concepteur peut ne pas avoir aperçues), inhérentes au *shift/reduce parsing* :
a) conflit shift/reduce -> connaissant le contenu de la pile et le prochain symbole à lire, l'analyseur syntaxique ne sait pas s'il doit procéder à une opération d'empilage (shift) ou de réduction (reduce),
b) conflit reduce/reduce -> connaissant le contenu de la pile et le prochain symbole à lire, il faut procéder à une réduction ... mais deux productions peuvent être utilisées et l'analyseur syntaxique ne sait pas laquelle choisir.

Sans règles qui enlèvent les ambiguïtés, un analyseur syntaxique généré par YACC prend les décisions par défaut que voici :
a) en cas de conflit shift/reduce, l'action par défaut est shift (empiler le symbole suivant) ... ce qui est satisfaisant dans bien des cas (nous verrons pourquoi plus loin).
b) en cas de conflit reduce/reduce, l'action par défaut consiste à opérer une réduction par la production qui apparaît le plus tôt dans la spécification.

Lorsque des conflits sont détectés à la compilation, les messages d'erreur proviennent du YACC. Nous conseillons vivement au futur concepteur (ou au lecteur intéressé) de passer quelques heures à étudier YACC. Nous ne pouvons en effet pas nous étendre plus sur le fonctionnement de YACC car il y a matière pour un autre mémoire.

6.5.5. Lien entre syntaxe concrète SSL et YACC

Nous pouvons, très brièvement, présenter YACC comme un utilitaire qui prend une grammaire context-free comme entrée et produit un automate à pile qui réalise une analyse syntaxique par la méthode "shift/reduce".

La grammaire qu'il faut fournir à YACC est un ensemble de productions de grammaire context-free; à chacune d'elles est associée une action :

non-terminal : liste de symboles { action } ;

La *liste de symboles* est composée de symboles terminaux (tokens venant de Lex ou constantes 'caractère') et de symboles non-terminaux.
L'action est une suite d'instructions C qui est exécutée à chaque fois que l'analyseur syntaxique opère une réduction par la production qui y est associée.

YACC accepte des règles qui indiquent le niveau de priorité et l'associativité de constantes 'caractère' ou de symboles terminaux, dans le même genre que celles du SSL.

La partie action est utilisée par le compilateur SSL pour générer les instructions qui permettent d'attribuer l'arbre concret d'un programme ou d'un terme.

6.5.6. Eléments méthodologiques

Cette partie méthodologique comporte trois volets : le premier présente quelques conseils pour réaliser un analyseur "shift/reduce", le deuxième insiste sur la bonne utilisation du mécanisme d'attribution pour construire un arbre abstrait à partir de l'arbre concret, enfin quelques idées sur la récupération d'erreurs referment ce triptyque.

6.5.6.1. convention d'écriture

Nous avons convenu qu'un nom de phylum concret commence par la lettre "y", afin de rappeler qu'il s'agit d'un non-terminal utilisé par YACC, et est en lettres minuscules.

Les autres conventions d'écriture restent d'application.

6.5.6.2. Quelques conseils pour le shift/reduce parsing

6.5.6.2.1. Familiarisation avec YACC

Nous conseillons vivement à un futur concepteur de se familiariser avec YACC. Il est important de pouvoir lire ses fichiers de diagnostic; ceux-ci donnent une représentation lisible de l'automate à pile que YACC construit, et met en exergue les conflits qui peuvent survenir. Ces diagnostics sont très utiles pour des conflits peu évidents à résoudre. De

plus, YACC prend une décision par défaut à chaque conflit : il est intéressant de connaître cette décision et de voir si elle correspond au comportement que l'on désire obtenir.

6.5.6.2.2. Conflits shift/reduce

Soient les productions

s ::= (IF c THEN s) (r1)
| (IF c THEN s ELSE s) (r2)

Soit l'état de l'automate que voici :

pile	symboles d'entrée
$ IF c THEN IF c THEN s	ELSE ... $

faut-il 1) faire glisser ELSE sur la pile (shift) ?
2) ou bien faut-il réduire d'abord le sommet de pile par la règle r1 pour obtenir :

pile	symboles d'entrée
$ IF c THEN s	ELSE ...$

L'automate considérerait que la chaîne

IF c_1 THEN IF c_2 THEN s_1 ELSE ...

signifie dans le premier cas : "IF c_1 THEN { IF c_2 THEN s_1 ELSE ... } ...",
dans le second cas : "IF c_1 THEN { IF c_2 THEN s_1 } ELSE ...

Ce conflit résulte des deux interprétations possibles des productions de *s* : dans le premier cas la partie "ELSE" est associée au plus proche "IF"-sans-"ELSE" qui précède, tandis que dans le second cas la partie "ELSE" est rattachée au plus antérieur des "IF"-sans-"ELSE".

L'automate que YACC génère réalise un "shift", par défaut, dans ce genre de conflit. D'ailleurs, il est possible de tirer parti de ce choix à des fins d'optimisation :

Exemple A : soient les productions

data ::= (name op op) (r1)
op ::= (opA) (r2)
| (opB) (r3)
| ( ) /* vide */ (r4)

On associe une condition à *r1* qui exprime qu'une donnée doit avoir un nom et peut avoir zéro, une ou deux options **différentes**.
Ces productions conduisent à un conflit shift/reduce.
Si l'état de l'automate est :

	pile	symboles d'entrée
1	$...name	opA...$
2	$...name	opB...$

En *1* il y a un conflit de shift/reduce, car faut-il faire glisser *opA* sur la pile (shift) considérant qu'il s'agit de la première option ou faut-il procéder à une réduction par *r4* considérant que la première option est vide et que le symbole *opA* est la deuxième ?

En *2* il y a conflit de shift/reduce pour les mêmes raisons. L'action par défaut est, dans ce type de conflit, de faire glisser le symbole suivant sur la pile (shift). Le lecteur remarquera que c'est l'action la plus satisfaisante.

Les deux scénarios suivants montrent qu'après avoir fait glisser le symbole *opA* ou *opB* (*opX* désignera l'un ou l'autre de ces symboles), l'automate reconnaît toujours le non-terminal *data* .

Scénario 1 : il y a une deuxième option.

	pile	symboles d'entrée	action
1	$...name opX	opX ... $	réduire par r2 ou r3
2	$...name op	opX ... $	shift
3	$...name op opX	... $	réduire par r2 ou r3
4	$...name op op	... $	réduire par r1
5	$...data	... $	C.Q.F.D.

scénario 2 : il n'y a pas de deuxième option.

	pile	symboles d'entrée	action
1	$...name opX	... $	réduire par r2 ou r3
2	$...name op	... $	réduire par r4[1]
3	$...name op op	... $	réduire par r1
4	$...data	... $	C.Q.F.D.

Reconnaître que deux options sont identiques et que le non-terminal *data* est malformé n'incombe pas à l'analyseur lexical mais à la vérification de la sémantique statique que nous verrons plus loin.

On aurait pu réaliser la même chose qu'à l'exemple A, avec les productions de l'exemple B qui suit.

Exemple B : aucune condition de sémantique statique ne sera nécessaire car toutes les combinaisons possibles sont représentées.

```
data ::= (name )
      | (name opA )
      | (name opB )
      | (name opA opB )
      | (name opB opA )
      ;
```

L'inconvénient majeur réside dans le fait que le nombre de productions peut devenir énorme et l'analyseur syntaxique correspondant ... gigantesque.
Dans le cas de nombreuses combinaisons, la méthode présentée à l'exemple A peut devenir avantageuse afin de limiter la taille de l'analyseur syntaxique[2].

---

[1] C'est comme si on faisait glisser un symbole "vide" sur la pile et puis qu'on procédait à la réduction par r4.

[2] Encore faut-il que les vérifications de sémantique statique additionnelles ne provoquent pas de surcharge trop grande à l'exécution. Tout est question de dosage et du coût qu'on est prêt à payer, en taille et en surcharge, ou encore du rapport gain de place - surcharge induite.

6.5.6.2.3. *Conflits reduce/reduce*

Soient les productions :

```
stmt ::= (CALL name )               (r1)
   | (CALL name call_list );        (r2)
call_list ::= ( ) /* vide */        (r3)
   | (id call_list);                (r4)
```

Un conflit de réduction entre *r1* et *r3* surgit lorsque l'automate est dans l'état suivant :

<u>pile</u>	<u>symboles d'entrée</u>
$ CALL name	x...$

où x est un symbole qui ne dérive <u>pas</u> de *id*, et où aucun préfixe des symboles d'entrée ne dérive de *id*.

Ce type de conflit est résolu en se basant sur l'ordre des productions : c'est la production, qui apparaît le plus tôt dans la spécification, qui est choisie.

Les conflits reduce/reduce sont **<u>à éviter absolument</u>** car il est néfaste d'avoir des spécifications dans lesquelles l'ordre importe : la maintenance peut être rendue très difficile, on ne peut maîtriser le comportement du système généré que si l'ordre des spécifications est commenté et justifié. C'est tout à fait inacceptable pour de longues spécifications.

6.5.6.3. Pour une bonne utilisation du mécanisme d'attribution

Nous avons vu qu'une grammaire concrète doit être exprimée en SSL sous forme de productions de grammaire context-free. De même que les expressions régulières ne peuvent pas tout décrire, une grammaire context-free ne peut décrire que ce qui ne dépend pas du contexte d'un langage. La syntaxe abstraite a pour but de décrire, quant à elle, un langage avec son contenu sémantique.

Alors, comment passer d'un arbre de syntaxe concrète à un arbre de syntaxe abstraite ? Dans certains cas ce passage est évident voire immédiat. Dans d'autres cas, il faut utiliser toute la puissance du mécanisme d'attribution que met le CSG à notre disposition.

Le lecteur trouvera quelques réflexions sur l'art de programmer avec des grammaires attribuées après l'exemple qui suit : il illustre un passage moins facile d'un arbre concret à un arbre abstrait.

6.5.6.3.1. Illustration

En COBOL, une donnée peut être groupée ou élémentaire. Une donnée groupée est subdivisée en plusieurs données, élémentaires ou groupées; le numéro de niveau d'une donnée groupée est inférieur aux numéros de niveau de ses composantes.

On peut considérer chacune des sections (linkage, working-storage et file section) de la "data-division" comme des listes de données, élémentaires ou groupées. Une donnée groupée contient une liste de données composantes. Notre structure abstraite en SSL donne donc :

```
ld : Ld0()
   | Ld2( d  ld ) ;
d : De( t )            /* donnée élémentaire */
  | Dg( t ld ) ;       /* t désigne un n-uplet (level name redef opt opt opt opt ) */
```

Pour l'analyse syntaxique, nous considérons une liste de lignes de définition de donnée :

```
yld ::= ( ) | (yd yld);
yd  ::= ( ylevel yname yredef yopt yopt yopt yopt '.');
```

Nous ne pouvons, en effet, pas décrire une structure emboîtée comme dans la syntaxe abstraite car la fin d'une donnée groupée[1] est déterminée par le fait que le numéro de niveau de la prochaine ligne de définition de donnée à lire est inférieur au numéro de niveau de la ligne courante. C'est inexprimable avec une grammaire context-free.

En attribuant l'arbre concret nous pouvons tenir compte du contexte dans lequel se trouve chaque ligne de définition de donnée : l'attribut hérité **s** mémorise ce qui a déjà été lu, tandis que l'attribut synthétisé **a** représente la structure abstraite de toutes les lignes de définition de donnée.

La structure de **s** est une pile. Chaque élément de **s** est une liste de données de domaine syntaxique *ld* , qui satisfait aux propriétés suivantes :

p1) "Tout élément d'une liste est une donnée élémentaire ou groupée dont le numéro de niveau est identique à celui des autres éléments de la liste" (on considérera que le niveau "77" est identique au niveau "01").

p2) Si la pile contient plus d'une liste : soient L1 la liste du sommet de **s** et L2 la liste située sous le sommet. "Le dernier élément de L2 est une donnée groupée *d'* de structure *Dg(t' ld')* dont le corps *ld'* est vide, <u>et</u> tous les éléments de L1 sont des composantes de la liste *ld'* ". En d'autres termes, le dernier élément de chaque liste qui n'est pas au sommet de la pile, est (idéalement) toujours une donnée groupée dont le corps est vide, c'est-à-dire une donnée groupée incomplète.

De ces propriétés nous pouvons imaginer le scénario de transmission en héritage, d'une pile *s* tout au long d'un arbre concret, de la manière suivante :

---

[1] On peut considérer une donnée groupée COBOL comme une structure emboîtée.

Soient m : le numéro de niveau de la ligne de donnée courante *dc*,

n : le numéro de niveau commun aux éléments de la liste située au sommet de la pile *s* reçue en héritage (on parlera de la liste-sommet de *s*).

Trois cas sont possibles :

m = n, signifie que *dc* est "sur le même niveau" que les données de la liste-sommet de *s* . Alors on transmet la pile *s* après avoir ajouté la donnée *dc* à la fin de sa liste-sommet.

m > n, signifie que *dc* est la première composante de la donnée groupée située en fin de la liste-sommet. Alors on transmet *s* après y avoir empilé une liste dont *dc* est le seul élément.

m < n, signifie que la liste-sommet constitue le corps entier de la donnée groupée située en fin de la liste de "dessous-le-sommet". Alors, il faut "compléter" cette donnée groupée à l'aide de la liste-sommet qu'on dépile, et répéter ce processus jusqu'à ce que le niveau de la nouvelle liste-sommet soit égal à m. On peut ensuite ajouter *dc* en fin de la liste-sommet et transmettre la nouvelle pile obtenue.

Le lecteur trouvera dans le schéma qui suit un petit exemple qui montre les différentes étapes de construction d'une pile s et, à la fin, les deux étapes de complément de la pile s, représentées par les deux encadrés dans le dernier arbre.

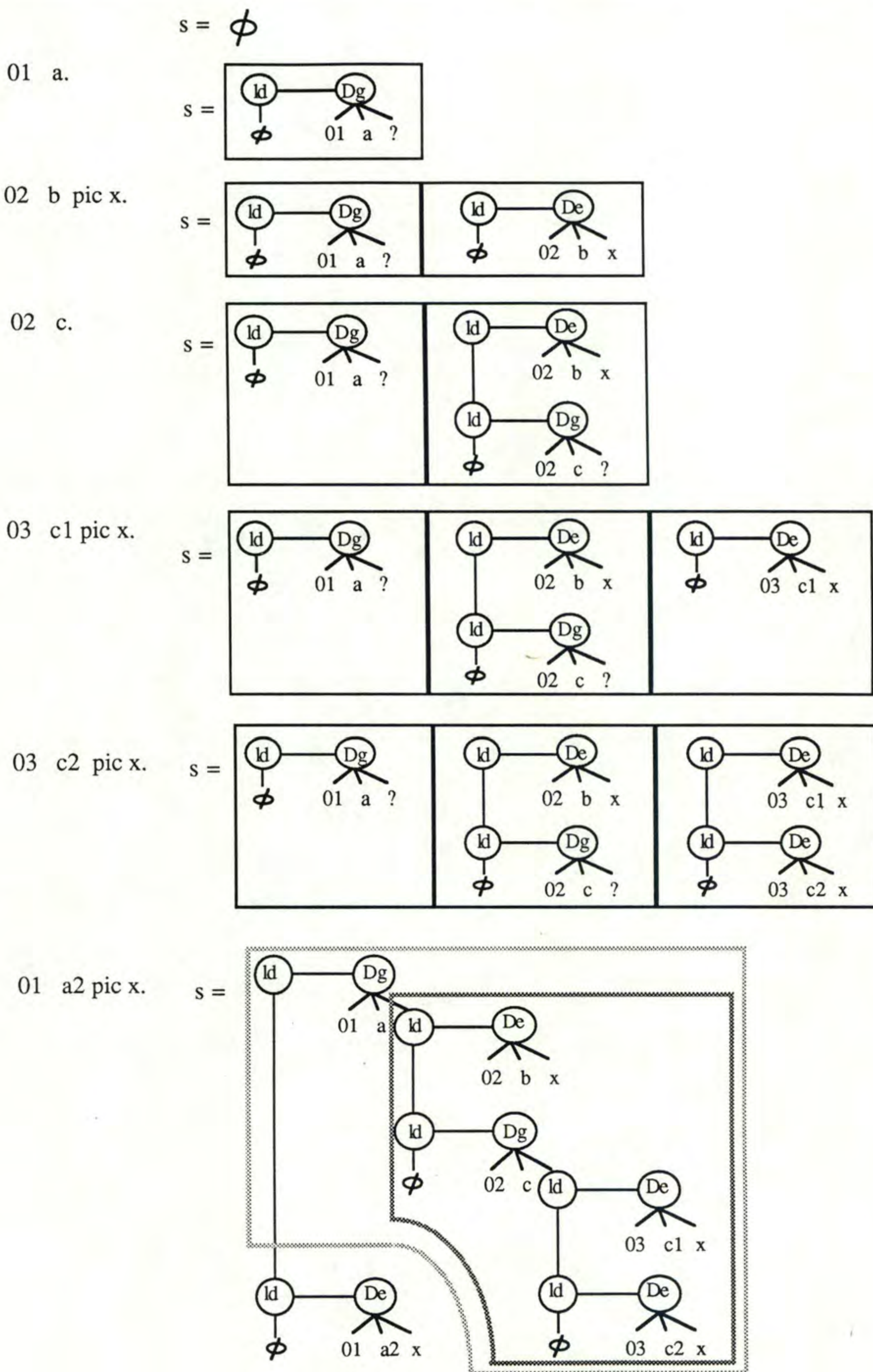


figure 4.5.6 : Evolution de la pile s

Après la dernière ligne de définition de donnée, et après complément de la pile *s*, le sommet contient la liste complète des définitions de données sous forme abstraite; il suffit alors de transmettre la liste-sommet de *s* comme attribut synthétisé via l'attribut *a* .

Cet exemple montre donc au lecteur que le mécanisme d'attribution permet de passer d'une structure concrète relativement simple à une structure abstraite beaucoup plus compliquée.

"Facile à dire" nous rétorquerez-vous. En effet, programmer en termes de grammaires attribuées remue beaucoup d'habitudes acquises avec un enseignement "traditionnel" de la programmation. Les paragraphes qui suivent essayeront de donner au lecteur quelques idées de réflexion.

6.5.6.3.2. "Programmer" en ... grammaires attribuées

Donner des spécifications de grammaire attribuée n'est jamais qu'une "autre" façon de programmer : un formalisme particulier, des règles propres, une philosophie qui sort des sentiers battus de la programmation procédurale.

Comme nous le disions au début, trouver des attributs, choisir leur mode de propagation (synthèse / héritage) et leur définir une équation sémantique peut être facile ... dans le cas de problèmes communs. L'exemple précédent a montré un cas moins facile : d'abord pour trouver les "bons" attributs et leurs propriétés, ensuite pour "programmer" tout cela en SSL.

En ce qui concerne la programmation, la difficulté résulte du fait qu'il faut penser autrement : une approche procédurale n'est pas de mise, une approche purement fonctionnelle non plus. Nous nous sommes forgés, petit à petit, un mode de pensée sous forme de questions qu'on se pose à chaque production de grammaire :
- pour définir les valeurs d'attributs hérités d'un noeud-fils, quelle contribution peut apporter le noeud courant aux valeurs d'attribut reçues en héritage du noeud-parent ?
- pour définir les valeurs d'attribut synthétisées du noeud courant, quelle contribution peut-il apporter aux valeurs d'attribut synthétisées par ses noeuds-fils ?

La "contribution" dont nous parlons dépend de *faits*, connus grâce aux valeurs d'attribut reçues en héritage du noeud-parent et grâce aux valeurs d'attribut synthétisées transmises par les noeuds-fils. Ces "faits connus" d'un noeud constituent son *contexte*.

6.5.6.4. La récupération d'erreurs

Lorsque l'analyseur syntaxique rencontre une suite de symboles qu'il n'arrive pas à "réduire" en start-symbol, il s'arrête d'analyser le texte d'entrée, place le curseur près du symbole qu'il ne peut pas reconnaître, affiche un laconique "syntax error" et provoque un

"bip" sonore : voilà une réaction à l'erreur par la panique !
Ce scénario se déroule uniquement lorsque c'est l'analyseur syntaxique qui détecte une erreur de grammaire "context-free".

Un autre mode de réaction aux erreurs est possible lorsqu'un texte, syntaxiquement bien-formé, est sémantiquement malformé : c'est la **récupération d'erreurs lors de la construction de l'arbre abstrait.** C'est évidemment au concepteur de prendre l'initiative d'une telle réaction : il étudiera les cas où un texte bien-formé syntaxiquement est inconsistant sémantiquement, afin de procéder à la récupération de *certaines* erreurs.

En général, lorsqu'on construit un arbre abstrait, c'est par attributs synthétisés : l'arbre abstrait d'un objet est construit en fonction des sous-arbres abstraits de ses composantes. Parfois une composante nécessaire manque : une récupération d'erreur pleine de bon sens consiste à considérer que le sous-arbre abstrait de cette composante est "non-développé".

Parfois il s'agit d'un choix audacieux qui trahit le sens de ce que l'auteur d'un programme voulait écrire. Souvent cela montre à l'auteur de programme qu'il a "oublié" un élément. Il peut donc rapidement développer celui-ci.

Exemple : Dans notre éditeur syntaxique COBOL
si l'auteur écrit :

```
01 toto.
   03 a pic 9(8).
   03 b pic 9(2).
   02 d pic x.
```

L'éditeur impose la présence des données de niveau 03 à l'intérieur d'une donnée de niveau 02, à cause de la présence, dans le texte, d'une donnée de niveau 02.
L'éditeur "corrige" en :

```
01 toto.
   02 <data-name>.
      03 a pic 9(8).
      03 b pic 9(2).
   02 d pic x.
```

Si l'auteur écrit :

```
procedure division.
   move a to b.
   ...
pa.
   ...
s section.
   accept b.
...
```

L'éditeur impose que des instructions se trouvent à l'intérieur de paragraphes, et de sections s'il en apparaît au moins une dans le texte d'entrée.
L'éditeur corrige en :

```
procedure division.
<section-name> section.
<paragraph-name>.
  move a to b.
  ...
pa.
  ...
s section.
<paragraph-name>.
  accept b.
...
```

Ce genre de réaction aux erreurs, même si la "correction" ne répond pas aux attentes de l'utilisateur, a l'avantage d'être moins frustrante et plus "amicale" (user-friendly) qu'un "syntax error" suivi d'un arrêt immédiat des opérations. Une correction peut également s'assortir d'un message préventif (warning).

Mieux qu'un message d'erreur, ce type de correction automatique montre à l'utilisateur comment il aurait pu écrire son programme pour qu'il soit bien-formé. Un correcteur automatique ne sait pas prévoir ce que l'utilisateur a voulu exprimer. Il donne une *idée* de correction tout en montrant qu'il y a eu erreur ... N'est-ce pas un peu mieux que certains messages d'erreur sibyllins provenant de compilateurs peu éloquents ?

6.6. Déclarations de transformation

Une déclaration de transformation spécifie une commande d'édition pour restructurer le contenu de la sélection lorsque ce contenu coïncide à un modèle structurel donné.

Après avoir présenté la façon dont les transformations sont spécifiées, nous montrerons comment utiliser celles-ci pour l'édition par gabarits et pour la restructuration de parties de programme.

6.6.1. Spécification de transformations

On spécifie une "transformation" comme suit :

transform ***phylum*** **on** ***transformation_name*** ***pattern*** **:** ***expression*** **;**

phylum est un phylum de la syntaxe abstraite,
transformation_name est une constante STR (une chaîne de caractères entre guillemets),
pattern est un modèle structurel, comme ceux qui peuvent apparaître dans une expression "with"
expression doit produire un résultat qui appartient au domaine syntaxique *phylum* .

On peut spécifier plusieurs transformations pour un même phylum comme suit :

transform ***phylum***
  on ***transformation_name*** ***pattern*** **:** ***expression*** **,**
  ...
  on ***transformation_name*** ***pattern*** **:** ***expression*** **,**

Rappelons-nous qu'un écran est divisé en quatre parties : la ligne de titre, la ligne de commande, la partie "objets" et la partie "aide". Cette dernière partie, outre le nom du phylum auquel appartient le terme de la sélection courante, affiche les noms des différentes transformations permises sur la sélection.

Une transformation, pour un phylum *p* , dont le modèle structurel est *m* , est active (ou encore permise) lorsque le contenu de la sélection est un terme, appartenant à *p*, qui coïncide avec *m* .

Seules les transformations permises sur la sélection sont affichées dans la partie "aide" de l'écran.

6.6.2. Edition structurelle avec gabarits

Les transformations les plus évidentes sont celles qui substituent un terme non-développé par un gabarit donné. Un gabarit est un terme dont les composantes sont non-développées.

Exemple : Soient les productions
expression : ExpNull() [@::= "<expression>"]
...
| ConditionalExp(expression expression expression)
[@ ::= "if" @ "then" @]
...
;
Soit la transformation
transform expression on "if" <expression>[1] : ConditionalExp ([expression], [expression], [expression]);

La transformation "if" permet de remplacer le placeholder term du phylum *expression* par le gabarit de l'expression conditionnelle : la sélection contenant le terme ExpNull(), est modifiée en : ConditionalExp(ExpNull(), ExpNull(), ExpNull()).
Visuellement,

<expression>

devient

if <expression> then <expression> else <expression>.

Donc, à partir de noms de commandes simples, il est possible de faire apparaître des gabarits complexes qui remplacent un terme sélectionné non-développé.

6.6.3. Restructurations complexes

Comme les transformations utilisent les possibilités du pattern-matching, on peut les utiliser pour réaliser des effets plus complexes sur un programme.

Exemple : la transformation suivante factorise une somme de produits
transform expression on "factoriser" Multiply(Plus(a, b), Plus(a, c)) : Multiply(a, Plus(b, c));

On pourrait imaginer une transformation qui remplace une procédure, avec appel récursif comme dernière instruction, en une procédure itérative. D'autres manipulations sont possibles, telle celle présentée dans l'exemple qui suit :

[1] Le CSG propose deux notations abréviées où
[phylum] désigne le completing term d'un phylum, et
<phylum> désigne le placeholder term d'un phylum.

Exemple : la transformation suivante remplace un "repeat" PASCAL par un "while"

```
transform statement on "rep-while"  RepeatStmt(statement, cond) :
                                    WhileStmt(Complement(cond), statement);
```

où Complement : condition ---> condition est une fonction qui prend pour argument un terme du phylum condition et a pour résultat le "complément" de ce terme.

```
condition Complement(condition c) (
        with(c) (
            Not(a) : a,
            GreaterThan(a, b) : LessOrEqual(a, b),
            Equal(a, b) : NotEqual(a, b),
            NotEqual(a, b) : Equal(a, b),
            LessThan(a, b) : GreaterOrEqual(a, b),
            GreaterOrEqual(a, b) : LessThan(a, b),
            LessOrEqual(a, b) : GreaterThan(a, b),
            And(a, b) : Or(Complement(a), Complement(b)),
            Or(a, b) : And(Complement(a), Complement(b))
        )
    );
```

Sans avoir présenté de long discours, nous espérons que le lecteur sera néanmoins convaincu que le mécanisme des transformations du CSG offre un outil puissant de manipulation d'arbres abstraits et permet au concepteur, avec simplicité, de se construire un jeu de commandes d'édition très puissant.

6.7. Vérification de la sémantique d'un programme

Un aspect fort intéressant d'un éditeur syntaxique, qui maintient un arbre attribué comme représentation de programme, est qu'il peut apporter une réaction immédiate aux erreurs que l'utilisateur commet. [Reps 1984] affirme que l'analyse incrémentale de la sémantique d'un programme permet de détecter trois sortes d'erreurs, autres que les erreurs de syntaxe :

1) les erreurs de sémantique statique,
2) certaines "anomalies",
3) l'incorrection d'un programme par rapport à sa spécification.

6.7.1. Vérification de la sémantique statique

Le mécanisme d'attribution permet d'apporter une réaction immédiate aux violations de certaines contraintes d'un langage telles que :

- identificateur non déclaré,
- types incompatibles,
- etc, ...

Afin réaliser de telles vérifications, il est nécessaire de connaître le contexte général d'un programme : par la transmission d'un attribut hérité qui contient la table des symboles déjà déclarés, et la vérification, dans une instruction, si un identificateur est connu ou non, si le type de donnée associé à un identificateur est valide, etc ...

Pour avertir l'utilisateur de telles violations de contraintes sémantiques, la solution la plus simple est de donner une réaction visuelle (voire sonore) via un attribut dont la valeur intervient dans un schéma de décompilation.

Exemple : L'exemple suivant montre comment il est possible d'imposer à des éléments optionnels d'intervenir une seule fois seulement dans la définition d'une donnée en COBOL.

```
d : De( level name opt opt opt opt )
  | Dg( level name opt opt opt opt ld);
opt : OptNull()
    | Pic( picstring )
    | Value( STR )
    | Usage( u )
    | Occurs( INT );
opt { inherited test check_in; /* attribut hérité */
      synthesizd test check_out; /* attribut synthétisé */
    };
test : T(     BOOL /* Pic déclaré */
              BOOL /* Value déclaré */
              BOOL /* Usage déclaré */
              BOOL /* Occurs déclaré */
       );
/* check_in, attribut hérité, décrit les clauses optionnelles déjà déclarées dans d, tandis que
   check_out, attribut synthétisé, reprend la valeur de check_in et y ajoute "true" à l'endroit ad
hoc pour signifier ce qui vient d'être déclaré.
Les équations sémantiques qui suivent montrent comment il est possible de vérifier l'unicité de
chaque clause et prévenir l'utilisateur en cas d'erreur */
d : De {      opt$1.check_in = T(false, false, false, false); /* au début, il n'y a encore rien de
                                                                    de déclaré */
              opt$2.check_in = opt$1.check_out;
              opt$3.check_in = opt$3.check_out;
              opt$4.check_in = opt$3.check_out;
       }
  | Dg {      équations identiques à celles de l'opérateur "De" ...  };
```

```
opt :  OptNull [@ ::= "<option-clause>" ]
            { $$.check_out = $$.check_in ; }
   | Pic [@ ::= "PICTURE IS " @ error ]
            { local STR error;
              error = with($$.check_in) (
                              T(true, *, *, *) : "<--- already declared ",    /* erreur */
                              default : ""
                        );
              $$.check_out = with($$.check_in) (
                                    T(*, a, b, c) : T(true, a, b, c)
                              );
            }
   | ... etc ...
```

Voici le schéma de transmission des attributs check_in et check_out

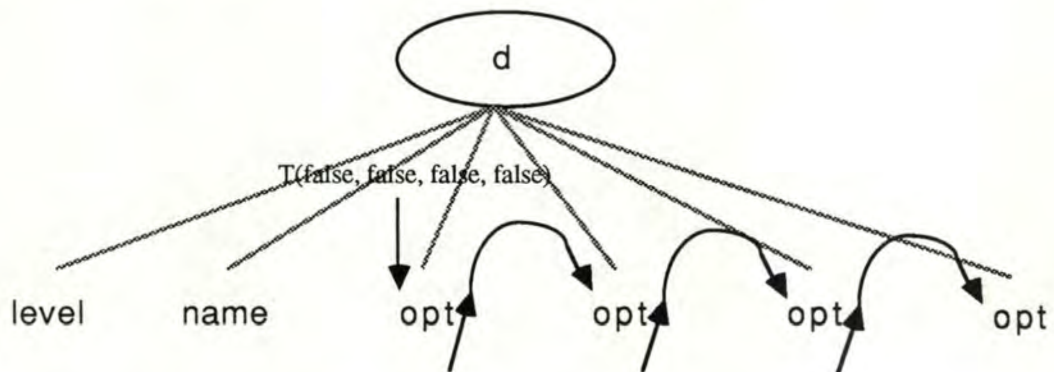


figure 4.7.1 : Schéma de transmission des attributs synthétisés et hérités du phylum *opt*

Etant donné que les attributs qui permettent de réaliser de telles vérifications expriment des contraintes sémantiques du langage, il est nécessaire de porter un soin tout à fait particulier lors de la définition des équations sémantiques d'attribut de productions complétives : elles doivent permettre de donner un sens aux attributs de termes non-développés.

6.7.2. Détection d'anomalies

Il est possible de pousser plus loin l'analyse d'un programme en étudiant le comportement possible du programme à l'exécution, sans toutefois l'exécuter. Détecter certaines anomalies peut être un processus non-décidable lorsque celles-ci dépendent de données dont le programme dispose à l'exécution seulement. Néanmoins, certaines vérifications peuvent être facilement réalisées afin de mettre en évidence une incohérence ou un oubli :

- variable utilisée sans avoir été initialisée.
- condition toujours vraie ou toujours fausse.
- division par zéro.

- partie de programme jamais exécutée parce qu'elle ne peut être atteinte,
- etc, ...

6.7.3. Vérification de la correction

Il est possible d'ajouter à un programme la spécification formelle de son comportement et de vérifier si le programme correspond bien à cette spécification. Nous *évoquons* uniquement cette possibilité car elle ne peut être réalisée que de manière limitée à cause de la complexité exponentielle de certaines vérifications.

Des éditeurs syntaxiques avec "vérificateur" incorporé existent : le CSG, par exemple, est livré avec un éditeur d'un langage de commandes gardées de Dijkstra, avec instruction d'assignations multiples, tableaux et variables entières. Des conditions de vérification sont générées et "tentative" (sic) est faite de les prouver automatiquement.

6.7.4. Conclusion - Performances

L'analyse incrémentale disponible avec des éditeurs générés par le CSG rend possible un grand nombre de vérifications. L'intérêt est de pouvoir se tailler un éditeur syntaxique "sur mesure" avec des vérifications plus ou moins sophistiquées.

Il **faudra trouver un compromis** entre une détection d'erreurs pertinente et complète, et une surcharge occasionnée par de tels mécanismes.

Maintenir un grand nombre d'attributs se paye en espace mémoire et en temps calcul. Bien que le coût de la mémoire diminue et que la puissance des processeurs augmente, il faut rester les pieds sur terre : une édition syntaxique avec toutes les vérifications possibles et imaginables est-elle vraiment réaliste si elle nécessite un CRAY pour donner un temps de réponse acceptable à l'utilisateur ?

Un éditeur syntaxique évite à l'utilisateur de devoir répéter n fois le cycle

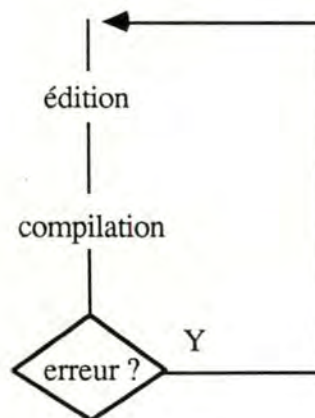


Figure 4.7.2 : Cycle de mise au point syntaxique

de "mise au point syntaxique" d'un programme. Nous pensons donc que le coût d'une édition syntaxique est acceptable s'il est inférieur ou égal au coût de n invocations du cycle précité.

L'avantage d'un éditeur syntaxique pourrait être que la charge demandée au système est répartie plus uniformément dans le temps, alors qu'un cycle édition - compilation (vérifications sémantico-syntaxique) utilise les ressources de façon beaucoup plus inégale.

Si vraiment l'utilisation d'éditeurs syntaxiques augmente la productivité des programmeurs, le rapport entre efficacité - coût - et satisfaction de l'utilisateur peut devenir avantageux : dans ce cas il est temps de sortir de l'ère du développement de programmes selon le "paradigme de la boîte à outils" pour entrer dans celle des environnements de programmation interactifs.

L'édition structurelle, avec gabarits uniquement, devient énervante lorsqu'on est familier avec le langage de programmation utilisé, sauf dans le cas de langages fort verbeux où elle économise un temps important.

L'édition textuelle - structurelle semble idéale car elle permet à l'utilisateur d'introduire textuellement les parties de programme simples et de développer par gabarits les parties fastidieuses à écrire.

Seconde partie :

Conception et réalisation d'un Compilateur Incrémental avec le CSG.

Jacques de Wergifosse

7. Conception du générateur de code C

7.1. Introduction

Ce chapitre présente la conception de notre générateur de code C. Il décrit les règles de traduction que nous avons utilisées afin d'effectuer le passage d'un programme COBOL à un programme C compilable et exécutable.

Ces règles sont exprimées sous la forme de fonctions associées aux symboles de la grammaire du sous-ensemble COBOL conservé reprise dans la section 3 de ce chapitre sous la forme d'une grammaire concrète BNF et représentent les attributs permettant la génération de code C.

Afin de rendre celles-ci lisibles, nous avons choisi une notation qui permet à la fois de se détacher du langage d'implémentation (le SSL dans notre cas) et de vérifier aisément le caractère correct de ces règles. La section 2 en énoncera les conventions.

La structure globale du programme C généré sera ensuite exposée dans la section 4.

La section 5 présentera la manière dont les données de la data division et les fichiers COBOL sont mémorisés et référencés dans le texte du programme C généré. Elle s'attardera aussi à décrire les environnements de compilation nécessaires afin de représenter les diverses instructions COBOL exposées dans la section 7.

La section 6 donnera une classification des fonctions C utilisées afin d'exécuter le programme C généré, ainsi que leurs spécifications.

La section 7 développera la manière de représenter la procedure division d'un programme COBOL, ainsi que les instructions COBOL.

7.2. Conventions

7.2.1. Grammaire BNF

Les symboles terminaux sont imprimés en **gras**.

Les symboles non-terminaux apparaissent entre "<" et ">".

"::=" sépare les parties gauche et droite d'une règle de production.

"|" sépare deux choix possibles.

Tout ce qui apparaît entre "[" et "]" est optionnel (0-1).

Le symbole **nil** dénote une suite de caractères vide.

7.2.2. Règles de traduction

Nos règles de traduction seront exprimées sous la forme de fonctions associées aux symboles de la grammaire abstraite correspondant à la grammaire concrète BNF exposée dans la section 3. Ces fonctions représentent les attributs associés aux symboles de la grammaire COBOL permettant de réaliser la génération de code C.

Toute règle de traduction aura la forme suivante :

nom_de_fonction (< symbole >) = valeur.

La **valeur** d'une fonction sera exprimée avec des caractères **gras**, des fonctions associées à d'autres symboles, des symboles, d'autres fonctions associées au symbole **symbole** ou la valeur **nil**. Elle définira l'équation sémantique associée à l'attribut dont elle détermine la valeur.

Les fonctions figurant dans cette **valeur** ne peuvent être définies avec la fonction dont elles déterminent la **valeur** afin de respecter la contrainte de non-circularité dans le graphe de dépendance des attributs.

Le caractère "|" sépare des choix possibles dans la définition de la **valeur** d'une fonction. Ces choix dépendront du contexte syntaxique du symbole auquel cette fonction est associée.

De plus, lorsqu'une fonction définit la valeur d'une fonction associée à un symbole **symbole** et que le symbole auquel elle est associée ne figure pas dans la règle de production de **symbole**, nous considérerons que l'attribut qu'elle représente a été hérité et/ou synthétisé de telle manière que sa valeur est disponible.

Une règle est à interpréter de la manière suivante :

1° Les caractères gras et la valeur des fonctions figurant dans la **valeur** d'une fonction doivent apparaître dans le texte du programme C généré,

2° de même que les symboles, dont il faut considérer le symbole terminal associé.

3° La valeur **nil** dénote une suite de caractères vide.

De plus, afin de faciliter la lecture des règles, lorsqu'une fonction dénote une valeur numérique et qu'elle figure dans la définition d'une fonction dans laquelle apparaissent des caractères **gras**, nous considérerons que celle-ci a été convertie en une chaîne de caractères.

Lorsqu'une règle de production contient plusieurs occurrences d'un symbole < symbole >, les différentes occurrences de < symbole > seront dénotées (de gauche à droite) par < symbole >$_1$, < symbole >$_2$, etc., afin de pouvoir spécifier leurs fonctions respectives.

Une fonction pourra également prendre la valeur d'une donnée appartenant à un environnement de compilation. C'est pourquoi, afin de référencer une de ces données et de faciliter la lecture des règles de traduction, les conventions suivantes ont été prises :

Tout environnement de compilation, implémenté en SSL par un phylum liste, sera considéré ici comme une table au sens de l'algèbre relationnelle, une table possédant un nom et étant composée d'un ensemble de lignes et colonnes, une ligne contenant une ou plusieurs données élémentaires et chaque colonne ayant un nom associé aux données élémentaires d'une ligne.

Exemple :

Soit la table de nom **PERSONNES** composée des lignes et colonnes suivantes :

nom	**prénom**	**âge**
Dupont	Alain	18
Lejeune	Louis	45

Une fonction associée à un symbole pourra prendre la valeur d'une donnée appartenant à un environnement de compilation grâce à la notation suivante :

nom_environnement [**colonne$_1$** = **valeur** , **colonne$_2$**],

exprimant dans l'environnement de nom **nom_environnement** la valeur de la donnée appartenant à la colonne de nom **colonne$_2$** de la ligne dont la valeur de la donnée correspondant à la colonne de nom **colonne$_1$** est égale à **valeur**.

Ainsi en se basant sur l'exemple pré-cité, afin de connaître l'âge de la personne dont le nom de famille est "Dupont", la notation utilisée sera : PERSONNES [nom = "Dupont", âge], la valeur renvoyée étant 18.

7.3. Grammaire BNF concrète du subset COBOL

< cob_pgm > ::= < id_div >
< env_div >
< data_div >
< proc_div >

< id_div > ::= **IDENTIFICATION DIVISION.**
PROGRAM-ID. < program_name >.

< program_name > ::= < userdef_name >

< env_div > ::= nil
| **ENVIRONMENT-DIVISION.**
< configuration_sec >
< inputoutput_sec >

< configuration_sec > ::= nil
| **CONFIGURATION-SECTION.**
< src_computer >
< obj_computer >
< spec_names >

< src_computer > ::= nil
| **SOURCE COMPUTER.** [< computer_name >].

< obj_computer > ::= nil
| **OBJECT COMPUTER.** [< computer_name >].

< computer_name > ::= < userdef_name >

< spec_names > ::= nil
| **SPECIAL-NAMES.**
| **SPECIAL-NAMES. DECIMAL-POINT IS COMMA.**

< inputoutput_sec > ::= nil
| **INPUT-OUTPUT SECTION.**
FILE-CONTROL.
< filectrlentrylist >

< filectrlentrylist> ::= nil
| < selectclause >
< filectrlentrylist >

< selectclause > ::= nil
| **SELECT** < file_name > **ASSIGN TO** < assignmentlist >
ORGANIZATION IS SEQUENTIAL
ACCESS MODE IS SEQUENTIAL
< fstat > **.**

< file_name > ::= < userdef_name >

< assignmentlist > ::= < assignment_item >**,** < assignmentlist>
| < assignment_item >

< assignment_item > ::= **"**< str >**"**
| **"stdout"**
| **"stdin"**

< fstat > ::= nil
| **FILE STATUS IS** < data_name >

< data_div > ::= nil
| **DATA DIVISION.**
< file_sec >
< ws_sec >
< lk_sec >

< file_sec > ::= nil
| **FILE SECTION.**
< fdlist >

< fdlist > ::= nil
| < fditem >
< fdlist >

< fditem > ::= nil
| < fdentry >
< des_list >

< fdentry > ::= **FD** < file_name >**.**

< des_list > ::= nil
| < recdes_entry >
< des_list >

< ws_sec > ::= nil
| **WORKING-STORAGE SECTION.**
< des_list >

< lk_sec > ::= nil
| **LINKAGE SECTION.**
< des_list >

< recdes_entry > ::= nil
| < recdes_rec >
| < recdes_data >

< recdes_rec > ::= < levelnr > < fill_or_data > < redefclause >
< valueclause > < occursclause >
< usageclause > < record_body > **.**

< recdes_data > ::= < levelnr > < fill_or_data > < redefclause >
< valueclause > < occursclause >
< usageclause > < pictureclause >**.**

< record_body > ::= nil
| < recdes_entry >
< record_body >

< fill_or_data > ::= **FILLER**
| < data_name >

< data_name > ::= < userdef_name >

< levelnr > ::= < level_number >

< redefclause > ::= nil
| **REDEFINES** < data_name >

< usageclause > ::= nil
| **USAGE IS COMPUTATIONAL**
| **USAGE IS DISPLAY**

< occursclause > ::= nil
| **OCCURS** < integer > **TIMES**

< valueclause > ::= nil
| **VALUE IS** < numeric_literal >
| **VALUE IS** < alphanum_literal >
| **VALUE IS** < fig_constant >

< numeric_literal > ::= < str >

< alphanum_literal > ::= < str >

```
< fig_constant > ::= ZERO
                   | SPACE
                   | HIGH-VALUE
                   | LOW-VALUE
                   | QUOTE

< pictureclause > ::= nil
                   | PICTURE IS < picstring >

< picstring > ::=       < numericint_format >
                   | < numericdec_format >
                   | < alphanumeric_format >

< numericint_format > ::=     S< integer_partie1>
                   |          < integer_partie1>
                   |          S< integer_partie1 >< power_partie>
                   |          < integer_partie1>< power_partie >

< numericdec_format > ::=     S< integer_partie1 >V< decimal_partie2 >
                   |          SV< decimal_partie2 >
                   |          SV< power_partie >< decimal_partie2 >
                   |          S< power_partie >< decimal_partie2 >
                   |          < integer_partie1 >V< decimal_partie2 >
                   |          V< decimal_partie2 >
                   |          V< power_partie >< decimal_partie2 >
                   |          < power_partie >< decimal_partie2 >

< alphanumeric_format > ::= X(< integer >)
                   | X< reptX >

< reptX > ::=           nil
                   | X< reptX >

< integer_partie1 > ::= 9< rept9 >
                   | 9(< integer >)


< decimal_partie2 > ::= 9< rept9 >
                   | 9(< integer >)

< rept9 > ::=           9< rept9 >
                   | nil

< power_partie > ::= P(< integer >)
                   | P< reptP >
```

< reptP > ::= **P**< reptP >
| nil

< proc_div > ::= nil
| **PROCEDURE DIVISION.**
< procdivbody >
| **PROCEDURE DIVISION USING** < parameterlist>.
< procdivbody >

< parameterlist > ::= < identifier > **,** < parameterlist >
| < identifier >

< procdivbody > ::= nil
| < sectionlist >
| < paragraphlist >

< sectionlist > ::= nil
| < section >
< sectionlist >

< paragraphlist > ::= nil
| < paragraph >
< paragraphlist >

< section > ::= nil
| < section_name > **SECTION.**
< paragraphlist >

< paragraph > ::= nil
| < paragraph_name > **.**
< sentencelist >

< section_name > ::= < procedure_name >

< paragraph_name > ::= < procedure_name >

< procedure_name > ::= < str >

< sentencelist > ::= nil
| < stmtlist >**.**
< sentencelist >

< stmtlist > ::= nil
| < stmt >
| < stmt > **,** < stmtlist >

< stmt > ::= nil
| < IfStmt >
| < ADDStmt >
| < SUBTRACTStmt >
| < MULTIPLYStmt >
| < DIVIDEStmt >
| < ACCEPTStmt >
| < DISPLAYStmt >
| < CLOSEStmt >
| < OPENStmt >
| < READStmt >
| < WRITEStmt >
| < CALLStmt >
| < EXITStmt >
| < EXITPGMStmt >
| < STOPRUNStmt >
| < PERFORMStmt >
| < MOVEStmt >

< IfStmt > ::= **IF** < condition > **THEN** < thenstmt > **ELSE** < elsestmt >[**END-IF**]
| **IF** < condition > **THEN** < thenstmt > [**END-IF**]

< thenstmt > ::= < stmtlist >
| **NEXT SENTENCE**

< elsestmt > ::= < stmtlist >
| **NEXT SENTENCE**

< ADDStmt > ::= **ADD** < oplist1 > **TO** < result_list >
< size_errorstmt > < notsize_errorstmt >
[**END-ADD**]
| **ADD** < oplist1 > **TO** < id_or_litnum > **GIVING** < result_list >
< size_errorstmt >
< notsize_errorstmt > [**END-ADD**]

< SUBTRACTStmt > ::= **SUBTRACT** < oplist1 > **FROM** < result_list >
< size_errorstmt > < notsize_errorstmt >
[**END-SUBTRACT**]
| **SUBTRACT** < oplist1 > **FROM** < id_or_litnum >
GIVING < result_list > < size_errorstmt >
< notsize_errorstmt > [**END-SUBTRACT**]

< MULTIPLYStmt > ::= **MULTIPLY** < id_or_litnum > **BY** < result_list >
< size_errorstmt > < notsize_errorstmt >
[**END-MULTIPLY**]

| **MULTIPLY** < id_or_litnum > **BY** < id_or_litnum >
GIVING < result_list > < size_errorstmt >
< notsize_errorstmt > **[END-MULTIPLY]**

< DIVIDEStmt > ::= **DIVIDE** < id_or_litnum > **BY** < id_or_litnum >
GIVING < result_list > < size_errorstmt >
< notsize_errorstmt > **[END-DIVIDE]**

< ACCEPTStmt > ::= **ACCEPT** < identifier >

< DISPLAYStmt > ::= **DISPLAY** < id_or_lit_list >

< CLOSEStmt > ::= **CLOSE** < filename_list >

< OPENStmt > ::= **OPEN** < openlist >

< READStmt > ::= **READ** < file_name > < into_id >
< atend_stmt > < notatend_stmt >
[END-READ]

< WRITEStmt > ::= **WRITE** < data_name > < from_id > **[END-WRITE]**

< CALLStmt > ::= **CALL** < alphanum_literal > **USING** < call_list >
| **CALL** < alphanum_literal >

< EXITStmt > ::= **EXIT**

< EXITPGMStmt > ::= **EXIT PROGRAM**

< STOPRUNStmt > ::= **STOP RUN**

< PERFORMStmt > ::= **PERFORM** < procedure_name > **[END-PERFORM]**
| **PERFORM** < stmtlist > **[END-PERFORM]**
| **PERFORM** < procedure_name > < id_or_int > **TIMES**
[END-PERFORM]
| **PERFORM** < id_or_int > **TIMES** < stmtlist >
[END-PERFORM]
| **PERFORM** < procedure_name > **UNTIL** < condition >
[END-PERFORM]
| **PERFORM UNTIL** < condition > < stmtlist >
[END-PERFORM]

< MOVEStmt > ::= **MOVE** < id_or_lit > **TO** < id_list >

< oplist1 > ::= < id_or_litnum >
| < id_or_litnum > **,** < oplist1 >

< id_or_litnum > ::= < identifier >
| < numeric_literal >

< result_list > ::= < result >
| < result > **,** < result_list >

< result > ::= < identifier >
| <identifier > **ROUNDED**

< size_errorstmt > ::= nil
| **ON SIZE ERROR** < stmtlist >

< notsize_errorstmt > ::= nil
| **NOT ON SIZE ERROR** < stmtlist >

< id_or_lit_list > ::= < id_or_lit >
| < id_or_lit > **,** < id_or_lit_list >

< id_or_lit > ::= < identifier >
| < alphanum_literal >
| < numeric_literal >
| < fig_constant >

< filename_list > ::= < file_name >
| < file_name > **,** < filename_list >

< openlist > ::= < openitem >
| < openitem > **,** < openlist >

< openitem > ::= **INPUT** < filename_list >
| **OUTPUT** < filename_list >

< into_id > ::= nil
| **INTO** < identifier >

< atend_stmt > ::= nil
| **AT END** < stmtlist >

< notatend_stmt > ::= nil
| **NOT AT END** < stmtlist >

< from_id > ::= nil
| **FROM** < identifier >

< call_list > ::= < id_list >

< id_list > ::= < identifier >
| < identifier > **,** < id_list >

< id_or_int > ::= < integer >
| < identifier >

< identifier > ::= < userdef_name >
| < userdef_name > **(** < subscript > **)**
| < userdef_name > **(** < subscript > **,** < subscript > **)**
| < userdef_name > **(** < subscript > **,** < subscript > **,**< subscript > **)**

< subscript > ::= < integer >
| < data_name >
| < data_name > **+** < integer >
| < data_name > **-** < integer >

< condition > ::= < id_or_lit > **>** < id_or_lit >
| < id_or_lit > **NOT >** < id_or_lit >
| < id_or_lit > **<** < id_or_lit >
| < id_or_lit > **NOT <** < id_or_lit >
| < id_or_lit > **=** < id_or_lit >
| < id_or_lit > **NOT =** < id_or_lit >
| < id_or_lit > **>=** < id_or_lit >
| < id_or_lit > **<=** < id_or_lit >
| < condition > **AND** < condition >
| **NOT** < condition >
| < identifier > **IS NUMERIC**
| < identifier > **IS NON NUMERIC**
| < identifier > **IS ALPHABETIC**
| < identifier > **IS NOT ALPHABETIC**

Remarque : Nous ne reprenons pas ici la syntaxe des symboles <userdef_name>, <integer> , < level_number > et < str >. Celle-ci est décrite en farde annexe avec des expressions régulières.

7.4. Structure globale du programme C généré

Le principe que nous utilisons consiste à représenter un programme COBOL, ainsi que les sections et paragraphes qui le composent, par des fonctions C. La génération de celles-ci sera réalisée par les fonctions **progr_c**, **repr_c_of_sections** et **repr_c_of_paragraphs** (cfr 7.7.1).

La fonction **progr_c** (cfr 7.4.1) représentera la structure du code C généré pour un programme COBOL particulier et diffère, comme nous le verrons ci-après, selon qu'il s'agit

d'un programme principal ou d'un programme pouvant être appelé par une instruction **CALL** que nous appelerons sous-programme.

Le corps de la fonction C représentant un programme contiendra dans la fonction **repr_c_of_proc_div** (cfr 7.7.1) l'appel des fonctions C représentant les sections ou paragraphes qui le composent et permettra ainsi de guider l'exécution de ce programme.

De plus, à toute fonction C représentant un programme COBOL, nous associerons une zone mémoire utilisée afin de mémoriser les données de la data division de celui-ci. L'instruction d'allocation de cette zone se fera dans la fonction **malloc_instruction** (cfr 7.5.2) et l'adresse de cette zone sera toujours mémorisée dans une variable que nous appelons **ptralloc.** La valeur de celle-ci sera toujours transférée aux fonctions C représentant les sections ou paragraphes d'un programme COBOL afin de toujours pouvoir calculer l'adresse de ses données (cfr 7.5.4).

Tout fichier COBOL sera représenté par un fichier C dont le nom interne est déclaré dans la fonction **decl_of_files** (cfr 7.5.5).

Les clauses **VALUE** des données de la working-storage section d'un programme COBOL nécessiteront la génération de variables C déclarées dans la fonction **decl_of_var_for_value_clauses** et seront représentées dans la fonction **repr_c_of_value_clauses** (cfr 7.7.11.1).

Une variable de type *int* appelée **size_error** sera déclarée dans la fonction **progr_c** du programme principal afin de pouvoir réaliser les clauses **SIZE ERROR** et **NOT SIZE ERROR** (cfr 7.7.5.5) des instructions arithmétiques. Il en sera de même pour les variables **highvalue** et **lowvalue** utilisées dans la représentation de l'instruction **DISPLAY** (7.7.6.2).

De plus, étant donné qu'un programme C commence toujours par une fonction **main**, la restriction a été faite à l'utilisateur de notre éditeur COBOL de donner le nom "**main**" à son programme principal afin de toujours spécifier dans le programme C généré cette fonction obligatoire. Pour tout sous-programme, les fonctions **decl_of_ptrs_for_param1**, **decl_of_ptrs_for_param2**, **progr_name_header** et **init_of_parameters** serviront à effectuer un passage de paramètres lors de l'appel de la fonction C représentant celui-ci et à mémoriser dans des variables globales l'adresse de toute donnée de niveau 01 ou 77 de sa linkage section (cfr 7.4.2).

Le programme C global généré sera composé du contenu de la fonction **progr_c** définie ci-après du programme COBOL principal suivi du contenu des fonctions **progr_c** (dans l'ordre spécifié par l'utilisateur) de ses sous-programmes. Afin de réaliser cela, une édition de lien, en dehors du processus de compilation incrémentale, sera réalisée.

7.4.1. Structure du code généré pour un programme COBOL

Deux cas sont à envisager.

Dans le cas où un programme COBOL s'appelle "**main**", sa fonction **progr_c** est définie comme suit :

```
progr_c ( < cob_pgm > ) =
        #include <stdio.h>
        int size_error;
        char highvalue;
        char lowvalue;
        decl_of_files ( < data_div > )
        main()
        {
        char *ptralloc;
        decl_of_var_for_value_clauses ( < data_div > )
        highvalue = -1;
        lowvalue = '\0';
        malloc_instruction ( < data_div > )
        repr_c_of_value_clauses ( < data_div > )
        repr_c_of_proc_div ( < proc_div > )
        }
        repr_c_of_sections ( < proc_div > )
        repr_c_of_paragraphs ( < proc_div > )
```

Dans le cas d'un sous-programme de nom quelconque, la structure du code généré associé à celui-ci est la suivante :

```
progr_c ( < cob_pgm > ) =
        decl_of_files ( < data_div > )
        decl_of_ptrs_for_param1 ( < proc_div > )
        decl_of_ptrs_for_param1 ( < data_div > )
        progr_name_header ( < cob_pgm > )
        decl_of_ptrs_for_param2 ( < proc_div > )
        {
        char *ptralloc;
        decl_of_var_for_value_clauses ( < data_div > )
        malloc_instruction ( < data_div > )
        init_of_parameters ( < proc_div > )
        init_of_parameters ( < data_div > )
        repr_c_of_value_clauses ( < data_div > )
        repr_c_of_proc_div ( < proc_div > )
        }
        repr_c_of_sections ( < proc_div > )
        repr_c_of_paragraphs ( < proc_div > )
```

7.4.2. Fonctions permettant de réaliser l'appel d'un sous-programme COBOL

L'appel d'un sous-programme est réalisé grâce aux fonctions suivantes :

- decl_of_ptrs_for_param1(<proc_div>),

- decl_of_ptrs_for_param1(<data_div>),

- decl_of_ptrs_for_param2(<proc_div>),

- progr_name_header(<id_div>),

- init_of_parameters(<proc_div>),

- init_of_parameters(<data_div>).

Ces fonctions dépendent de la spécification ou non de la clause **USING** apparaissant dans l'en-tête de la procedure division d'un programme COBOL. Elles permettent de passer à la fonction C représentant le programme appelé l'adresse des données de niveau 01 et 77 déclarées dans sa linkage section. Le mécanisme par lequel ces adresses sont rendues disponibles aux fonctions C représentant les sections ou paragraphes d'un programme COBOL consiste à utiliser des variables globales (pointeurs) que nous initialisons à la valeur des arguments passés à la fonction C représentant ce programme.

La fonction **decl_of_ptrs_for_param1** déclare ces variables globales. Celles-ci portent le même nom que les données de niveau 01 et 77 de la linkage section mais précédé du nom du programme suivi du caractère "_" afin d'éviter lors de la compilation du texte C global généré de déclarer plusieurs fois une même variable et afin de retrouver toujours à partir de ce nom la variable mémorisant l'adresse d'une donnée de niveau 01 ou 77 de la linkage section.

Ces adresses sont d'abord mémorisées dans des variables locales représentant les arguments passés à la fonction appelée. Le nom de celles-ci est par convention composé du caractère "**p**" et d'un numéro (fonction **nvar**) variant de 1 à n si n représente le nombre d'arguments de la clause USING. Ces variables (pointeurs) sont déclarées dans la fonction **decl_of_ptrs_for_param2** et lors de l'exécution du programme C généré, les variables globales déclarées dans la fonction **decl_of_ptrs_for_param1** sont initialisées dans la fonction **init_of_parameters** à la valeur de celles-ci.

Dans le cas où aucune clause USING n'est spécifiée, la fonction C représentant le sous-programme ne possède aucun paramètre et aucune variable n'est générée.

Ces fonctions sont définies de la manière suivante :

```
decl_of_ptrs_for_param1 ( < proc_div > ) =
          nil     si < proc_div > ::= nil | < procdivbody >
    |     decl_of_ptrs_for_param1 ( < parameterlist > )
                  si < proc_div > ::= < parameterlist > < procdivbody >

decl_of_ptrs_for_param1 ( < parameterlist >1 ) =
          decl_of_ptr_for_param1 ( < identifier > )
                  si < parameterlist >1 ::= < identifier >
    |     decl_of_ptr_for_param1 ( < identifier > )
          decl_of_ptrs_for_param1 ( < parameterlist >2)
                  si < parameterlist >1::= < identifier > < parameterlist >2

decl_of_ptr_for_param1 ( < identifier > ) =
          char *progr_name ( < id_div > )_< userdef_name >;

progr_name ( < id_div > ) = < program_name >

progr_name_header ( < cob_pgm  > ) =
          progr_name ( < id_div > ) adr_of_parameters ( < proc_div > )

adr_of_parameters ( < proc_div > ) =
          ( adr_of_parameters ( < parameterlist > ) )
                  si < proc_div > ::= < parameterlist > < procdivbody >
    |     ()      si < proc_div > ::= nil | < procdivbody >

adr_of_parameters ( < parameterlist >1 ) =
          adr_of_a_parameter ( < identifier > )
                  si < parameterlist >1 ::= < identifier >
    |     adr_of_a_parameter ( < identifier > ) , adr_of_parameters  (<parameterlist>2)
                  si < parameterlist >1 ::= < identifier >< parameterlist>2

adr_of_a_parameter ( < identifier > ) = pnvar( < identifier > )

nvar ( < parameterlist > = 1  si  < proc_div > ::= < parameterlist >
                                                   < procdivbody >

nvar ( < identifier > )  =  nvar ( < parameterlist >1 )
                  si < parameterlist >1 ::= < identifier >
                                  |     < identifier > < parameterlist>2

nvar ( < parameterlist >2 ) = rnvar ( < identifier > )
                  si < parametelist >1 ::= < identifier > < parameterlist >2

rnvar ( < identifier > ) = nvar ( < identifier > ) + 1
```

decl_of_ptrs_for_param2 (< proc_div >) =
nil si < proc_div > ::= nil | < procdivbody >
| decl_or_ptrs_for_param2 (< parameterlist >)
si < proc_div > ::= < parameterlist > < procdivbody >

decl_or_ptrs_for_param2 (< parameterlist >$_1$) =
decl_of_ptr_for_param2 (< identifier >)
si < parameterlist >$_1$ ::= < identifier >
| decl_of_ptr_for_param2 (< identifier >)
decl_of_ptrs_for_param2 (< parameterlist >$_2$)
si < parameterlist >$_1$ ::= < identifier > < parameterlist >$_2$

decl_of_ptr_for_param2 (< identifier >) =
char *pnvar (< identifier > **);**

init_of_parameters (< proc_div >) =
nil si < proc_div > ::= nil | < procdivbody >
| init_of_parameters (< parameterlist >)
si < proc_div > ::= < parameterlist > < procdivbody >

init_of_parameters (< parameterlist >$_1$) =
init_of_a_parameter (< identifier >)
si < parameterlist >$_1$ ::= < identifier >
| init_of_a_parameter (< identifier >)
init_of_parameters (< parameterlist >$_2$)
si < parameterlist >$_1$ ::= < identifier > < parameterlist >$_2$

init_of_a_parameter (< identifier >) =
progr_name (< id_div >)_< userdef_name > **= p**nvar
(< identifier >);

decl_of_ptrs_for_param1 (< data_div >) =
decl_of_ptrs_for_param1 (< lk_sec >)
si < data_div > ::= < file_sec >
< ws_sec >
< lk_sec >
| nil

decl_of_ptrs_for_param1 (< lk_sec >) =
decl_of_ptrs_for_param1 (< des_list >)
si < lk_sec > ::= < des_list >
| nil si < lk_sec > ::= nil

decl_of_ptrs_for_param1 (< des_list >$_1$) =
dpp1 (< recdes_entry >)
decl_of_ptrs_for_param1 (< des_list >$_2$)
si < des_list >$_1$ ::= < recdes_entry >< des_list >$_2$
| nil si < des_list >$_1$ ::= nil

dpp1 (< recdes_entry >) =
nil si < redefclause > ::= nil
| **char** *progr_name (<id_div>)_name (<fill_or_data >);

name (< fill_or_data >) = < userdef_name >

init_of_parameters (< data_div >) =
init_of_parameters (< lk_sec >)
si < data_div > ::= < file_sec >
< ws_sec >
< lk_sec >
| nil si < data_div > ::= nil

init_of_parameters (< lk_sec >) =
nil si < lk_sec > ::= nil
| init_of_parameters (< des_list >)
si < lk_sec > ::= < des_list >

init_of_parameters (< des_list >$_1$) =
nil si < des_list >$_1$ ::= nil
| initpar (< recdes_entry >)
init_of_parameters (< des_list >$_2$)
si < des_list >$_1$ ::= < recdes_entry > < des_list >$_2$

initpar (< recdes_entry >) =
nil si < redefclause > ::= nil
| progr_name (< id_div >)_name (< fill_or_data >) =
progr_name (< id_div >)_name (< redefclause >);

name (< redefclause >) = name (< data_name >)
= < userdef_name >

Exemple :

Soit le programme COBOL de nom **prgcob** dont l'en-tête de la procedure division est :

PROCEDURE DIVISION USING a, b, c.

Si sa linkage section se présente de la manière suivante :

```
LINKAGE SECTION.
01 a ....
01 b ....
77 c ....
01 d REDEFINES a ....
01 e REDEFINES b ....
```

La valeur des fonctions présentées ci-avant sont les suivantes :

decl_of_ptrs_for_param1(< proc_div >) =

```
char *prgcob_a;
char *prgcob_b;
char *prgcob_c;
```

decl_of_ptrs_for_param1(< data_div >) =

```
char *prgcob_d;
char *prgcob_e;
```

progr_name_header(< cob_pgm >) =

```
prgcob( p1, p2, p3 )
```

decl_of_ptrs_for_param2(< proc_div >) =

```
char *p1;
char *p2;
char *p3;
```

init_of_parameters(< proc_div >) =

```
prgcob_a = p1;
prgcob_b = p2;
prgcob_c = p3;
```

init_of_parameters(< data_div >) =

```
prgcob_d = prgcob_a;
prgcob_e = prgcob_b;
```

7.4.3. Conclusion

Cette section nous a permis de définir la structure globale du programme C généré, ainsi que la valeur de certaines fonctions. Les sections suivantes permettront de définir la valeur des fonctions n'ayant pas été traitées ici.

7.5. Représentation de la Data Division

7.5.1. Introduction

Cette section expose la manière de mémoriser et de référencer dans le programme C généré, les données déclarées en data division d'un programme COBOL. Elle décrit également la façon de représenter les fichiers COBOL et les opérations sur ceux-ci.

7.5.2. Principe de mémorisation

Au sein d'une même zone mémoire allouée lors de l'exécution du programme C généré, sont regroupées les données déclarées en file et working-storage section d'un programme COBOL.

Les données de la linkage section ne doivent pas être mémorisées dans cette zone car leur adresse est toujours transférée lors des instructions CALL appelant le programme (cfr 7.4.2 et 7.7.8).

Cette zone mémoire est demandée explicitement dans la fonction C représentant un programme COBOL par l'appel de la fonction **malloc(n)** permettant de réclamer au système d'exploitation une zone constituée de **n** caractères consécutifs et renvoyant un pointeur vers le début de cette zone (fonction **malloc_instruction** cfr 7.5.2.4.1). Ce pointeur est représenté dans le programme C par la variable **ptralloc.**

7.5.2.1. Description de la zone mémoire

Cette zone mémoire comprend deux parties.

La première sert à mémoriser les données ou buffers de la file section. La seconde permet la mémorisation des données de la working-storage section.

7.5.2.2. Principes de calcul de la taille mémoire réservée pour une donnée

Nous examinerons d'abord le problème de la taille mémoire à allouer aux données COBOL élémentaires et ensuite celui des données structurées.

7.5.2.2.1. Calcul de la taille mémoire réservée pour une donnée élémentaire

Précisons d'abord ici que nous n'avons pas tenu compte de la clause **USAGE IS COMPUTATIONAL** pouvant figurer dans les déclarations de données numériques; nous avons donc attribué systématiquement aux données numériques la clause **USAGE IS DISPLAY**. Il s'agit ici pour nous d'un choix de représentation en mémoire des données numériques permetttant de simplifier grandement la gestion de cette mémoire, ainsi que la représentation des instructions COBOL.

Nous examinerons d'abord le cas des données alphanumériques et ensuite celui des données numériques.

a) Taille mémoire des données alphanumériques

Le principe que nous utilisons ici est le suivant :

1° Toute donnée alphanumérique ne possédant pas de clause **OCCURS** se voit attribuer une zone dont la taille correspond au nombre spécifié entre parenthèses succédant le caractère "**X**" lorsque son format <alphanumeric_format> est "**X**(<integer>)", soit au nombre de caractères "**X**" lorsque son format est "**X**<reptX>".

2° Toute donnée alphanumérique possédant une clause **OCCURS** voit sa taille mémoire définie de la même manière que dans le cas précédent mais celle-ci est multipliée par le nombre spécifié par <integer> apparaissant dans sa clause **OCCURS**.

3° Toute donnée alphanumérique possédant une clause **REDEFINES** se voit allouer la zone mémoire de la donnée qu'elle redéfinit.

b) Taille mémoire des données numériques

Le principe de calcul que nous utilisons dans ce cas s'exprime de la manière suivante :

Toute donnée numérique ne possédant pas de clause **OCCURS** se voit attribuer une zone mémoire composée des éléments suivants :

- un caractère servant à mémoriser son signe s'il est spécifié dans sa clause **PICTURE** par le caractère "**S**", signe que nous représentons en mémoire par le caractère "+" ou "-".

- une zone afin de mémoriser sa partie entière, dont la longueur est égale au nombre <integer> spécifié entre parenthèses succédant le caractère "**9**" lorsque <integer_partie1> ::= **9**(<integer>), soit au nombre de caractères "**9**" apparaissant dans <integer_partie1> lorsque son format est "**9**<rept9>", soit à 0 si celle-ci n'est pas spécifiée.

- et une zone servant à mémoriser sa partie décimale, dont la longueur est définie de la même façon que pour la partie entière mais appliqué à <decimal_partie2>.

Nous ne mémorisons donc pas la virgule d'une donnée numérique, ainsi que la suite de caractères "0" associée à la partie <power_partie> pouvant figurer dans sa clause PICTURE. Nous utiliserons afin de connaître ces caractéristiques l'environnement des données ENVDATA décrit dans le paragraphe 7.5.3.

Les règles définies concernant la clause OCCURS et la clause REDEFINES pour les données alphanumériques s'appliquent également ici.

7.5.2.2.2. Calcul de la taille mémoire réservée pour une donnée structurée

Une donnée structurée se voit allouer une zone mémoire dont la taille est égale à la somme des tailles mémoire allouées à ses composants. Si elle possède une clause OCCURS, cette somme est multipliée par le nombre <integer> spécifié dans celle-ci. De plus, si cette donnée possède une clause REDEFINES, les règles définies précédemment restent d'application.

7.5.2.3. Principe d'affectation d'une adresse en mémoire à une donnée COBOL

Le principe est le suivant :

Nous considérons avant tout examen des déclarations de données de la file et working-storage section, que l'adresse en mémoire de la première donnée rencontrée en data division est égale à 0 et que à chaque déclaration de donnée, selon l'ordre d'apparition dans le programme COBOL, elle se voit incrémentée du nombre de caractères requis afin de mémoriser cette donnée.

Les exceptions suivantes sont cependant à apporter à ce principe :

- L'adresse en mémoire des données définissant le buffer d'un fichier est la même pour chacune d'elles.

- L'adresse du premier composant d'une donnée structurée est la même que cette donnée structurée.

- L'adresse d'une donnée possédant une clause **REDEFINES** est celle de la donnée qu'elle redéfinit.

7.5.2.4. Fonctions servant au calcul de la taille de la mémoire et à l'attribution d'une adresse à une donnée

Afin d'appliquer les principes de calcul définis auparavant, nous utiliserons les fonctions suivantes :

-**n**(<data_div>) ayant pour valeur la taille de la mémoire à allouer aux données de la data division.

-**longueur**(<pictureclause>) spécifiant la taille de la zone mémoire attribuée à une donnée élémentaire.

-**longueur**(**<symbole>**) indiquant la taille mémoire requise afin de mémoriser la ou les données déclarées dans la partie de la data division spécifiée par le symbole **<symbole>**.

-**adr**(**<symbole>**) spécifiant l'adresse à attribuer à la première donnée déclarée dans la partie de la data division spécifiée par le symbole **<symbole>** ou à la donnée déclarée par <recdes_entry> si cette fonction lui est associée.

-**radr**(**<symbole>**) spécifiant l'adresse de la prochaine donnée qui sera rencontrée dans la partie de la data division succédant la partie spécifiée par **<symbole>**.

-**adr**(<**fill_or_data**>) spécifiant toujours l'adresse dans la zone mémoire de la donnée de nom <fill_or_data>.

-**adr**(<**data_name**>) spécifiant l'adresse en zone mémoire de la donnée de nom <data_name>.

7.5.2.4.1. Fonctions de calcul de la taille de la zone mémoire des données d'un programme COBOL et d'allocation de celle-ci

n (< data_div >) = 0 si < data_div > ::= nil

n (< data_div >) = longueur (< file_sec >) + longueur (< ws_sec >)
si < data_div > ::= < file_sec > < ws_sec > < lk_sec >

malloc_instruction (< data_div >) =
ptralloc = malloc(n (< data_div >) **);**

7.5.2.4.2. Fonctions de calcul de la taille mémoire allouée aux données de la file section

adr (< file_sec >) = 0

Si < file_sec > ::= nil,
longueur (< file_sec >) = 0
radr (< file_sec >) = adr (< file_sec >)

Si < file_sec > ::= < fdlist >,
longueur (< file_sec >) = longueur (< fdlist >)
radr (< file_sec >) = radr (< fdlist >)
adr (< fdlist >) = adr (< file_sec >)

Si < fdlist > ::= nil,
longueur (< fdlist >) = 0
radr (< fdlist >) = adr (< fdlist >)

Si < fdlist >$_1$ ::= < fditem > < fdlist >$_2$,
longueur (< fdlist >$_1$) = longueur (< fditem >)
+ longueur (< fdlist >$_2$)
adr (< fditem >) = adr (< fdlist >$_1$)
adr (< fdlist >$_2$) = radr (< fditem >)
radr (< fdlist >$_1$) = radr (< fdlist >$_2$)

Si < fditem > ::= nil,
longueur (< fditem >) = 0
radr (< fditem >) = adr (< fditem >)

Si < fditem > ::= < fdentry > < des_list >,
longueur (< fditem >) = longueur (< des_list >)
adr (< des_list >) = adr (< fditem >)
radr (< fditem >) = radr (< des_list >)

Si < des_list > ::= nil,
longueur (< des_list >) = 0
radr (< des_list >) = adr (< des_list >)

Si < des_list >$_1$ ::= < recdes_entry > < des_list >$_2$,
longueur (< des_list >$_1$) = longueur (< recdes_entry >)
adr (< recdes_entry >) = adr (< des_list >$_1$)
adr (< des_list >$_2$) = adr (< des_list >$_1$)
radr (< des_list >$_1$) = radr (< recdes_entry >)

7.5.2.4.3. Fonctions de calcul de la taille mémoire allouée aux données de la working-storage section

adr (< ws_sec >) = radr (< file_sec >)

Si < ws_sec > ::= nil,
longueur (< ws_sec >) = 0

Si < ws_sec > ::= < des _list >,
longueur (< ws_sec >) = longueur (< des_list >)
adr (< des_ list >) = adr (< ws_sec >)

Si < des_list > ::= nil,
longueur (< des_list >) = 0
radr (< des_list >) = adr (< des_list >)

Si < des_list >$_1$ ::= < recdes_entry > < des_list >$_2$,
longueur (< des_list >$_1$) = longueur (< recdes_entry >)
+ longueur (< des_list >$_2$)
adr (< recdes_entry >) = adr (< des_list >$_1$)
adr (< des_list >$_2$) = radr (< recdes_entry >)
radr (< des_list >$_1$) = radr (< des_list >$_2$)

7.5.2.4.4. Fonctions de calcul de la taille mémoire allouée à une donnée

Si < recdes_entry > ::= nil,
longueur (< recdes_entry >) = 0
radr (< recdes_entry >) = adr (< recdes_entry >)

Si < recdes_entry > ::= < recdes_rec >,
si < redefclause > ::= nil,
adr (< fill_or_data >) = adr (< recdes_entry >)
adr (< record_body >) = adr (< recdes_entry >)
radr (< recdes_entry >) = radr (< record_body >)
redef (< recdes_entry >) = 0
si < occursclause > ::= nil,
longueur (< recdes_entry >) =
longueur (< record_body >)
si < occursclause > ::= < integer >,
oc (< occursclause >) = < integer >
longueur (< recdes_entry >) =
longueur (< record_body >) *
oc (< occursclause >)

```
        si < redefclause > ::= < data_name >,
                adr ( < fill_or_data > ) = adr ( < data_name > )
                adr ( < record_body > ) = adr ( < fill_or_data > )
                radr ( < recdes_entry > ) = adr ( < recdes_entry > )
                redef ( < recdes_entry > ) = 1
                si < occursclause > ::= nil,
                        longueur ( < recdes_entry > ) = longueur ( < record_body>)
                si < occursclause > ::= < integer >,
                        oc ( < occursclause > ) = < integer >
                        longueur ( < recdes_entry > ) =
                                oc ( < occursclause > ) *
                                longueur ( < record_body > )

Si < recdes_entry > ::= < recdes_data >,
        si < redefclause > ::= nil,
                adr ( < fill_or_data > ) = adr ( < recdes_entry > )
                radr ( < recdes_entry > ) = adr ( < recdes_entry > )
                        + longueur ( < recdes_entry > )
                redef ( < recdes_entry > ) = 0
                si < occursclause > ::= nil,
                        longueur ( < recdes_entry > ) = longueur ( <pictureclause>)
                si < occursclause > ::= < integer >,
                        oc ( < occursclause > ) = < integer >
                        longueur ( < recdes_entry > ) =
                                longueur ( < pictureclause > ) *
                                oc ( < recdes_entry > )
        si < redefclause > ::= < data_name >,
                adr ( < fill_or_data > ) = adr ( < data_name > )
                radr ( < recdes_entry > ) = adr ( < recdes_entry >)
                redef ( < recdes_entry > ) = 1
                si < occursclause > ::= nil,
                        longueur ( < recdes_entry > ) = longueur ( <pictureclause>)
                si < occursclause > ::= < integer >,
                        oc ( < occursclause > ) = < integer >
                        longueur ( < recdes_entry > ) =
                                oc ( < occursclause > ) *
                                longueur ( < pictureclause > )
```

7.5.2.4.5. Fonctions de calcul de la taille mémoire allouée aux composants d'une donnée structurée

```
Si < record_body > ::= nil,
                longueur ( < record_body > ) = 0
                radr ( < record_body > ) = adr ( < record_body > )
```

Si < record_body >$_1$ ::= < recdes_entry > < record_body >$_2$,
si redef (< recdes_entry >) = 0,
longueur (< record_body >$_1$) =
longueur (< recdes_entry >) +
longueur (< record_body >$_2$)
adr (< record_body >$_2$) = radr (< recdes_entry >)
si redef (< recdes_entry >) = 1,
longueur (< record_body >$_1$) = longueur (< record_body >$_2$)
adr (< record_body >$_2$) = adr (< record_body >$_1$)
adr (< recdes_entry >) = adr (< record_body >$_1$)
radr (< record_body >$_1$) = radr (< record_body >$_2$)

7.5.2.4.6. Exemple

Soit la data division suivante :

```
DATA DIVISION.
FILE SECTION.
FD clients.
01 cli-passé.
      02 nom.
            03 prénom PIC X(20).
            03 nom-de-famille PIC X(20).
      02 FILLER PIC X(24).
01 cli-présent.
      02 nom-cli.
            03 prénom-cli PIC X(20).
            03 nom-client PIC X(20).
      02 sommes-versées OCCURS 3 TIMES.
            03 versé PIC 9(6)V9(2).
WORKING-STORAGE SECTION.
01 ok PIC X(2).
01 a  PIC X(20).
01 b  REDEFINES a OCCURS 4 TIMES.
      02 c PIC X(5).
```

La fonction n(<data_div>) spécifiant la taille de la mémoire à allouer aura comme valeur :

n(<data_div>) = longueur(<file_sec>) + longueur(<ws_sec>) = 86
longueur(<file_sec>) = 64
longueur(<ws_sec>) = 22.

Le tableau suivant donne la valeur des fonctions adr, longueur et radr associées aux données de la data division :

fill_or_data	adr	longueur	radr
cli-passé	0	64	64
nom	0	40	40
prénom	0	20	20
nom-de-famille	20	20	40
FILLER	40	24	64
cli-présent	0	64	64
nom-cli	0	40	40
prénom-cli	0	20	20
nom-client	20	20	40
sommes-versées	40	24	64
versé	40	8	48
ok	64	2	66
a	66	20	86
b	66	20	86
c	66	5	71

7.5.2.4.7. Fonction permettant de calculer l'adresse des données de niveau supérieur à 01 de la linkage section

Il est nécessaire de calculer pour toute donnée de la linkage section la valeur de la fonction **adr**, car seule l'adresse des données de niveau 01 ou 77 est transférée lors d'un appel du programme (cfr 7.4.2 et 7.7.8). C'est la raison pour laquelle nous considérons que la valeur de la fonction **adr** de toute donnée de niveau 01 ou 77 est égale à 0 et que celle des données de numéro de niveau supérieur est calculée par rapport à cette valeur 0 initiale de la même manière que celle exposée dans le paragraphe 7.5.2.4.4.

La valeur de la fonction **adr** des données de niveau 01 ou 77 est donc établie de la manière suivante :

Si $< \text{des_list} >_1 ::= < \text{recdes_entry} > < \text{des_list} >_2$,

$\text{adr} (< \text{recdes_entry} >) = 0$

$\text{adr} (< \text{des_list} >_2) = 0$

7.5.3. Description de l'environnement des données

Afin de représenter les instructions d'un programme COBOL, nous fournissons à chaque composant de la procedure division un environnement de compilation que nous appelons **ENVDATA** rassemblant les caractéristiques relatives à chacune de ses données.

7.5.3.1. Caractéristiques communes aux données alphanumériques et numériques

Pour toute donnée déclarée en data division, cet environnement reprend les informations suivantes :

- le nom de cette donnée,
- son adresse en zone mémoire,
- le nombre de caractères requis afin de la mémoriser,
- et le type de cette donnée (numérique ou alphanumérique).

Le nom des colonnes reprenant ces caractéristiques dans l'environnement des données sont appelées respectivement **name**, **adresse**, **np** et **pic**.

a) définition de la colonne name

Cette colonne reprend le nom de toutes les données déclarées en data division, excepté celles dont le nom est 'FILLER', et dont la valeur est celle de la fonction **name** définie de la manière suivante :

name (< recdes_entry >) = name (< fill_or_data >)
= name (< data_name >)
= < userdef_name >.

b) définition de la colonne adresse

Cette colonne reprend la valeur des fonctions **adr(<fill_or_data>)** des données de la data division.

c) définition de la colonne np

Cette colonne reprend la valeur des fonctions **longueur** des données de la data division.

d) définition de la colonne pic

Cette colonne rassemble la valeur des fonctions **pic** des données déclarées en data division définies comme suit :

pic (< recdes_entry >) = "**X**" si < recdes_entry > ::= < recdes_rec >
| pic (< recdes_data >)
si < recdes_entry > ::= < recdes_data >

pic (< recdes_data >) = pic (< pictureclause >)

pic (< pictureclause >) = pic (< picstring >)

pic (< picstring >) = "**X**" si < picstring > ::= < alphanumeric_format >
| "**9**" si < picstring > ::= < numericint_format >
| < numericdec_format >

7.5.3.2. Caractéristiques propres aux données numériques

Afin de reporter dans cet environnement les caractéristiques propres aux données numériques, nous ajoutons ici quatre nouvelles colonnes de noms **usage**, **p**, **v** et **s** servant à caractériser respectivement le type de clause USAGE, le facteur de puissance, la position de la virgule et le caractère signé ou non de ces données.

La connaissance du type de clause USAGE pour une donnée numérique n'est pas tellement intéressante dans le cadre de notre génération de code vu la représentation selon l'usage DISPLAY en mémoire de ces données, mais s'avère une information importante lors de certaines vérifications sémantiques. Les trois autres caractéristiques sont par contre essentielles, elles permettent en connaissant la longueur et l'adresse d'une donnée numérique de pouvoir restituer sa valeur (cfr 7.6.1.2.1 et 7.6.1.2.2).

a) définition de la colonne usage

Cette colonne reprend la valeur des fonctions **usage** caractérisant le type de clause USAGE pouvant figurer dans la déclaration des données de la data division et définies comme suit :

usage (< recdes_entry >) = "**D**" si < usageclause > ::= nil
| **USAGE IS DISPLAY**
| "**C**" si < usageclause > ::=
USAGE IS COMPUTATIONAL

b) définition de la colonne p

Etant donné que les caractères ('0') correspondant au facteur de puissance ne sont pas mémorisés en zone mémoire (cfr 7.5.2.2.1), nous reprenons dans la colonne que nous appelons **p** la valeur des fonctions **p(<recdes_entry>)** indiquant le nombre de ces caractères. La valeur de cette fonction est égale au nombre de caractères "**P**" apparaissant dans la partie <power_partie> lorsque son format est "**P<reptP>**", soit au nombre <integer> spécifié entre

parenthèses succédant le caractère "**P**" lorsque son format est "**P**(<integer>)", soit à 0 quand cette partie n'est pas spécifiée ou que la donnée n'est pas de type numérique.

c) définition de la colonne v

La virgule d'une donnée numérique n'étant pas mémorisée en zone mémoire (cfr 7.5.2.2.1), cette colonne reporte la valeur des fonctions **v**(<**recdes_entry**>) permettant de définir en zone mémoire la position de celle-ci. Ces fonctions reçoivent la valeur -1 dans le cas où il n'existe pas de partie décimale (<decimal_partie2> non spécifiée), la valeur 0 s'il n'y a seulement des chiffres qu'après la virgule (<integer_partie1> non spécifiée), ou une valeur supérieure à 0 s'il existe à la fois une partie entière et décimale, et correspondant à la longueur de la zone mémoire allouée afin de mémoriser la partie entière (cfr 7.5.2.2.1).

Remarquons également que la valeur en colonne **p** d'une donnée correspond au facteur de puissance pour la partie entière dans le cas où v(<recdes_entry>) = -1 et pour la partie décimale quand v (<recdes_entry>) = 0. De plus, nous mettons par convention dans ces colonnes les valeurs 0 (dans **p**) et -1 (dans **v**) pour toute donnée n'étant pas de type numérique.

d) définition de la colonne s

Le signe d'une donnée numérique étant mémorisé, cette colonne reprend la valeur des fonctions **s**(<**recdes_entry**>) spécifiant le caractère signé ou non des données numériques. Ces fonctions reçoivent la valeur 1 dans le cas d'une donnée signée ou la valeur 0 dans le cas inverse. De plus, nous mettons par convention dans ces colonnes la valeur 0 pour toute donnée n'étant pas numérique.

7.5.3.3. Caractéristiques nécessaires afin de calculer l'adresse de toute donnée COBOL définie par <data_name> ou <identifier>

Afin d'établir les formules du calcul d'adresse du paragraphe 7.5.4, sont ajoutées dans l'environnement des données les colonnes de nom **lk, lkname, locc1**, **locc2** et **locc3** définies ci-dessous :

a) Définition de la colonne lk

Cette colonne regroupe la valeur des fonctions **lk**(<**recdes_entry**>) servant à spécifier l'appartenance ou non d'une donnée à la linkage section. Il est en effet important de connaître cette propriété car le calcul d'adresse des données de la linkage section se base sur la valeur que reçoivent les pointeurs définis dans la fonction **decl_of_ptrs_for_param1** (cfr 7.4.2).

La valeur d'une fonction **lk**(<**recdes_entry**>) est égale à 1 si la donnée définie par ce <recdes_entry> est déclarée en linkage section, à 0 dans le cas contraire.

b) Définition de la colonne lkname

Etant donné que seule l'adresse des données de niveau 01 ou 77 déclarées en linkage section est transférée lors de l'appel d'un programme, cette colonne contient pour toute donnée déclarée en linkage section le nom de la donnée de niveau 01 dans la déclaration de laquelle elle apparaît. Cette information est primordiale car elle permet de retrouver le nom du pointeur déclaré dans la fonction **decl_of_ptrs_for_param1** et de calculer l'adresse de cette donnée à partir de sa fonction **adr**.

Les données n'appartenant pas à la linkage section reçoivent par convention dans cette colonne la valeur **nil**, sinon celle de la fonction **lkname** définie ci-dessous.

Afin d'établir la valeur à mettre dans cette colonne sont également utilisées les fonctions **levelone** et **lkname** définies de la manière suivante :

levelone (< recdes_entry >) = 1 si < des_list >$_1$ ::= < recdes_entry > < des_list >$_2$
| 0 si < record_body >$_1$ ::= <recdes_entry> < record_body >$_2$

Si levelone (< recdes_entry >) = 1,
lkname (< recdes_entry >) = name (< fill_or_data >)
si < recdes_entry > ::= < recdes_rec >,
lkname (< record_body >) = name (< fill_or_data >)

Si levelone (< recdes_entry >) = 0,
lkname (< recdes_entry >) = lkname (< record_body >$_1$)
lkname (< record_body >$_2$) = lkname (< record_body >$_1$)

c) Définition des colonnes locc1, locc2, locc3

Ces trois colonnes sont utilisées afin de calculer l'adresse des éléments de tableaux COBOL. La fonction **adr** définie précédemment ne donne seulement l'adresse que du premier élément d'un tableau. C'est pourquoi, afin de connaître l'adresse exacte d'un élément de tableau, nous utilisons les fonctions **locc1**, **locc2** et **locc3** suivantes dont les valeurs permettent de déterminer le déplacement en mémoire qu'il faut effectuer afin de se positionner exactement sur l'élément de tableau désigné par les indices <subscript>.

Afin de définir ces trois fonctions, nous utilisons les nouvelles fonctions suivantes :

- **occurslevel(<symbole>)** indiquant le nombre de clauses OCCURS rencontrées avant la déclaration des données de la partie de la data division spécifiée par **<symbole>**,

- **hlocc1**, **hlocc2** et **hlocc3** associées à un symbole **<symbole>** et servant à transmettre à la prochaine déclaration de donnée rencontrée dans la partie de la data division spécifiée par

<symbole> , la valeur des fonctions **locc1**, **locc2** et **locc3** de la donnée de niveau inférieur précédant cette partie.

Ces fonctions sont définies comme suit :

```
occurslevel ( < recdes_entry > ) = 0 quand < des_list > ::= < recdes_entry >
                                                          < des_list >

Si < occursclause > ::= nil,
        si occurslevel ( < recdes_entry > ) = 0,
                locc1 ( < recdes_entry > ) = 0
                locc2 ( < recdes_entry > ) = 0
                locc3 ( < recdes_entry > ) = 0
                si < recdes_entry > ::= < recdes_rec >,
                        occurslevel ( < record_body > ) = 0
        si occurslevel ( < recdes_entry > ) = 1,
                locc1 ( < recdes_entry > ) =
                        hlocc1 ( < recdes_entry > )
                locc2 ( < recdes_entry > ) = 0
                locc3 ( < recdes_entry > ) = 0
                si < recdes_entry > ::= < recdes_rec >,
                        occurslevel ( < record_body > ) = 1
        si occurslevel ( < recdes_entry > ) = 2,
                locc1 ( < recdes_entry > ) =
                        hlocc1 ( < recdes_entry > )
                locc2 ( < recdes_entry > ) =
                        hlocc2 ( < recdes_entry > )
                locc3 ( < recdes_entry > ) = 0
                si < recdes_entry > ::= < recdes_rec >,
                        occurslevel ( < record_body > ) = 2
        si occurslevel ( < recdes_entry > ) = 3,
                locc1 ( < recdes_entry > ) =
                        hlocc1 ( < recdes-entry > )
                locc2 ( < recdes_entry > ) =
                        hlocc2 ( < recdes_entry > )
                locc3 ( < recdes_entry > ) =
                        hlocc3 ( < recdes_entry > )
                si < recdes_entry > ::= < recdes_rec >,
                        occurslevel ( < record_body > ) = 3

Si < occursclause > ::= < integer >,
        si occurslevel ( < recdes_entry > ) = 0,
                locc1 ( < recdes_entry > ) =
                        longueur ( < recdes_entry > ) /
                        oc ( < occursclause > )
                locc2 ( < recdes_entry > ) = 0
```

locc3 (< recdes_entry >) = 0
si < recdes_entry > ::= < recdes_rec >,
occurslevel (< record_body >) = 1
si occurslevel (< recdes_entry >) = 1,
locc1 (< recdes_entry >) =
hlocc1 (< recdes_entry >)
locc2 (< recdes_entry >) =
longueur (< recdes_entry >) /
oc (< occursclause >)
locc3 (< recdes_entry >) = 0
si < recdes_entry > ::= < recdes_rec >,
occurslevel (< record_body >) = 2
si occurslevel (< recdes_entry >) = 2,
locc1 (< recdes_entry >) =
hlocc1 (< recdes_entry >)
locc2 (< recdes_entry >) =
hlocc2 (< recdes_entry >)
locc3 (< recdes_entry >) =
longueur (< recdes_entry >) /
oc (< occursclause >)
si < recdes_entry > ::= < recdes_rec >,
occurslevel (< record_body >) = 3

Si < recdes_entry > ::= < recdes_rec >,
hlocc1 (< record_body >) = locc1 (< recdes_entry >)
hlocc2 (< record_body >) = locc2 (< recdes_entry >)
hlocc3 (< record_body >) = locc3 (< recdes_entry >)

Si < record_body >$_1$ ::= < recdes_entry > < record_body >$_2$,
hlocc1 (< recdes_entry >) =
hlocc1 (< record_body >$_1$)
hlocc2 (< recdes_entry >) =
hlocc2 (< record_body >$_1$)
hlocc3 (< recdes_entry >) =
hlocc3 (< record_body >$_1$)
hlocc1 (< record_body >$_2$) =
hlocc1 (< record_body >$_1$)
hlocc2 (< record_body >$_2$) =
hlocc2 (< record_body >$_1$)
hlocc3 (< record_body >$_2$) =
hlocc3 (< record_body >$_1$)

Exemple :

Soit la déclaration de tableau suivante définie en working-storage section :

```
01 a.
     02 b OCCURS 2 TIMES.
          03 c OCCURS 3 TIMES.
               04 d OCCURS 2 TIMES.
                    05 e PIC X(20).
               04 f PIC X(10).
          03 g OCCURS 2 TIMES.
               04 h PIC X(10).
```

En considérant que l'adresse de début du tableau a est 0 en zone mémoire, le tableau suivant présente le contenu des colonnes name, adresse, np, locc1, locc2 et locc3 de l'environnement des données ENVDATA qui nous intéressent ici :

name	adresse	np	locc1	locc2	locc3
a	0	340	0	0	0
b	0	340	170	0	0
c	0	150	170	50	0
d	0	40	170	50	20
e	0	20	170	50	20
f	40	10	170	50	0
g	150	20	170	10	0
h	150	10	170	10	0

7.5.4. Calcul d'adresse à l'exécution

Dans ce paragraphe est exposée la manière de calculer à partir de l'environnement ENVDATA l'adresse des données <data_name> et <identifier> figurant dans la représentation des instructions COBOL de la section 7.

7.5.4.1. Mode de calcul de l'adresse d'une donnée <data_name>

L'adresse de ce type de donnée est représentée par une fonction que nous appelons **adresse** définie à partir des fonctions **adr**, **lk** et **lkname** spécifiées ci-dessous :

adr (< data_name >) = ENVDATA (name = < userdef_name > , adresse),

lk (< data_name >) = ENVDATA (name = < userdef_name > , lk),

lkname (< data_name >) = ENVDATA (name = < userdef_name >, lkname).

Elle se calcule de la manière suivante :

Si lk (< data_name >) = 0,
adresse (< data _name >) = **ptralloc** + adr (< data_name >)

si lk (< data_name >) = 1,
adresse (< data _name >) = progr_name (< id_div >)_lkname
(< data_name >) + adr (< data_name >).

On observe ainsi qu'à l'exécution du programme C, l'adresse dans l'environnement ENVDATA de la donnée de nom <userdef_name> est incrémentée de la valeur que reçoit le pointeur **ptralloc** si cette donnée n'est pas déclarée en linkage section, et que dans le cas contraire, lui est ajoutée la valeur du pointeur déclaré dans la fonction **decl_of_ptrs_for_param1** (cfr 7.4.2).

Cette observation s'appliquera également pour le calcul d'adresse à l'exécution des données <identifier>.

7.5.4.2. Mode de calcul de l'adresse d'une donnée <identifier>

L'adresse de ce type de donnée est représentée par une fonction que nous appelons également **adresse** définie grâce aux fonctions suivantes :

adr (< identifier >) = ENVDATA (name = < usedef_name >, adresse),

lk (< identifier >) = ENVDATA (name = < userdef_name >, lk),

lkname (< identifier >) = ENVDATA (name = < userdef_name >,lkname),

locc1 (< identifier >) = ENVDATA (name = < userdef_name >, locc1),

locc2 (< identifier >) = ENVDATA (name = < userdef_name >, locc2),

locc3 (< identifier >) = ENVDATA (name = < userdef_name >, locc3).

Elle se calcule comme suit :

Si < identifier > ::= < userdef_name >,
si lk (< identifier >) = 0,
adresse (< identifier >) = **ptralloc** + adr (< identifier >)
si lk (< identifier >) = 1,
adresse (< identifier >) = progr_name (< id_div >)_lkname (<identifier>)
+ adr (<data_name>)

Si < identifier > ::= < userdef_name > < subscript >,
si lk (< identifier >) = 0,

adresse (< identifier >) = **ptralloc** + adr (<identifier >) +
(intvalue (< subscript >) - **1**)*locc1 (<identifier>)

si lk (< identifier >) = 1,

adresse (< identifier >) = progr_name (< id_div >)_lkname
(< identifier >) + adr (< data_name >) +
(intvalue (<subscript >) - **1**)*locc1 (< identifier >)

Si < identifier > ::= < userdef_name >< subscript >$_1$< subscript>$_2$,

si lk (< identifier >) = 0,

adresse (< identifier >) = **ptralloc** + adr (< identifier>) +
(intvalue (< subscript >$_1$) - **1**)*locc1 (< identifier>)+
(intvalue (< subscript >$_2$) - **1**)*locc2(< identifier>)

si lk (< identifier >) = 1,

adresse (< identifier >) = progr_name (< id_div >)_lkname
(< identifier >) + adr (< data_name >) +
(intvalue (< subscript >$_1$) - **1**)*locc1 (< identifier >) +
(intvalue (< subscript >$_2$) - **1**)*locc2 (<identifier >)

Si < identifier > ::= < userdef_name >< subscript >$_1$< subscript >$_2$< subscript >$_3$,

si lk (< identifier >) = 0,

adresse (< identifier >) = **ptralloc** + adr
(< identifier >) + (intvalue (< subscript >$_1$) **-1**)*locc1(< identifier >)+
(intvalue (< subscript >$_2$) - **1**)*locc2(< identifier >)+
(intvalue(< subscript >$_3$) - **1**)*locc3(< identifier >)

si lk (< identifier >) = 1,

adresse (< identifier >) = progr_name (< id_div >)_
lkname (< identifier >) + adr (< data_name >) +
(intvalue (< subscript >$_1$) - **1**)*locc1 (< identifier >) +
(intvalue (< subscript >$_2$) - **1**)*locc2 (< identifier >) +
(intvalue (< subscript >$_3$) - **1**)*locc3 (< identifier >)

intvalue (< subscript >) = < integer >
si < subscript > ::= < integer >
| **ntoi**(attr_of_id (<data_name>))
si < subscript > ::= < data_name >
| (**ntoi**(attr_of_id (<data_name>)) + <integer>)
si < subscript > ::= <data_name > + <integer>
| (**ntoi**(attr_of_id (<data_name>)) -<integer>)
si < subscript > ::= <data_name > - <integer>

ntoi est une fonction C que nous spécifierons dans la section 6 et renvoyant à partir des arguments décrits dans la fonction **attr_of_id** (cfr 7.7.5.1) la valeur entière correspondant à la donnée <data_name>.

Exemple :

Nous nous baserons ici sur l'exemple du paragraphe 7.5.3.3 afin de définir l'adresse à l'exécution des quelques données suivantes :

```
adresse ( b(2) )     = ptralloc + adr ( b ) + ( 2 - 1 )*locc1 ( b )
                     = ptralloc +   0     +    1   * 170.

adresse ( c(1,3) )   = ptralloc + adr ( c ) + ( 1 - 1 )*locc1 ( c ) + ( 3 - 1 )*locc2 ( c )
                     = ptralloc +   0     +    0   * 170        +    2   * 50.

adresse ( e(2,2,4) ) = ptralloc + adr ( e ) + ( 2 - 1 )*locc1 ( e ) +
                     + ( 2 - 1 )*locc2 ( e ) + ( 4 - 1 )*locc3 ( e )
                     = ptralloc +   0   +    1   *  170    +
                     +    1   *  50   +    3   *  20.
```

7.5.5. Représentation des opérations sur les fichiers COBOL

Ce paragraphe développe la manière de représenter les fichiers COBOL dans le programme C généré, ainsi que le principe de réalisation des opérations d'ouverture, fermeture, lecture et écriture sur ceux-ci. Il décrit également les environnements utilisés afin de réaliser ces opérations.

7.5.5.1. Représentation des fichiers COBOL

Tout fichier COBOL sera représenté par un fichier C (une suite de caractères).

7.5.5.2. Opérations sur les fichiers COBOL

7.5.5.2.1. L'ouverture d'un fichier

Nous utilisons en C afin de représenter l'instruction COBOL **OPEN**, la fonction **fopen** dont le premier argument est le nom externe du fichier et le second le "mode" indiquant la manière dont on a l'intention d'utiliser ce fichier. Les modes permis sont la lecture "**r**", l'écriture "**w**" et correspondent respectivement à la spécification des options **INPUT** et **OUTPUT** de l'instruction COBOL **OPEN**.

Cette fonction transmet après avoir exécuté certains traitements qui ne nous concernent pas ici un nom interne devant être utilisé lors des opérations de lecture et écriture sur ce fichier. Ce nom interne s'appelle aussi "**pointeur de fichier**" et la déclaration dont on a besoin pour celui-ci est citée ci-dessous en exemple :

FILE *fp; indiquant que fp représente un pointeur de fichier.

Un appel à la fonction **fopen** s'écrit donc en C :

pointeur de fichier = fopen(nom externe de fichier, mode).

Ainsi, afin de disposer d'un pointeur de fichier pour tout fichier COBOL dont le ou les "buffers" sont déclarés en file-section, nous utilisons le nom interne du fichier spécifié dans le <fdentry> de chaque <fditem> de la file section précédé du nom de programme et du caractère "_" afin de ne pas déclarer plusieurs fois ce même pointeur dans le programme C global généré. La fonction **decl_of_files** définie dans le paragraphe 7.5.5.5 sert à déclarer ces pointeurs de fichiers.

De plus, par convention, les noms externes de fichiers devront être indiqués par l'utilisateur de notre éditeur dans le premier <assignment_item> de la liste <assignmentlist> des clauses **SELECT** spécifiées en input-output section des programmes COBOL. Le nom

externe d'un fichier pourra également correspondre à l'entrée et la sortie standard pour lesquelles la valeur "stdin" et "stdout" doit être spécifiée dans ce même <assignment_item> et ne doit être déclaré aucun pointeur de fichier; ceux-ci sont représentés par les pointeurs de fichiers du même nom.

Dans une clause **SELECT** peut également être associée à un fichier une donnée composée de deux caractères et déclarée en working-storage section, contenant après toute opération sur ce fichier une valeur spécifique représentant le résultat de l'exécution de celle-ci. Cette donnée est appelée dans la terminologie COBOL un "**file status**".

La valeur du file status pouvant donc éventuellement être mise à jour lors de l'ouverture du fichier auquel elle est associée, nous utilisons une fonction de nom **fsopen** (cfr 7.6.4.1) réalisant à la fois l'opération d'ouverture du fichier (**fopen**) et cette mise à jour à partir de la valeur retournée par la fonction **fopen.**

Il nous faut donc disposer pour représenter ces opérations d'un environnement que nous appelerons **ENVFILES1** rassemblant pour tout fichier spécifié dans une clause **SELECT** son nom interne, son nom externe, une valeur 1 ou 0 selon la spécification ou non d'un file status dans cette clause, ainsi que le nom de la donnée utilisée comme file status associé à ce fichier.

Précisons également ici qu'un fichier associé à l'entrée et la sortie standard ne doit pas être ouvert et que seuls les fichiers dont le nom externe est différent de "stdin" et "stdout" doivent l'être.

7.5.5.2.2. La fermeture d'un fichier

L'instruction **CLOSE** est représentée en C par la fonction **fclose** dont le seul argument est un pointeur de fichier. De plus cette fonction **fclose** ne renvoyant aucune valeur particulière, nous ne savons mettre à jour la valeur du file status et l'utilisateur de notre éditeur devra veiller à ne pas tester la valeur de celui-ci après une instruction CLOSE; celle-ci est insignifiante après cette opération lors de l'exécution du programme C généré.

7.5.5.2.3. La lecture et l'écriture d'un fichier

Nous utilisons en C afin de représenter ces opérations les fonctions **read** et **write** pour lesquelles le premier argument est un pointeur de fichier, le second, une zone mémoire tampon ("buffer") de laquelle doit provenir la donnée ou dans laquelle elle doit aller. Le troisième argument représente le nombre de caractères à transférer.

Pour ces fonctions doivent donc être fournis : un pointeur de fichier, l'adresse en zone mémoire du buffer réservé pour ce fichier et le nombre de caractères à transférer correspondant à la taille de ce buffer. C'est la raison pour laquelle, afin de représenter l'instruction WRITE, un nouvel environnement que nous appelerons **ENVFILES2** doit être

créé; celui-ci associera à tout buffer déclaré dans le <fditem> d'un fichier son pointeur de fichier et permettra donc d'obtenir le nom interne du fichier devant être utilisé dans la fonction **write** du langage C. L'adresse du buffer et sa longueur se trouvent eux dans l'environnement **ENVDATA** décrit précédemment.

De plus, les fonctions **read** et **write** renvoyant une valeur concernant le résultat de leur exécution, sont utilisées deux nouvelles fonctions, **fsread** (cfr 7.6.4.2) et **fswrite** (cfr 7.6.4.3), permettant d'initialiser en fonction du résultat des opérations de lecture et écriture le file status du fichier concerné par ces opérations s'il a été spécifié pour celui-ci. Ces deux nouvelles fonctions doivent donc également disposer de l'environnement **ENVFILES1**.

7.5.5.3. Description de l'environnement ENVFILES1

Cet environnement est constitué des quatre colonnes suivantes :

- une colonne de nom **fname** contenant le nom des pointeurs de fichiers,

- une colonne de nom **efn** contenant le nom externe des fichiers dont le nom interne est donné en colonne **fname**,

- une colonne **fssp** de valeurs 1 ou 0 selon qu'un file status est associé ou non aux fichiers de la colonne **fname**,

- une colonne **fsname** contenant pour chaque fichier le nom de son file status s'il a été spécifié, sinon la valeur **nil.**

Exemple :

Soient les clauses SELECT suivantes déclarées dans un paragraphe file-control :

SELECT clients **ASSIGN TO** "clients.c"
ORGANIZATION IS SEQUENTIAL
ACCESS MODE IS SEQUENTIAL
FILE STATUS IS c.
SELECT toto **ASSIGN TO** "stdin"
ORGANIZATION IS SEQUENTIAL
ACCESS MODE IS SEQUENTIAL.

L'environnement des fichiers ENVFILES1 se présente de la manière suivante :

fname	**efn**	**fssp**	**fsname**
clients	clients.c	1	c
toto	stdin	0	""

7.5.5.4. Description de l'environnement ENVFILES2

Cet environnement est constitué des colonnes suivantes :

- une colonne de nom **fname** contenant des noms internes de fichiers,

- et d'une colonne **bufname** associant à ces noms internes le nom d'un buffer.

Afin de constituer cet environnement, nous utilisons une fonction de nom **filename** définie de la manière suivante :

filename (< fdentry >) = filename (< file_name >)
= < userdef_name >
si < fditem > ::= < fdentry > < des_list >

filename (< des_list >) = filename (< fdentry >)
si < fditem > ::= < fdentry > < des_list >

Si < des_list >$_1$ ::= < recdes_entry > < des_list >$_2$,
filename (< recdes_entry >) = filename (< des_list >$_1$)
filename (< des_list >$_2$) = filename (< des_list >$_1$).

La colonne de nom **fname** contient la valeur des fonctions **filename(<recdes_entry>)** des buffers déclarés dans la <des_list> de tout <fditem>, tandis que la colonne **bufname** celle des fonctions **name** (**<fill_or_data>**) de ces mêmes buffers.

Exemple :

Soit la file section suivante :

```
FD clients.
01 personnes ... .
01 fournisseurs .... .
FD etudiants.
01 etudiant ... .
```

Les fonctions **filename** et **name** associées aux buffers déclarés dans cette file section sont rassemblées dans l'environnement **ENVFILES2** suivant :

fname	**bufname**
clients	personnes
clients	fournisseurs
etudiants	etudiant

7.5.5.5. Fonction de déclaration des pointeurs de fichiers

```
decl_of_files ( < data_division > ) =
                nil      si < data_div > ::= nil
        |       decl_of_files ( < file_sec > )
                         si < data_div > ::= < file_sec >
                                             < ws_sec >
                                             < lk_sec >

decl_of_files ( < file_sec > ) =
                nil      si < file_sec > ::= nil
        |       decl_of_files ( < fdlist > )
                         si < file_sec > ::= < fdlist >

decl_of_files ( < fdlist >1 ) =
                nil      si < fdlist >1 ::= nil
        |       decl_of_file ( < fditem > )
                decl_of_files ( < fdlist >2 )
                         si < fdlist >1 ::= < fditem >< fdlist >2

decl_of_file ( < fditem > ) =
        specif_of_file_descr_in_c ( < fdentry > )
                si < fditem > ::= < fdentry > < des_list >

specif_of_file_descr_in_c ( < fdentry > ) =
                nil      si ENVFILES1 ( fname = < file_name >, efn ) =
                                                 "stdin" | "stdout"
        |       FILE *progr_name ( < id_div > )_<file_name >;
```

7.6. Spécifications des fonctions C utilisées

Cette section présente une classification des fonctions C utilisées afin de réaliser dans le programme C généré les diverses instructions COBOL et en donne les spécifications.

7.6.1. Fonctions d'assignation d'une valeur à une donnée

Les fonctions décrites dans ce paragraphe se basent sur les règles de cadrage de l'information utilisées dans le langage COBOL et sont essentiellement utilisées afin de réaliser en C les instructions COBOL arithmétiques, la clause **VALUE** et l'instruction **MOVE**. Nous indiquerons déjà ici au lecteur que les opérations arithmétiques effectuées dans le programme C généré traitent des valeurs flottantes double précision et que le résultat de celles-ci est ensuite mémorisé dans la zone mémoire.

7.6.1.1. Spécification de la fonction "dton"

Cette fonction apparaissant sous la forme **dton(adresse, longueur, p, v, s, rounded, &size_error, d**) sert à effectuer en zone mémoire le cadrage d'une donnée de type numérique.

Elle code une valeur flottante double précision **d** dans une donnée numérique dont on donne l'adresse **adresse** et la longueur **longueur** en zone mémoire en tenant compte du facteur de puissance **p**, de la position de la virgule **v** et du signe **s** de cette donnée, et renvoie la valeur **d**.

Cette fonction initialise également la variable globale entière de nom **size_error** (cfr 7.4.1) servant à réaliser les clauses **ON SIZE ERROR** et **NOT ON SIZE ERROR** de toute instruction arithmétique COBOL à la valeur 1 dans le cas où une troncature (perte de caractères) a lieu lors du cadrage en mémoire de la valeur **d** et prend également en charge la réalisation de la clause **ROUNDED.** Afin d'effectuer ces opérations, l'adresse de la variable **size_error** et une valeur 1 ou 0 dans l'argument **rounded** indiquant qu'une clause ROUNDED a été spécifiée ou non pour la donnée qu'il faut cadrer en mémoire, sont passées comme arguments de cette fonction.

Remarquons également que la valeur de la variable **size_error** ne peut être modifiée par cette fonction si elle est égale à la valeur 1 lors du début de l'exécution de celle-ci, de manière à toujours garder trace d'une éventuelle troncature apparue dans une fonction **dton** utilisée comme argument **d** de cette fonction. C'est également la raison pour laquelle avant l'exécution dans le programme C de toute opération arithmétique, la variable **size_error** est initialisée à la valeur 0 (cfr 7.7.5).

7.6.1.2. Spécification des instructions "ntod" et "ntoi"

Etant donné notre représentation en mémoire des données numériques, nous utiliserons les fonctions décrites dans ce paragraphe afin de restituer la valeur de ces données.

7.6.1.2.1. Spécification de la fonction "ntod"

Cette fonction convertit la représentation en mémoire d'une donnée numérique en une valeur flottante double précision qu'elle renvoie. Cette conversion s'effectue à partir de l'adresse, la longueur en zone mémoire, le facteur de puissance, la position de la virgule et le signe de cette donnée représentant dans cet ordre les arguments de cette fonction.

7.6.1.2.2. Spécification de la fonction "ntoi"

Cette fonction renvoie en utilisant les mêmes arguments que la fonction **ntod** une valeur entière représentant la valeur de la partie entière d'une donnée numérique.

7.6.1.3. Spécification de la fonction "codelitnumtodouble"

Cette fonction sert à convertir la représentation textuelle d'un littéral numérique COBOL en une valeur flottante double précision. Elle reçoit comme seul argument la chaîne de caractères représentant ce littéral et renvoie la valeur flottante double précision.

7.6.1.4. Fonctions d'initialisation d'une donnée

Afin de représenter la clause **VALUE** et l'instruction COBOL **MOVE**, nous utiliserons les cinq fonctions suivantes réalisant l'initialisation d'une donnée à la valeur d'une constante figurative.

7.6.1.4.1. Spécification de la fonction "initzero"

Cette fonction remplit la zone mémoire de la donnée dont on donne l'adresse et la longueur en premier et second arguments avec des caractères "zero". Son troisième argument possède une valeur 1 ou 0 selon le caractère signé ou non de la donnée qu'il faut initialiser. Si cette valeur est égale à 1, le remplissage de la donnée commence à partir de l'adresse spécifiée augmentée de 1 et son signe devient par convention le caractère "+".

7.6.1.4.2. Spécification de la fonction "initspace"

Cette fonction est utilisée afin d'initialiser à la valeur de la constante figurative **SPACE** les données de type alphanumérique. Elle remplit donc la zone dont on donne l'adresse et la longueur avec des caractères "espace".

7.6.1.4.3. Spécification de la fonction "inithighvalue"

Cette fonction est utilisée afin d'initialiser à la valeur de la constante figurative **HIGH-VALUE** les données de type alphanumérique. Elle remplit donc la zone dont on donne l'adresse et la longueur avec des caractères "**highvalue**" (cfr 7.4.1) représentant le caractère le plus haut (tous ses bits sont à 1) dans la séquence de classement de l'alphabet du langage C.

7.6.1.4.4. Spécification de la fonction "initlowvalue"

Cette fonction sert à initialiser une donnée de type alphanumérique à la valeur de la constante figurative **LOW-VALUE** et remplit de caractères "**lowvalue**" (cfr 7.4.1) représentant le caractère le plus bas (tous ses bits sont à 0) dans la séquence de classement de l'alphabet du langage C la zone dont on donne l'adresse et la longueur.

7.6.1.4.5. Spécification de la fonction "initquote"

Cette fonction est utilisée afin d'initialiser une donnée de type alphanumérique à la valeur de la constante figurative **QUOTE**. Elle remplit la zone dont on donne l'adresse et la longueur avec des caractères "guillemet".

7.6.1.5. Spécification de la fonction "move"

Cette fonction apparaissant sous la forme **move(adresse1, longueur1, adresse2, longueur2**) sert à effectuer le cadrage en mémoire d'une donnée de type alphanumérique.

Elle copie caractère par caractère et de gauche à droite le contenu de la donnée dont on donne l'adresse et la longueur en premier et second arguments dans la donnée d'adresse **adresse2** et de longueur **longueur2.** Ce copiage s'effectue selon les règles suivantes. Lorsque **longueur1** est inférieure à **longueur2**, le reste de la donnée réceptrice est rempli avec des caractères "espace"; dans le cas où la première longueur est supérieure à la seconde, seul le nombre de caractères correspondant à la donnée réceptrice est copié.

7.6.1.6. Spécification de la fonction "copier"

Cette fonction apparaît sous la forme **copier(adresse1, adresse2, longueur2**) et copie dans une chaîne de caractères dont on donne l'adresse en premier argument le contenu de la zone mémoire dont on donne l'adresse **adresse2** et la longueur **longueur2.**

7.6.2. Fonctions de comparaison

Ces fonctions sont utilisées afin de représenter les conditions COBOL (cfr 7.7.12).

7.6.2.1. Fonctions de comparaison des données alphanumériques avec une constante figurative

Ces fonctions se basent sur les règles définies par le langage COBOL et servent à comparer la représentation d'une donnée alphanumérique avec la valeur d'une constante figurative.

7.6.2.1.1. Spécification de la fonction "zerocomp"

Cette fonction permet de tester si la zone mémoire dont on donne l'adresse et la longueur est initialisée à la valeur de la constante figurative **ZERO** et renvoie une valeur **true** ou **false** d'après le résultat positif ou non de cette comparaison.

7.6.2.1.2. Spécification de la fonction "spacecomp"

Cette fonction teste si la zone dont on donne l'adresse et la longueur est initialisée à la valeur de la constante figurative **SPACE** et retourne une valeur **true** ou **false**.

7.6.2.1.3. Spécification de la fonction "highcomp"

Cette fonction teste si la zone dont on donne l'adresse et la longueur est initialisée à la valeur de la constante figurative **HIGH-VALUE** et retourne une valeur **true** ou **false**.

7.6.2.1.4. Spécification de la fonction "lowcomp"

Cette fonction teste si la zone dont on donne l'adresse et la longueur est initialisée à la valeur de la constante figurative **LOW-VALUE** et renvoie une valeur **true** ou **false**.

7.6.2.1.5. Spécification de la fonction "quotecomp"

Cette fonction teste si la zone dont on donne l'adresse et la longueur est initialisée à la valeur de la constante figurative **QUOTE** et retourne une valeur **true** ou **false**.

7.6.2.2. Fonctions de comparaison de deux données COBOL

Ces fonctions sont utilisées afin de comparer selon les règles définies par le langage COBOL deux données de type alphanumérique et une donnée numérique avec une donnée alphanumérique.

7.6.2.2.1. Spécification de la fonction "compare"

Cette fonction compare selon les règles définies par le langage COBOL la zone dont on donne l'adresse et la longueur en premier et second arguments à celle dont l'adresse et la longueur sont respectivement les troisième et quatrième arguments à partir du cinquième

argument (une chaîne de deux caractères) spécifiant le type de comparaison à effectuer. Elle retourne ensuite une valeur **true** ou **false** selon le résultat de cette comparaison.

7.6.2.2.2. Spécification de la fonction "compareX9"

Cette fonction est utilisée afin de comparer une donnée numérique avec une de type alphanumérique et tient compte afin de réaliser cette comparaison du facteur de puissance, de la position de la virgule et du signe de la donnée numérique, ainsi que des règles de comparaison définies en COBOL pour ce type de comparaison.

Ses arguments sont dans cet ordre l'adresse et la longueur de la donnée alphanumérique, l'adresse, la longueur en zone mémoire, le facteur de puissance, la position de la virgule et le caractère signé ou non de la donnée numérique, ainsi qu'une chaîne de deux caractères spécifiant le type de comparaison à réaliser.

7.6.2.3. Fonctions de vérification du type d'une donnée

Ces fonctions sont utilisées afin de vérifier l'appartenance d'une donnée à un type et sont utilisées afin de représenter les conditions COBOL lorsque leur format est :

< condition > ::= < identifie > **IS NUMERIC**
| < identifier > **IS NUMERIC**
| < identifier > **IS ALPHABETIC**
| < identifier > **IS NOT ALPHABETIC.**

7.6.2.3.1. Spécification de la fonction "numericid"

Cette fonction teste si la donnée dont passe l'adresse et la longueur en zone mémoire est composée exclusivement des caractères 0,1, ... , 9, excepté son signe éventuel. Si oui, elle retourne la valeur **true**, sinon **false**.

7.6.2.3.2. Spécification de la fonction "alphabeticid"

Cette fonction teste si la donnée dont on passe l'adresse et la longueur en zone mémoire est composée exclusivement des lettres majuscules A, ... , Z, du caractère "espace" et des lettres minuscules a, ... , z. Si oui, elle renvoie la valeur **true**, sinon **false**.

7.6.3. Fonctions utilisées pour la réalisation des instructions ACCEPT et DISPLAY

7.6.3.1. Spécification de la fonction "acceptstr"

Cette fonction a été conçue afin de pallier le fait que l'instruction **scanf** du langage C n'accepte pas les caractères "espace" introduits par l'utilisateur. Elle apparaît sous la forme **acceptstr(a , longueur**) et lit caractère par caractère en utilisant la fonction C **getchar** les caractères introduits par l'utilisateur. Le nombre de ces caractères est limité à la valeur de l'argument **longueur** et sont copiés dans la chaîne de caractères dont on donne l'adresse **a**.

7.6.3.2. Spécification de la fonction "signe"

Cette fonction renvoie le caractère "+" ou "-" représentant le signe d'une donnée numérique et se trouvant à l'adresse spécifiée dans son seul argument.

7.6.3.3. Spécification de la fonction "replacedoublequotes"

Cette fonction est utilisée afin de supprimer dans tout littéral numérique les caractères "guillemet" ne devant pas figurer à l'écran.

Elle remplace dans la chaîne de caractères qui lui est passée en argument les suites de deux caractères "guillemet" successifs par un seul caractère "guillemet" et copie la chaîne de caractères ainsi modifiée dans une autre dont l'adresse est renvoyée.

7.6.3.4. Spécification de la fonction "ntochar"

Cette fonction a été conçue afin de réaliser l'instruction DISPLAY lorsque celle-ci est utilisée afin d'afficher la valeur d'une donnée numérique lorsque la clause DECIMAL-POINT IS COMMA est spécifiée dans un programme COBOL.

Cette fonction apparaît sous la forme suivante : **ntochar(a, adresse, longueur, p , v**) et est utilisée afin de copier dans la chaîne de caractères dont on donne l'adresse **a**, le contenu en mémoire de la donnée dont on donne l'adresse **adresse** et la longueur **longueur** et à laquelle est ajoutée **p** caractères "zero" à droite si l'argument **v** est égal à -1 ou à gauche si la valeur de **v** est 0.

7.6.4. Fonctions de gestion des fichiers COBOL

Comme nous l'avions mentionné dans les paragraphes 7.5.5.2.1 et 7.5.5.2.3, afin de mettre à jour la valeur du file status associé à un fichier, nous utilisons les fonctions décrites dans ce paragraphe. Cette mise à jour sera cependant limitée par les possibilités qu'offre le

langage C car le file status d'un fichier ne sera initialisé à une valeur que dans la mesure où celle-ci peut l'être à partir des valeurs que retournent les fonctions C utilisées. Nous ne reprendrons donc qu'un nombre limité des différentes valeurs spécifiées dans la norme COBOL que peut prendre un file status, l'utilisateur de notre éditeur devra donc avoir connaissance des mises à jour possibles de celui-ci.

7.6.4.1. Spécification de la fonction "fsopen"

Cette fonction apparaît sous la forme **fsopen(efn, mode, afs, fssp)**. Elle réalise la fonction C **fopen** à partir de ses deux premiers arguments (un nom externe de fichier et le mode d'ouverture que l'on désire effectuer pour celui-ci) et met à jour le file status dont on lui donne l'adresse **afs** si la valeur de son argument **fssp** est égale à 1. Cette mise à jour s'effectue selon les règles suivantes. Si **fopen** renvoie la valeur du pointeur nul *NULL* , une erreur quelconque s'est produite et le file status est initialisé à la valeur "**30**", dans le cas contraire, celui-ci possède alors la valeur "**00**" exprimant la réussite de l'opération d'ouverture du fichier. Cette fonction renvoie alors la valeur retournée par **fopen.**

7.6.4.2. Spécification de la fonction "fsread"

Cette fonction apparaît sous la forme **fsread(fname, ab, lb, afs, fssp**) et réalise par la fonction C **read** à partir de ses trois premiers arguments (un pointeur de fichier, l'adresse du buffer où doit être transférée l'information et la longueur de ce dernier) l'opération de lecture dans le fichier et met ensuite à jour le file status dont on lui donne l'adresse **afs** si la valeur de son argument **fssp** est égale à **1**. Cette mise à jour s'effectue selon les conventions suivantes :

- Si **read** renvoie la valeur 0, la fin de fichier a été rencontrée et le file status est mis à "**10**",

- Si **read** renvoie -1, une erreur quelconque s'est produite et le file status devient "**30**",

- Dans les autres cas, la valeur retournée par la fonction **read** représente le nombre de caractères transférés par celle-ci. Si ce nombre est inférieur à la longueur du buffer donné dans l'argument **lb**, le file status est mis à la valeur "**04**" et si égalité il y a, à "**00**".

Cette fonction renvoie alors la valeur retournée par la fonction **read.**

7.6.4.3. Spécification de la fonction "fswrite"

Cette fonction apparaît sous la forme **fswrite(fname, ab, lb, afs, fssp**). Elle réalise à partir des arguments **fname**, **ab** et **lb** (représentant respectivement un pointeur de fichier, l'adresse du buffer d'où provient l'information à transférer et sa longueur) l'opération

d'écriture grâce à la fonction C **write** et met ensuite à jour le file status dont on donne l'adresse **afs** si la valeur de son argument **fssp** est égale à 1 selon la régle suivante :

- Si le nombre de caractères transférés que retourne la fonction **write** est inférieur à la longueur du buffer spécifié dans l'argument **lb**, le file status est mis à la valeur "**04**", sinon à "**00**".

Elle retourne ensuite la valeur renvoyée par la fonction **write**.

7.7. Représentation de la procedure division

7.7.1. Principe de représentation des sections et paragraphes

Comme nous l'avions déjà indiqué dans la section 4, les sections et paragraphes COBOL sont représentés par des fonctions C.

Afin de ne pas déclarer dans le programme C généré plusieurs fois une même fonction, le nom d'une section ou d'un paragraphe est précédé du nom du programme COBOL auquel ils appartiennent et du caractère "_". De plus, à chaque fonction C représentant une section ou un paragraphe est passée la valeur du pointeur **ptralloc** permettant de calculer l'adresse des données utilisées dans les instructions de celle-ci.

La fonction **repr_c_of_proc_div** sert à déclencher l'exécution séquentielle des sections ou paragraphes composant un programme COBOL par appel des fonctions C qui les représentent. Elle permet de guider à la manière d'un programme COBOL le processus d'exécution du programme C généré.

La fonction **repr_c_of_sections** est utilisée afin de générer les fonctions C représentant des sections, ainsi que celles des paragraphes de ceux-ci.

La fonction **rcs(<section>)** génère une fonction C représentant une section COBOL. Cette fonction lors de son appel reçoit comme argument la valeur du pointeur **ptralloc** et déclenche l'exécution séquentielle des paragraphes qui composent la section grâce à la fonction **rcpd** par appel des fonctions C représentant ceux-ci. La fonction **rcp** génère les fonctions C représentant ces paragraphes.

La fonction **repr_c_of_paragraphs** sert à générer les fonctions C représentant les paragraphes d'un programme COBOL lorsque la procedure division de celui-ci est composée de paragraphes et non de sections.

La fonction **rcp(<paragraph>)** génére une fonction C représentant un paragraphe COBOL. Cette fonction reçoit également la valeur du pointeur **ptralloc** et déclenche l'exécution séquentielle des instructions COBOL qui la composent grâce à la fonction **repr_c_of_statements**. Afin de représenter ces dernières, des variables C déclarées dans la fonction **decl_of_variables_for_statements** sont utilisées dans certains cas comme nous le verrons dans la suite.

Ces fonctions sont définies comme suit :

```
repr_c_of_proc_div ( < proc_div > ) =
                              nil    si < proc_div > ::= nil
                        |     repr_c_of_proc_div ( < procdivbody > )
                                     si < proc_div > ::=  < procdivbody >
```

| < procdivbody >
< parameterlist >

repr_c_of_proc_div (<procdivbody >) =
nil si < procdivbody > ::= nil
| rcpd (< sectionlist >)
si < procdivbody > ::= < sectionlist >
| rcpd (< paragraphlist >)
si < procdivbody > ::= < paragraphlist >

rcpd (< sectionlist >$_1$) =
nil si < sectionlist >$_1$ ::= nil
| rcpd (< section >)
rcpd (< sectionlist >$_2$)
si < sectionlist >$_1$ ::= < section >
< sectionlist >$_2$

rcpd (< paragraphlist >$_1$) =
nil si < paragraphlist >$_1$ ::= nil
| rcpd (< paragraph >)
rcpd (< paragraphlist >$_2$)
si < paragraphlist >$_1$ ::= < paragraph >
< paragraphlist >$_2$

rcpd (< section >) =
nil si < section > ::= nil
| progr_name (< id_div >)_< section_name >(**ptralloc** **);**
si < section > ::= < section_name >
< paragraphlist >

rcpd (< paragraph >) =
nil si < paragraph > ::= nil
| progr_name (< id_div >)_< paragraph_name >(**ptralloc** **);**
si < paragraph > ::= < paragraph_name >
< sentencelist >

repr_c_of_sections (< proc_div >) =
nil si < proc_div > ::= nil
| repr_c_of_sections (< procdivbody >)
si < proc_div > ::= < procdivbody >
| < parameterlist >
< procdivbody >

```
repr_c_of_paragraphs ( < proc_div > ) =
                    nil     si < proc_div > ::= nil
            |       repr_c_of_paragraphs ( < procdivbody > )
                            si < proc_div > ::= < procdivbody >
                                            |   < parameterlist >
                                                < procdivbody >

repr_c_of_sections ( < procdivbody > ) =
                    nil     si < procdivbody > ::= nil
                                            | < paragraphlist >
            |       rcs ( < sectionlist > )
                            si < procdivbody > ::= < sectionlist >

repr_c_of_paragraphs ( < procdivbody > ) =
                    nil     si < procdivbody > ::= nil
                                            |  < sectionlist >
            |       rcp ( < paragraphlist > )
                            si < procdivbody > ::= < paragraphlist >

rcs (  < sectionlist >1 ) =
                    nil     si < sectionlist >1 ::= nil
            |       rcs ( < section > )
                    rcs ( < sectionlist >2 )
                            si < sectionlist >1 ::= < section >
                                                    < sectionlist >2

rcp ( < paragraphlist >1 ) =
                    nil     si < paragraphlist >1 ::= nil
            |       rcp ( < paragraph > )
                    rcp ( < paragraphlist >2 )
                            si < sectionlist >1 ::= < paragraph >
                                                    < paragraphlist >2

rcs ( < section > ) =
                    nil     si < section > ::= nil
        |       progr_name ( < id_div > )_< section_name >( ptralloc )
                char *ptralloc;
                {
                rcpd ( < paragraphlist > )
                }
                rcp ( < paragraphlist > )
                            si < section > ::= < section_name >
                                               < paragraphlist >
```

```
rcp ( < paragraph > ) =
                    nil      si < paragraph > ::= nil
       |  progr_name ( < id_div > )_< paragraph_name >( ptralloc )
          char *ptralloc;
          {
          decl_of_variables_for_statements ( < sentencelist > )
          repr_c_of_statements ( < sentencelist > )
          }
                             si < paragraph > ::= < paragraph_name >
                                                  < sentencelist >
```

Exemple :

Soit le programme de nom **prgcob** composé des sections et paragraphes suivants :

```
a SECTION.
    a1.
        ...
    a2.
        ...
b SECTION.
    b1.
        ...
    b2.
        ...
```

La valeur des fonctions définies ci-dessus sont :

repr_c_of_proc_div (< proc_div >) =

```
prgcob_a( ptralloc );
prgcob_b( ptralloc );
```

repr_c_of_sections (< proc_div >) =

```
prgcob_a( ptralloc )
char *ptralloc;
{
prgcob_a1( ptralloc );
prgcob_a2( ptralloc );
}
prgcob_a1( ptralloc )
char *ptralloc;
{
    ...
}
prgcob_a2( ptralloc )
char *ptralloc;
{
    ...
}
prgcob_b( ptralloc )
```

```
char *ptralloc;
{
prgcob_b1( ptralloc );
prgcob_b2( ptralloc );
}
prgcob_b1( ptralloc )
char *ptralloc;
{
        ...
}
prgcob_b2( ptralloc )
char *ptralloc;
{
        ...
}
```

7.7.2. Fonctions de déclaration de variables C et de représentation d'instructions COBOL

decl_of_variables_for_statements (< sentencelist >$_1$) =
nil   si < sentencelist >$_1$ ::= nil
| dvs (< stmtlist >)
dvs (< sentencelist >$_2$)
si < sentencelist >$_1$ ::= < stmtlist >
< sentencelist >$_2$

repr_c_of_statements (< sentencelist >$_1$) =
nil   si < sentencelist >$_1$ ::= nil
| rcs (< stmtlist >)
rcs (< sentencelist >$_2$)
si < sentencelist >$_1$ ::= < stmtlist >
< sentencelist >$_2$

dvs (< stmtlist >$_1$) =
nil   si < stmtlist >$_1$ ::= nil
| dvs (< stmt >)   si < stmtlist >$_1$ ::= < stmt >
| dvs (< stmt >)
dvs (< stmtlist >$_2$)
si < stmtlist >$_1$ ::= < stmt >
< stmtlist >$_2$

rcs (< stmtlist >$_1$) =
nil   si < stmtlist >$_1$ ::= nil
| rcs (< stmt >)   si < stmtlist >$_1$ ::= < stmt >
| rcs (< stmt >)
rcs (< stmtlist >$_2$)
si < stmtlist >$_1$ ::= < stmt >
< stmtlist >$_2$

dvs (< stmt >) = nil si < stmt > ::= nil

rcs (< stmt >) = nil si < stmt > ::= nil

La fonction **rcs(<stmt>)** définie dans les paragraphes suivants nous permettra de représenter une instruction COBOL particulière. La fonction **dvs(<stmt>)** représente les déclarations de variables C générées afin de représenter celle-ci.

7.7.3. Principe de numérotation des variables générées

Le nom des variables générées utilisées dans la représentation d'une instruction COBOL est toujours composé du caractère "**c**" et d'un numéro. Cette numérotation locale à un paragraphe commence à partir de 0 et se poursuit par incrément de 1 à chaque variable devant être générée. Afin d'effectuer celle-ci, nous utilisons une fonction de nom **nvar** représentant le numéro à attribuer à une variable devant être générée, ainsi qu'une fonction **rnvar** représentant lorsque des variables sont générées le numéro de la variable suivante.

Ces fonctions se définissent comme suit :

nvar (< sentencelist >) = 0 si < paragraph > ::= < sentencelist >

Si < sentencelist >$_1$ ::= < stmtlist > < sentencelist >$_2$,
nvar (< stmtlist >) = nvar (< sentencelist >$_1$)
nvar (< sentencelist >$_2$) = rnvar (< stmtlist >)

Si < stmtlist >$_1$ ::= < stmt >
| < stmt >
< stmtlist >$_2$,
nvar (< stmt >) = nvar (< stmtlist >$_1$)
nvar (< stmtlist >$_2$) = rnvar (< stmt >)
rnvar (< stmtlist >$_1$) = rnvar (< stmlist >$_2$)

Les fonctions **nvar**(**<stmt>**) et **rnvar**(**<stmt>**) seront définies dans les paragraphes suivants.

7.7.4. Représentation de la structure COBOL IF

Afin de représenter la structure COBOL **IF**, nous utilisons la structure **if-else** du langage C. Les fonctions utilisées sont les suivantes :

- **rcs(<stmtlist>)** permettant de représenter les instructions COBOL figurant dans les parties < thenstmt > et < elsestmt >,

- **init_of_variables_for_a_condition** (cfr 7.7.12) utilisée pour affecter une valeur aux variables déclarées afin de représenter la condition COBOL,

- **repr_of_a_condition** (cfr 7.7.12) étant la représentation C de la condition.

rcs (< IFStmt >) = rcs (< stmt >)

rcs (< stmt >) =
init_of_variables_for_a_condition (< condition >)
if (repr_of_a_condition (< condition >))
{
rcs (< stmtlist >)
}
si < IFStmt > ::= **IF** < condition > **THEN** <thenstmt >
avec < thenstmt > ::= < stmtlist >
| **IF** < condition > **THEN** < thenstmt >
ELSE NEXT SENTENCE
avec < thenstmt > ::= <stmtlist >

| init_of_variables_for_a_condition (< condition >)
if !(repr_of_a_condition (< condition >))
{
rcs (< stmtlist >)
}
si < IFStmt > ::= **IF** < condition > **THEN NEXT SENTENCE** **ELSE** < elsestmt >
avec < elsestmt > ::= < stmtlist >

| init_of_variables_for_a_condition (< condition >)
if (repr_of_a_condition (< condition >))
{
;
}
si < IFStmt > ::= **IF** < condition > **THEN NEXT SENTENCE**

| init_of_variables_for_a_condition (< condition >)
if (repr_of_a_condition (< condition >))
{
rcs (< stmtlist >$_1$)
}
else
{
rcs (< stmtlist >$_2$)
}

si < IFStmt > ::= < condition >< thenstmt >< elsestmt >
avec < thenstmt > ::= < stmtlist >$_1$
et < elsestmt > ::= < stmtlist >$_2$

dvs (< stmt >) = dvs (< condition >)
dvs (< thenstmt >)
dvs (< elsestmt >)

nvar (< condition >) = nvar (< stmt >)
nvar (< thenstmt >) = rnvar (< condition >)
nvar (< elsestmt >) = rnvar (< thenstmt >)
rnvar (< stmt >) = rnvar (< elsestmt >)

Si < thenstmt > ::= **NEXT SENTENCE**,
dvs (< thenstmt >) = nil
rnvar (< thenstmt >) = nvar (< thenstmt >)

Si < thenstmt > ::= < stmtlist >,
dvs (< thenstmt >) = dvs (< stmtlist >)
rnvar (< thenstmt >) = rnvar (< stmtlist >)
nvar (< stmtlist > = nvar (< thenstmt >)

Si < elsestmt > ::= **NEXT SENTENCE**,
dvs (< elsestmt >) = nil
rnvar (< elsestmt >) = nvar (< elsestmt >)

Si < elsestmt > ::= < stmtlist >,
dvs (< elsestmt >) = dvs (< stmtlist >)
rnvar (< elsestmt >) = rnvar (< stmtlist >)
nvar (< stmtlist >) = nvar (< elsestmt >)

7.7.5. Représentation des instructions arithmétiques ADD, SUBTRACT, MULTIPLY et DIVIDE

Afin de représenter les instructions arithmétiques, nous utilisons les fonctions C **dton** (cfr 7.6.1.1), **ntod** (cfr 7.6.1.2.2) et **codelitnumtodouble** (cfr 7.6.1.3).

Les opérations arithmétiques spécifiées par ces instructions sont effectuées en C sur des valeurs flottantes double précision. Leurs résultats sont ensuite cadrés en mémoire par la fonction **dton**.

Le passage par des valeurs flottantes double précision est dû à la représentation en zone mémoire des données numériques selon l'usage DISPLAY et au désir de pouvoir effectuer ces opérations sur des valeurs de données qui pourront grâce à ce choix être les plus grandes possibles.

Avant chacune de celles-ci, la variable **size_error** (cfr 7.4.1 et 7.6.1.1) est initialisée à O afin de pouvoir ensuite tester son éventuelle mise à 1 dans une fonction **dton** (cfr 7.6.1.1) indiquant une troncature dans le cadrage d'une donnée résultat en mémoire.

Pour toute instruction arithmétique, on génère autant d'appels à la fonction **dton** qu'il y a d'éléments spécifiés dans sa <result_list>. Les éléments de celle-ci représentent les données résultats dont la valeur doit être cadrée en mémoire.

Les fonctions **nvar**, **rnvar** et **dvs** de ces instructions sont les suivantes :

nvar (< size_errorstmt >) = nvar (< stmt >)
nvar (< notsize_errorstmt >) = rnvar (< size_errorstmt >)
rnvar (< stmt >) = rnvar (< notsize_errorstmt >)
dvs (< stmt >) = dvs (< size_errorstmt >)
dvs (< notsize_errorstmt >)

7.7.5.1. Représentation de l'instruction ADD

a) Cas où <ADDStmt> ::= <oplist1> <result_list> <size_errorstmt> <notsize_errorstmt>

rcs (< ADDStmt >) = rcs (< stmt >)

rcs (< stmt >) = **size_error = 0;**
dton_instruction_list (< result_list >)
repr_of_size_error_stmts (< stmt >)

dton_instruction_list (< result_list >$_1$) =
dton_instruction_add (< result >)
si < result_list >$_1$ ::= < result >
| dton_instruction_add (< result >)
dton_instruction_list (< result_list >$_2$)
si < result_list >$_1$ ::= < result >
< result_list >$_2$

dton_instruction_add (< result >) =
dton(attr_of_id (< identifier >)**, 0, &size_error,**
(ntod(attr_of_id (< identifier >)) calcul_of_double_value_add (< oplist1 >) **));**
si < result > ::= < identifier >

```
        |   dton( attr_of_id ( < identifier > ), 1, &size_error, ( ntod( attr_of_id
            ( < identifier> ) ) calcul_of_double_value_add ( < oplist1 > ) ));
                        si < result > ::= < identifier > ROUNDED

calcul_of_double_value_add ( < oplist1 >1 ) =
            ntod_function_add ( < id_or_litnum > )
                        si < oplist1 >1 ::= < id_or_litnum >
        |   ntod_function_add ( < id_or_litnum > )
            calcul_of_double_value_add ( < oplist1 >2 )
                        si < oplist1 >1 ::= < id_or_litnum >
                                            < oplist1 >2

ntod_function_add ( < id_or_litnum > ) =
            + ntod( attr_of_id ( < identifier > ) )
                        si < id_or_litnum > ::= < identifier >
        |   + codelitnumtodouble ( "< numeric_litteral >" );
                        si < id_or_litnum > ::= < numeric_literal >

attr_of_id ( < identifier > ) = adresse ( < identifier > ),
                                np ( < identifier > ),
                                p ( < identifier > ),
                                v ( < identifier > ),
                                s ( < identifier > )

np ( < identifier > ) = ENVDATA ( name = < userdef_name >, np )
p ( < identifier > )  = ENVDATA ( name = < userdef_name >, p )
v ( < identifier > )  = ENVDATA ( name = < userdef_name >, v )
s ( < identifier > )  = ENVDATA ( name = < userdef_name >, s )
```

Exemple :

Soit l'instruction : ADD a,b TO c, d.

Sa représentation en C est :

```
size_error = 0;
dton( attr_of_id( c ), 0, &size_error, ( ntod( attr_of_id( c ) ) +
ntod( attr_of_id( a ) ) + ntod( attr_of_id( b ) ) ));
dton( attr_of_id( d ), 0, &size_error, ( ntod( attr_of_id( d ) ) +
ntod( attr_of_id( a ) ) + ntod( attr_of_id( b ) ) ));
```

b) Cas où <ADDStmt> ::= <oplist1> <id_or_litnum> <result_list> <size_errorstmt> <notsize_errorstmt>

rcs (< ADDStmt >) = rcs (< stmt >)

rcs (< stmt >) =
size_error = 0;
dton_function_list (< result_list >)
(double_value_for_op (< id_or_litnum >)
calcul_of_double_value_add (< oplist1 >))
parentheses (< result_list >);
repr_of_size_error_stmts (< stmt >)

dton_function_list (< result_list >$_1$) =
dton_function (< result >)
si < result_list >$_1$ ::= < result >
| dton_fuction (< result >)
dton_function_list (< result_list >$_2$)
si < result_list >$_1$ ::= < result >
< result_list >$_2$

dton_function (< result >) =
dton(attr_of_id (< identifier >), **0, &size_error,**
si < result > ::= < identifier >
| **dton(** attr_of_id (< identifier >), **1, &size_error,**
si < result > ::= < identifier > **ROUNDED**

double_value_for_op (< id_or_litnum >) =
ntod(attr_of_id (< identifier >))
si < id_or_litnum > ::= < identifier >
| **codelitnumtodouble(** "< numeric_literal >")
si < id_or_litnum > ::= < numeric_literal >

parentheses (< result_list >$_1$) =
parenthese (< result >)
si < result_list >$_1$ ::= < result >
| parenthese (< result >)
parentheses (< result_list >$_2$)
si < result_list >1 ::= < result >
< result_list >$_2$

parenthese (< result >) = ")" si < result > ::= < identifier >
| <identifier > **ROUNDED**

Exemple :

Soit l'instruction : ADD a, b TO c GIVING d, e.

Sa représentation est :

size_error = 0;
dton(attr_of_id(d), 0, &size_error, dton(attr_of_id(e), 0, &size_error, (ntod(attr_of_id(a)) + ntod(attr_of_id (b)) + ntod(attr_of_id(c)))));

7.7.5.2. Représentation de l'instruction SUBTRACT

a) Cas où <SUBTRACTStmt> ::= <oplist1> <result_list> <size_errorstmt> <notsize_errorstmt>

rcs (< SUBTRACTStmt >) = rcs (< stmt >)

rcs (< stmt >) = **size_error = 0;**
dton_instruction_list (< result_list >)
repr_of_size_error_stmts (< stmt >)

dton_instruction_list (< result_list $>_1$) =
dton_instruction_sub (< result >)
si < result_list $>_1$ ::= < result >
| dton_instruction (< result >)
dton_instruction_list (< result_list $>_2$)
si < result_list $>_1$:= < result> < result_list $>_2$

dton_instruction_sub (< result >) =
dton(attr_of_id (< identifier >), **0, &size_error,**
(ntod(attr_of_id (< identifier >)) calcul_of_double_value_sub (< oplist1 >) **));**
si < result > ::= < identifier >
| **dton(** attr_of_id (< identifier >), **1, &size_error,**
(ntod(attr_of_id (< identifier >)) calcul_of_double_value_sub (< oplist1 >) **));**
si < result > ::= < identifier > **ROUNDED**

calcul_of_double_value_sub (< oplist1 $>_1$) =
ntod_function_sub (< id_or_litnum >)
si < oplist1 $>_1$ ::= < id_or_litnum >
| ntod_function_sub (< id_or_litnum >)
calcul_of_double_value_sub (< oplist1 $>_2$)

si < oplist1 >$_1$ ::= < id_or_Litnum >
< oplist1 >$_2$

ntod_function_sub (< id_or_litnum >) =
- **ntod** (attr_of_id (< identifier >))
si < id_or_litnum > ::= < identifier >
| - **codelitnumtodouble** ("< numeric_literal >")
si < id_or_litnum > ::= < numeric_literal >

Exemple :

Soit l'instruction : SUBTRACT a, b FROM c, d.

Sa représentation est :

```
size_error = 0;
dton( attr_of_id( c ), 0, &size_error, ( ntod( attr_of_id( c )) -
ntod( attr_of_id( a ) ) - ntod( attr_of_id( b ) ) ));
dton( attr_of_id( d ), 0, &size_error, ( ntod( attr_of_id( d )) -
ntod( attr_of_id( a ) ) - ntod( attr_of_id( b ) ) ));
```

b) Cas où <SUBTRACTStmt> ::= <oplist1> <id_or_litnum> <result_list> <size_errorstmt> <notsize_errostmt>

rcs (< SUBTRACTStmt >) = rcs (< stmt >)

rcs (< stmt >) = **size_error = 0;**
dton_function_list (< result_list >)
(double_value_for_op (< id_or_litnum >)
calcul_of_double_value_sub (< oplist1 >))
parentheses (< result_list >)**;**
repr_of_size_error_stmts (< stmt >)

Exemple :

Soit l'instruction : SUBTRACT a, b FROM c GIVING d, e.

Sa représentation est :

```
size_error = 0;
dton( attr_of_id( d ), 0, &size_error, dton( attr_of_id ( e ), 0,
&size_error, ( ntod( attr_of_id( c ) ) - ntod( attr_of_id( a )) -
ntod( attr_of_id( b ) ) ) ));
```

7.7.5.3. Représentation de l'instruction MULTIPLY

a) Cas où <MULTIPLYStmt> ::= <id_or_litnum> <result_list> <size_errorstmt> <notsize_errorstmt>

rcs (< MULTIPLYStmt >) = rcs (< stmt >)

rcs (< stmt >) = **size_error = 0;**
dton_instruction_list (< result_list >)
repr_of_size_error_stmts (< stmt >)

dton_instruction_list (< result_list >$_1$) =
dton_instruction_mult (< result >)
si < result_list >$_1$ ::= < result >
| dton_instruction_mult (< result >)
dton_instruction_list (< result_list >$_2$)
si < result_list >$_1$ ::= < result > < result_list >$_2$

dton_instruction_mult (< result >) =
dton(attr_of_id (< identifier >)**, 0, &size_error,**
(**ntod(** attr_of_id (< identifier >) **)*double_value_for_op** (< id_or_litnum >) **));**
si < result > ::= < identifier >
| **dton(** attr_of_id (< identifier >)**, 1, &size_error,**
(**ntod(** attr_of_id (< identifier >)**)*** double_value_for_op (< id-or_litnum >) **));**
si < result > ::= < identifier > **ROUNDED**

Exemple :

Soit l'instruction : MULTIPLY a BY c, d.

Sa représentation est :

```
size_error = 0;
dton( attr_of_id( c ), 0, &size_error, ( ntod( attr_of_id( c ) )*
ntod( attr_of_id( a ) ) ) );
dton( attr_of_id( d ), 0, &size_error, ( ntod( attr_of_id( d ) )*
ntod( attr_of_id( a ) ) ) );
```

b) Cas où <MULTIPLYStmt> ::= <id_or_litnum>$_1$<id_or_litnum>$_2$ <result_list> <size_errorstmt> <notsize_errorstmt>

rcs (< MULTIPLYStmt >) = rcs (< stmt >)

rcs (< stmt >) = **size_error = 0;**
dton_function_list (< result_list >)
(double_value_for_op (< id_or_litnum >$_1$)*
double_value_for_op (< id_or_litnum > $_2$))
parentheses (< result_list >);
repr_of_size_error_stmts (< stmt >)

Exemple :

Soit l'instruction : MULTIPLY a BY b GIVING c, d.

Sa représentation est :

```
size_error = 0;
dton( attr_of_id( c ), 0, &size_error, dton( attr_of_id( d ), 0,
&size_error, ( ntod( attr_of_id( a ) )*ntod( attr_of_id( b ) ) )));
```

7.7.5.4. Représentation de l'instruction DIVIDE

rcs (< DIVIDEStmt >) = rcs (< stmt >)

rcs (< stmt >) = **size_error = 0;**
dton_function_list (< result_list >)
(double_value_for_op (< id_or_litnum >$_1$)/
double_value_for_op (< id_or_litnum >$_2$))
parentheses (< result_list >);
repr_of_size_error_stmts (< stmt >)

Exemple :

Soit l'instruction : DIVIDE a BY b GIVING c, d.

Sa représentation est :

```
size_error = 0;
dton( attr_of_id( c ), 0, &size_error, dton( attr_of_id( d ), 0,
&size_error, ( ntod( attr_of_id( a ) )/ntod( attr_of_id( b ) ) ) ));
```

7.7.5.5. Représentation des clauses ON SIZE ERROR et NOT ON SIZE ERROR

Le principe utilisé ici consiste à tester la valeur 1 ou 0 de la variable **size_error** et de déclencher en fonction de celle-ci les instructions soit de la clause **ON SIZE ERROR**, soit de la clause **NOT ON SIZE ERROR**, par utilisation de la structure C **if-else**.

repr_of_size_error_stmts (< stmt >) =
if (size_error = 1)
{
rcs (< stmtlist >)
}
si < size_errorstmt > ::= < stmtlist >
et < notsize_errorstmt > ::= nil

| **if (size_error = 0)**
{
rcs (< stmtlist >)
}
si < size_errorstmt > ::= nil
et < notsize_errorstmt > ::= < stmtlist >

| **if (size_error = 1)**
{
rcs (< stmtlist >$_1$)
}
else
{
rcs (< stmtlist >$_2$)
}
si < size_errorstmt > ::= < stmtlist >$_1$
et < notsize_errorstmt > ::= < stmtlist >$_2$

| nil
si < size_errorstmt > ::=nil
et < notsize_errorstmt > ::= nil

Exemple :

Soit l'instruction :
ADD a TO b GIVING c
ON SIZE ERROR PERFORM x
NOT ON SIZE ERROR PERFORM y,

figurant dans un programme de nom **pgmcob**.

Sa représentation est :

```
size_error = 0;
dton( attr_of_id ( c ), 0, &size_error, ( ntod( attr_of_id( a ) )
+ ntod( attr_of_id( b ) ) );
if ( size_error = 1 )
{
pgmcob_x( ptralloc );
}
else
{
pgmcob_y( ptralloc );
}
```

Les fonctions **rnvar** et **dvs** de ces clauses sont les suivantes :

Si < size_error > ::= nil,
rnvar (< size_errorstmt >) = nvar (< size_errorstmt >)
dvs (< size_errorstmt >) = nil

Si < size_error > ::= < stmtlist >,
rnvar (< size_errorstmt >) = rnvar (< stmtlist >)
nvar (< stmtlist >) = nvar (< size_errorstmt >)
dvs (< size_errorstmt >) = dvs (< stmtlist >)

Si < notsize_errorstmt > ::= nil,
rnvar (< notsize_errorstmt >) = nvar (< notsize_errorstmt >)
dvs (< notsize_errorstmt >) = nil

Si < notsize_errorstmt > ::= < stmtlist >,
rnvar (< notsize_errorstmt >) = rnvar (< stmtlist >)
nvar (< stmtlist >) = nvar (< notsize_errorstmt >)
dvs (< notsize_errorstmt >) = dvs (< stmtlist >)

7.7.6. Représentation des instructions ACCEPT et DISPLAY

Nous utilisons en C afin de représenter les instructions ACCEPT et DISPLAY, les fonctions C d'entrées sorties avec format appelées **scanf** et **printf**, ainsi que les fonctions décrites dans les paragraphes 7.6.3 et 7.6.1.2.

Les deux fonctions **scanf** (pour l'entrée) et **prinf** (pour la sortie) permettent la traduction de quantités numériques en chaînes de caractères et vice-versa, ainsi que l'interprétation ou la génération de lignes mises avec format.

La fonction **scanf** peut être décrite de la manière suivante :

scanf(control, arg$_1$, arg$_2$, ...).

Elle lit les caractères introduits à l'entrée standard, les interprète selon le format précis décrit dans la chaîne de caractères control et mémorise les résultats en fonction des arguments qui restent; ceux-ci sont les adresses indiquant l'endroit où doivent être mémorisées les entrées.

La fonction **printf** peut être décrite quant à elle de la manière suivante :

printf(control, arg_1, arg_2, ...).

Elle convertit, met en forme et imprime ses arguments sur la sortie standard sous le contrôle de la chaîne de caractères **control**; les arguments arg_1, arg_2, etc. sont des variables représentant les données à imprimer.

7.7.6.1. Représentation de l'instruction ACCEPT

Lorsque la donnée <identifier> figurant dans l'instruction ACCEPT est de type alphanumérique, nous utilisons la fonction **acceptstr** (cfr 7.6.3.1) afin de copier les caractères introduits dans une variable de type *char* , et la fonction **move** (cfr 7.6.1.5) pour cadrer la valeur de cette variable dans la donnée.

Lorsque la donnée <identifier> est de type numérique, nous calculons le nombre de caractères maximum que peut entrer l'utilisateur afin d'introduire la valeur de celle-ci (fonction **lpic**) et l'utilisons dans la chaîne de control de l'instruction **scanf**. La valeur entrée est ensuite mémorisée dans une variable de type *double* et cadrée dans la donnée par la fonction **dton**.

lpic (< stmt >) = np (< identifier >) + 1
si v (< identifier >) > 0
| np (< identifier >) + p (< identifier >) + 2
si v (< identifier >) = 0
| np (< identifier >) + p (< identifier >)
si v (< identifier >) = -1

pic (< identifier >) = ENVDATA (name = < userdef_name >, pic)

rcs (< ACCEPTStmt >) = rcs (< stmt >)

rcs (< stmt >) =
acceptstr (np (< identifier >), cnvar(< stmt >));
move (cnvar (< stmt >), **strlen**(cnvar (< stmt >)), adresse (< identifier >), np (< identifier >));
si pic (< identifier >) = "**X**"

```
|
scanf ( " % lpic ( < stmt > )lf", &cnvar ( < stmt > ) );
dton ( attr_of_id ( < identifier > ), 0, &size_error, cnvar ( < stmt > ) );
        si pic ( < identifier > ) = "9"

rnvar ( < stmt > ) = nvar ( < stmt > ) + 1

dvs ( < stmt > ) = double cnvar ( < stmt > );
                    si pic ( < identifier > ) = "9"
            |     char cnvar ( < stmt > )[ np ( < identifier > ) ];
                    si pic ( < identifier > ) = "X"
```

7.7.6.2. Représentation de l'instruction DISPLAY

Les éléments de l'instruction DISPLAY pouvant être des données, des littéraux ou des constantes figuratives, il convient de préciser chacun de ces cas.

Dans le cas où un littéral numérique est spécifié, celui-ci figure dans la chaîne de contrôle de la fonction **printf** et aucun argument n'est spécifié.

Dans le cas où un littéral alphanumérique est spécifié, la fonction **replacedoublequotes** (cfr 7.6.3.3) est utilisée afin de modifier le contenu de ce littéral. Le littéral modifié est copié (fonction **rdqbqf**) dans une variable de type *char* utilisée comme argument de la fonction **printf**.

Dans le cas où une donnée est de type alphanumérique, une variable de type *char* dans laquelle est copiée la valeur de la donnée en utilisant la fonction **copier** (cfr 7.6.1.6) (fonction **cf**) sert d'argument à la fonction **printf**.

Dans le cas où une donnée est de type numérique et que la clause **DECIMAL-POINT IS COMMA** a été spécifiée, nous nous arrangeons (fonction **ntof**) en utilisant les fonctions **signe**, **ntoi** et **ntochar** (cfr section 7.6) pour imprimer séparément le signe, la partie entière, un caractère "," et la partie décimale de cette donnée. Pour ce faire, des variables de type *char* et *long* sont générées.

Dans le cas où une donnée est de type numérique et que la clause **DECIMAL-POINT IS COMMA** n'a pas été spécifiée, nous copions avec la fonction **ntod** (cfr 7.6.1.2.1) (fonction **ntof**) la valeur de cette donnée dans une variable de type *double* que nous imprimons ensuite.

Dans le cas où une constante figurative est spécifiée, le caractère représentant celle-ci figure dans la chaîne de contrôle, excepté dans le cas où cette constante est **HIGH-VALUE** ou **LOW-VALUE** pour laquelle les variables **highvalue** et **lowvalue** (cfr 7.4) sont utilisées comme arguments de la fonction **printf.** Dans ces deux derniers cas cependant, l'utilisateur

de notre éditeur pourra constater qu'aucun caractère n'est affiché; la raison de ce phénomène est simplement que les caractères **highvalue** et **lowvalue** sont non éditables.

Les principales fonctions utilisées pour représenter cette instruction sont les suivantes :

- la fonction **control_chains_for_id_or_lit** sert à spécifier la chaîne de contrôle de la fonction **printf**;

- la fonction **variables** sert à spécifier ses arguments;

-les fonctions **nto_functions**, **copier_functions** et **replace_double_quotes_by_quote_functions** sont utilisées afin d'initialiser les variables spécifiées comme arguments de **printf**;

- la fonction **dpic** possédant une valeur 1 ou 0 représente la spécification ou non de la clause **DECIMAL-POINT IS COMMA.**

```
rcs ( < DISPLAYStmt > ) = rcs ( < stmt > )

rcs ( < stmt > ) =
                nto_functions ( < id_or_lit_list > )
                copier_functions ( < id_or_lit_list >)
                replace_double_quotes_by_quote_functions ( < id_or_lit_list > )
                printf ( "control_chains_for_id_or_lit ( < id_or_lit_list > )\n",
                variables ( < id_or_lit_list > ) );
                        si dvs ( < id_or_lit_list > ) <> nil
            |   printf ( "control_chains_for_id_or_lit ( < id_or_lit_list > )\n" );
                        si dvs ( < id_or_lit_list > ) = nil

dvs ( < stmt > ) = dvs ( < id_or_lit_list > )

rnvar ( < stmt > ) = rnvar ( < id_or_lit_list > )
nvar ( < id_or_lit_list > ) = nvar ( < stmt > )

Si < id_or_lit_list > ::= < id_or_lit >,
        nvar ( < id_or_lit > ) = nvar ( < id_or_lit_list > )
        rnvar ( < id_or_lit_list > ) = rnvar ( < id_or_lit > )
        dvs ( < id_or_lit_list > ) = dvs ( < id_or_lit > )
        nto_functions ( < id_or_lit_list > ) =
                ntof ( < id_or_lit > )
        copier_functions ( < id_or_lit_list > ) =
                cf ( < id_or_lit > )
        replace_double_quotes_by_quote_functions ( < id_or_lit_list > ) =
                rdqbqf ( < id_or_lit > )
```

variables (< id_or_lit_list >) = var (< id_or_lit >)
control_chains_for_id_or_lit (< id_or_lit_list >) = cc (< id_or_lit >)

Si < id_or_lit_list >$_1$ ::= < id_or_lit > < id_or_lit_list >$_2$,
nvar (< id_or_lit >) = nvar (< id_or_lit_list >$_1$)
nvar (< id_or_lit_list >$_2$) = rnvar (< id_or_lit >)
rnvar (< id_or_lit_list >$_1$) = rnvar (< id_or_lit_list >$_2$)
dvs (< id_or_lit_list >$_1$) = dvs (< id_or_lit >)
dvs (< id_or_lit_list >$_2$)
nto_functions (< id_or_lit_list >$_1$)
= ntof (< id_or_lit >)
nto_functions (< id_or_lit_list >$_2$)
copier_functions (< id_or_lit_list >$_1$)
= cf (< id_or_lit >)
copier_functions (<id_or_lit_list>$_2$)
replace_double_quotes_by_quote_functions (< id_or_lit_list >$_1$)
= rdqbqf (< id_or_lit >)
replace_double_quotes_by_quote_functions (< id_or_lit_list >$_2$)
variables (< id_or_lit_list >$_1$)
=
variables (< id_or_lit_list >$_2$)
si < id_or_lit > ::= < numeric_literal >
| **ZERO**
| **SPACE**
| **QUOTE**
| var (< id_or_lit >),
variables (< id_or_lit_list >$_2$)
control_chains_for_id_or_lit (< id_or_lit_list >$_1$)
= cc (< id_or_lit >)
control_chains_for_id_or_lit (< id_or_lit_list >$_2$)

dpic (< spec_names >) = 1
si < spec_names > ::= **SPECIAL-NAMES.**
DECIMAL-POINT IS COMMA.
| 0
si < spec_names > ::= **SPECIAL-NAMES.**

nvar2 (< id_or_lit >) = nvar (< id_or_lit >) + 1

Si < id_or_lit > ::= < identifier >,
si pic (< identifier >) = "**9**",
l1 (< id_or_lit >) =
v (< identifier >) + s (< identifier >)
si v (< identifier >) > 0
| s (< identifier >) + 1

```
l2 ( < id_or_lit > ) =
        np ( < identifier > ) - l1 ( < id_or_lit > )
              si v ( < identifier > ) > 0
      | p ( < identifier > ) + ( np ( < identifier > )
        - s ( < identifier > ) )
cc ( < id_or_lit > ) =
        %d   si v ( < identifier > ) = -1
      | %l1( < id_or_lit > ).l2 ( < id_or_lit > )lf
              si v ( < identifier > ) <> -1
              et dpic ( < spec_names > ) = 0
      | %d,%d
              si v ( < identifier > ) > 0
              et dpic ( < spec_names > ) = 1
      | 0,%s
              si v ( < identifier > ) = 0
              et s ( < identifier > ) = 0
              et dpic ( < spec_names > ) = 1
      | %c0,%s
              si v ( < identifier > ) = 0
              et s ( < identifier > ) = 1
              et dpic ( < spec_names > ) = 1
dvs ( < id_or_lit > ) =
        long cnvar ( < id_or_lit > );
              si v ( < identifier > ) = -1
      | long cnvar ( < id_or_lit > );
        long cnvar2 ( < id_or_lit > );
              si v ( < identifier > ) > 0
              et dpic ( < spec_names > ) = 1
      | char cnvar ( < id_or_lit > );
        char cnvar2 (< id_or_lit >)[ l2 (< id_or_lit > ) ];
              si dpic ( < spec_names > ) = 1
              et v ( < identifier > ) = 0
              et s ( < identifier > ) = 1
      | char cnvar ( <id_or_lit> )[ l2 ( < id_or_lit > ) ];
              si dpic ( < spec_names > ) = 1
              et v ( < identifier > ) = 0
              et s ( < identifier > ) = 0
      | double cnvar ( < id_or_lit > );
cf ( < id_or_lit > ) = nil
ntof ( < id_or_lit > ) =
        cnvar ( < id_or_lit > ) = ntod( attr_of_id
        ( < identifier > ) );
              si dpic ( < spec_names > ) = 0
              et v ( < identifier > ) <> -1
```

```
            | cnvar ( < id_or_lit > ) = ntoi( attr_of_id
              ( < identifier > ) );
                    si v ( < identifier > ) = -1
            | cnvar ( < id_or_lit > ) = ntoi( adresse
              ( < identifier > ), l1 ( < id_or_lit > ), 0, -1,
              s ( < identifier > ) );
              cnvar2 ( < id_or_lit > ) = ntoi( adresse
              ( < identifier > ) + l1 ( < id_or_lit > ), l2
              ( < id_or_lit > ), 0, -1, 0 );
                    si dpic ( < spec_names > ) = 1
                    et v ( < identifier > ) > 0
            | ntochar ( cnvar ( < id_or_lit > ), adresse
              ( < identifier > ), np ( < identifier > ), p
              ( < identifier > ), v ( < identifier > ) );
                    si dpic ( < spec_names > ) = 1
                    et v ( < identifier > ) = 0
                    et s ( < identifier > ) = 0
            | cnvar ( < id_or_lit > ) = signe ( adresse
              ( < identifier > ) );
              ntochar ( cnvar2 ( < id_or_lit > ), adresse
              ( < identifier > ) + 1, np ( < identifier > ), p
              ( < identifier > ), v ( < identifier > ) );
                    si dpic ( < spec_names > ) = 1
                    et v ( < identifier > ) = 0
                    et s ( < identifier > ) = 1
si pic ( < identifier > ) = "X",
        cc ( < id_or_lit > ) = %s
        dvs ( < id_or_lit > ) =
              char cnvar ( <id_or_lit> )[ np ( < identifier > ) ];
        cf ( < id_or_lit > ) = copier( cnvar ( < id_or_lit > ), adresse
                             ( <identifier>), np ( < identifier > ) );
        ntof ( < id_or_lit > ) = nil
var ( < id_or_lit > ) =
              cnvar ( < id_or_lit > ), cnvar2 ( < id_or_lit > )
                    si pic ( < identifier > ) = "9"
                    et dpic ( < spec_names > ) = 1
                    et ( v ( < identifier > ) > 0 ou ( v ( <
                           identifier > ) = 0 et s ( < identifier > ) = 1 ) )
            | cnvar ( < id_or_lit > )
rnvar ( < id_or_lit > ) =
              nvar ( < id_or_lit > ) + 2
                    si pic ( < identifier > ) = "9"
                    et dpic ( < spec_names > ) = 1
                    et ( v ( < identifier > ) > 0 ou
                    ( v ( < identifier > ) = 0 et s ( < identifier > ) = 1 ) )
```

| nvar (< id_or_lit >) + 1
rdqbqf (< id_or_lit >) = nil

Si < id_or_lit > ::= < alphanum_literal >,
cc (< id_or_lit >) = **%s**
ntof (< id_or_lit >) = nil
rdqbqf (< id_or_lit >) =
strcpy(cnvar (< id_or_lit >)**, replacedoublequotes("**alvalue (< alphanum_literal >)**");**
dvs (< id_or_lit >) =
char cnvar (< id_or_lit >)[allongueur (< alphanum_literal >) **];**
cf (< id_or_lit >) = nil
var (< id_or_lit >) = cnvar (< id_or_lit >)
rnvar (< id_or_lit >) = nvar (< id_or_lit >) + 1

alvalue (< alphanum_literal >) = < str >
allongueur (< alphanum_literal >) = strlen (< str >)

Si < id_or_lit > ::= < numeric_literal >,
cc (< id_or_lit >) = nlvalue (< numeric_literal >)
ntof (< id_or_lit >) = nil
rdqbqf (< id_or_lit >) = nil
dvs (< id_or_lit >) = nil
cf (< id_or_lit >) = nil
var (< id_or_lit >) = nil
rnvar (< id_or_lit >) = nvar (< id_or_lit >)

nlvalue (< numeric_literal >) = < str >

Si < id_or_lit > ::= < fig_constant >,
cc (< id_or_lit >) =
"**0**" si < fig_constant > ::= **ZERO**
" " si < fig_constant > ::= **SPACE**
'**"**' si < fig_constant > ::= **QUOTE**
%c si < fig_constant > ::= **HIGH-VALUE**
| **LOW-VALUE**
ntof (< id_or_lit >) = nil
rdqbqf (< id_or_lit >) = nil
var (< id_or_lit >) =
highvalue si < fig_constant > ::= **HIGH-VALUE**
| **lowvalue** si < fig_constant > ::= **LOW-VALUE**
| nil
cf (< id_or_lit > = nil
dvs (< id_or_lit >) = nil
rnvar (< id_or_lit >) = nvar (< id_or_lit >)

7.7.7. Représentation des instructions CLOSE, OPEN, READ et WRITE

Nous rappelerons ici seulement que les instructions d'opérations sur les fichiers sont représentées par les fonctions **fclose**, **fsopen**, **fsread** et **fswrite** décrites dans le paragraphe 7.6.4.

7.7.7.1. Représentation de l'instruction CLOSE

L'instruction CLOSE est représentée par la fonction C **fclose** (cfr 7.5.4.2.2).

rcs (< CLOSEStmt >) = rcs (< stmt >)

dvs (< stmt >) = nil
rnvar (< stmt >) = nvar (< stmt >)

rcs (< stmt >) = close_statements (< filename_list >)

close_statements (< filename_list >$_1$) =
close_statement (< file_name >)
si < filename_list >$_1$ ::= < file_name >
| close_statement (< file_name >)
close_statements (< filename_list >$_2$)
si < filename_list >$_1$ ::= < file_name >
< filename_list >$_2$

close_statement (< file_name >) =
fclose(progr_name (< id_div >)_< userdef_name > **);**

7.7.7.2. Représentation de l'instruction OPEN

L'instruction OPEN est représentée par la fonction **fsopen** (cfr 7.5.4.2.1 et 7.6.4.1).

rcs (< OPENStmt >) = rcs (< stmt >)

dvs (< stmt >) = nil
rnvar (< stmt >) = nvar (< stmt >)

rcs (< stmt >) = open_files (< openlist >)

open_files (< openlist >$_1$) =
open_files (< openitem >)
si < openlist >$_1$ ::= < openitem >

| open_files (< openitem >)
open_files (< openlist >$_2$)
si < openlist >$_1$ ::= < openitem >
< openlist >$_2$

open_files (< openitem >) =
open_input_files (< filename_list >)
si < openitem > ::= **INPUT** < filename_list >
| open_output_files (< filename_list >)
si < openitem > ::= **OUTPUT** < filename_list >

open_input_files (< filename_list >$_1$) =
open_input_file (< file_name >)
si < filename_list >$_1$ ::= < file_name >
| open_input_file (< file_name >)
open_input_files (< filename_list >$_2$)
si < filename_list >$_1$ ::= < file_name >
< filename_list >$_2$

open_output_files (< filename_list >$_1$) =
open_output_file (< file_name >)
si < filename_list >$_1$ ::= < file_name >
| open_output_file (< file_name >)
open_output_files (< filename_list >$_2$)
si < filename_list >$_1$ ::= < file_name >
< filename_list >$_2$

open_input_file (< file_name >) =
progr_name (< id_div >)_< userdef_name > **= fsopen**
(efn (< file_name >)**, "r",** afs (< file_name >)**,**
fssp (< file_name >) **);**

open_output_file (< file_name >) =
progr_name (< id_div >)_< userdef_name > **= fsopen**
(efn (< file_name >)**, "w",** afs (< file_name >)**,**
fssp (< file_name >) **);**

efn (< file_name >) = ENVFILES1 (fname = < userdef_name >, efn)
afs (< file_name >) = ENVDATA (name = ENVFILES1 (fname =
< userdef_name >, fsname), adresse)
fssp (< file_name >) = ENVFILES1 (fname = < userdef_name >, fssp)

7.7.7.3. Représentation de l'instruction READ

L'instruction READ est représentée par la fonction **fsread** (cfr 7.5.4.2.3 et 7.6.4.2).

Afin de représenter les clause **AT END** et **NOT AT END**, nous nous basons sur la valeur renvoyée par la fonction **fsread** (cfr 7.6.4.2). Si celle-ci est égale à 0, la fin du fichier a été rencontrée et les instructions de la clause **AT END** sont générées. Si la valeur renvoyée est supérieure à 0, les caractères de la donnée lue dans le fichier ont été transférés et les instructions de la clause **NOT AT END** sont générées. Ces règles s'appliquent évidemment uniquement dans le cas où ces clauses ont été spécifiées pour l'instruction **READ**.

Afin de copier dans la donnée spécifiée dans la clause **INTO** la donnée lue dans le fichier, nous utilisons la fonction **move** (cfr 7.6.1.5).

rcs (< READStmt >) = rcs (< stmt >)

pfname (< stmt >) = **stdin**   si efn (< file_name >) = "**stdin**"
 | progr_name (< id_div >)_< file_name >

rcs (< stmt >) =
fsread(pfname (< stmt >), **ptralloc** + ab (<file_name>),
lb (< file_name >), **ptralloc** + afs (< file_name >),
fssp (< file_name >) **);**
  si < into_id > ::= nil
  et < atend_stmt > ::= nil
  et < notatend_stmt > ::= nil
| **fsread(** pfname (< stmt >), **ptralloc** + ab (<file_name>),
lb (< file_name >), **ptralloc** + afs (< file_name >),
fssp (< file_name >) **);**
move(ptralloc + ab (< file_name >), lb (< file_name >),
adresse (< into_id >), np (< into_id >) **);**
  si < into_id > ::= < identifier >
  et < atend_stmt > ::= nil
  et < notatend_stmt > ::= nil
| **if (fsread(** pfname(< stmt >), **ptralloc** + ab(<file_name>),
lb (< file_name >), **ptralloc** + afs (< file_name >),
fssp (< file_name >)) **= = 0**)
{
rcs (< atend_stmt >)
}
  si < into_id > ::= nil
  et < atend_stmt > ::= < stmtlist >
  et < notatend_stmt > ::= nil

| **if (fsread(** pfname(< stmt >**), ptralloc** + ab(<file_name>),
lb (< file_name > **), ptralloc** + afs (< file_name >),
fssp (< file_name >)) **> 0**)
{
rcs (< notatend_stmt >)
}
si < into_id > ::= nil
et < atend_stmt > ::= nil
et < notatend_stmt > ::= < stmtlist >

| **if (fsread(** pfname(< stmt >**), ptralloc** + ab(<file_name>),
lb (< file_name > **), ptralloc** + afs (< file_name >),
fssp (< file_name >)) **= = 0**)
{
rcs (< atend_stmt >)
}
else
{
rcs (< notatend_stmt >)
}
si < into_id > ::= nil
et < atend_stmt > ::= < stmtlist >
et < notatend_stmt > ::= < stmtlist >

| **if (fsread(** pfname(< stmt>**), ptralloc** + ab(<file_name>),
lb (< file_name > **), ptralloc** + afs (< file_name >),
fssp (< file_name >)) **= = 0**)
{
rcs (< atend_stmt >)
}
else
{
move(ptralloc + ab (< file_name >), lb (< file_name >),
adresse (< into_id >), np (< into_id >));
}
si < into_id > ::= < identifier >
et < atend_stmt > ::= < stmtlist >
et < notatend_stmt > ::= nil

| **if (fsread(** pfname (<stmt>)**, ptralloc** + ab (<file_name>),
lb (< file_name >)**, ptralloc** + afs (< file_name >),
fssp (< file_name >)) **> 0**)
{
move(ptralloc + ab (< file_name >), lb (< file_name >),
adresse (< into_id >), np (< into_id >));
rcs (< notatend_stmt >)
}
si < into_id > ::= < identifier >
et < atend_stmt > ::= nil

et < notatend_stmt > ::= < stmtlist >

| **if (fsread(** pfname(<stmt>), **ptralloc** + ab (<file_name>),
lb (< file_name >), **ptralloc** + afs (< file_name >),
fssp (< file_name >)) = = **0**)
{
rcs (< atend_stmt >)
}
else
{
move(**ptralloc** + ab (< file_name >), lb (< file_name >),
adresse (< into_id >), np (< into_id >));
rcs (< notatend_stmt >)
}

si < into_id > ::= < identifier >
et < atend_stmt > ::= < stmtlist >
et < notatend_stmt > ::= < stmtlist >

bufname (< file_name >) = ENVFILES2 (fname = < file_name >, bufname)
lb (< file_name >) = ENVDATA (name = bufname (< file_name >), longueur)
ab (< file_name >) = ENVDATA (name = bufname (< file_name >), longueur)

adresse (< into_id >) = adresse (< identifier >)
si < into_id > ::= < identifier >

np (< into_id >) = np (< identifier >)
si < into_id > ::= < identifier >

dvs (< stmt >) = dvs (< atend_stmt >)
dvs (< notatend_stmt >)
dvs (< atend_stmt >) =
nil si < atend_stmt > ::= nil
| dvs (< stmtlist >)
si < atend_stmt > ::= < stmtlist >
dvs (< notatend_stmt >) =
nil si < notatend_stmt > ::= nil
| dvs (< stmtlist >)
si < notatend_stmt > ::= < stmtlist >

nvar (< atend_stmt >) = nvar (< stmt >)
nvar (< notatend_stmt >) = rnvar (< atend_stmt >)
nvar (< stmtlist >) =
nvar (< atend_stmt >)
si < atend_stmt > ::= <stmtlist>
| nvar (< notatend_stmt >)
si < notatend_stmt > ::= < stmtlist >

rnvar (< atend_stmt >) =
nvar (< atend_stmt >) si < atend_stmt > ::= nil
| rnvar (< stmtlist >) si < atend_stmt > ::= < stmtlist >
rnvar (< notatend_stmt >) =
nvar (< notatend_stmt >) si < notatend_stmt > ::= nil
| rnvar (< stmtlist >) si < notatend_stmt> ::=<stmtlist>

rnvar (< stmt >) = rnvar (< notatend_stmt >)

rcs (< atend_stmt >) = rcs (< stmtlist >)
rcs (< notatend_stmt >) = rcs (< stmtlist >)

7.7.7.4. Représentation de l'instruction WRITE

L'instruction WRITE est représentée par la fonction **fswrite** (cfr 7.5.4.2.3 et 7.6.4.3).

Afin de mettre dans le buffer de nom <data_name> le contenu de la donnée spécifiée dans la clause **FROM**, nous utilisons la fonction **move** (cfr 7.6.1.5).

rcs (< WRITEStmt >) = rcs (< stmt >)

rcs (< stmt >) =
fswrite(pfname (< data_name >)**, ptralloc** + ab (< data_name >), lb (< data_name >)**, ptralloc** + afs (< data_name >)**,** fssp (< data_name >) **);**
si < from_id > ::= nil
| **move (** adresse (< from_id >), np (< from_id >)**,**
ptralloc + ab (< data_name >), lb (< data_name >) **);**
fswrite(pfname (< data_name >)**, ptralloc** + ab (< data_name >), lb (< dat_name >)**, ptralloc** + afs (< data_name >)**,** fssp (< data_name >) **);**
si < from_id > ::= < identifier >

pfname (< data_name >) =
"**stdout**" si efn (< data_name >) = "**stdout**"
| progr_name (< id_div >)_pfn (< data_name >)

pfn (< data_name >) = ENVFILES2 (bufname = < data_name >, fname)
ab (< data_name >) = ENVDATA (name = < data_name >, adresse)
lb (< data_name >) = ENVDATA (name = < userdef_name >, np)
efn (< data_name >) = ENVFILES1 (fname = pfn (< data_name >), efn)
fsname (< data_div >) = ENVFILES1 (fname = pfn (< data_name >), fsname)
afs (< data_div >) = ENVDATA (name = fsname (< data_name>), adresse)
fssp (< data_name >) = ENVFILES1 (fname = pfn (< data_name >), fssp)

adresse (< from_id >) = adresse (< identifier >)
si < from_id > ::= < identifier >

np (< from_id >) = np (< identifier >)
si < from_id > ::= < identifier >

dvs (< stmt >) = nil
rnvar (< stmt >) = nvar (< stmt >)

7.7.8. Représentation de l'instruction CALL

L'instruction CALL génère l'appel de la fonction C représentant le programme COBOL dont le nom est spécifié dans le littéral alphanumérique et lui passe l'adresse des données spécifiées dans sa clause **USING** si celle-ci est spécifiée.

rcs (< CALLStmt >) = rcs (< stmt >)

rcs (< stmt >) =
< alphanum_literal >();
si < CALLStmt > ::= < alphanum_literal >
| < alphanum_literal >(adr_of_identifiers (< call_list >));
si < CALLStmt > ::= < alphanum_literal >
< call_list >

adr_of_identifiers (< call_list >) = adr_of_identifiers (< id_list >)

adr_of_identifiers (< id_list >$_1$) =
adresse (< identifier >)
si < id_list >$_1$ ::= < identifier >
| adresse (< identifier >) adr_of_identifiers (< id_list >$_2$)
si < id_list >$_1$ ::= < identifier >
< id_list >$_2$

rnvar (< stmt >) = nvar (< stmt >)
dvs (< stmt >) = nil

7.7.9. Représentation des instructions EXIT, EXIT PROGRAM et STOP RUN

Les instructions de terminaison sont représentées par les fonctions C **return(0)** et **exit(0)**. La fonction **return(0)** provoque la fin d'exécution de la fonction C dans laquelle elle se trouve et le retour à la fonction appelante. La fonction **exit(0)** arrête l'exécution du programme C.

Si < stmt > ::= < EXITStmt >,
dvs (< stmt >) = nil
rnvar (< stmt >) = nvar (< stmt >)
rcs (< stmt >) = **return(0);**

Si < stmt > ::= < EXITPGMStmt >,
dvs (< stmt >) = nil
rnvar (< stmt >) = nvar (< stmt >)
rcs (< stmt >) = **return(0);**

Si < stmt > ::= < STOPRUNStmt >,
dvs (< stmt >) = nil
rnvar (< stmt >) = nvar (< stmt >)
rcs (< stmt >) = **exit(0);**

7.7.10. Représentation de l'instruction PERFORM

Rappelons avant d'examiner la manière de représenter l'instruction PERFORM en C, que chaque section et chaque paragraphe devient une fonction C à laquelle on passe la valeur du pointeur **ptralloc**.

Dans le cas où <PERFORMStmt> ::= <procedure_name>, l'instruction PERFORM est représentée par l'appel de la fonction C représentant cette section ou ce paragraphe.

Dans le cas où <PERFORMStmt> ::= <stmtlist>, nous générons les instructions de cette <stmtlist>.

Dans le cas où une condition est spécifiée, nous utilisons la boucle **while** dont la condition d'exécution du corps de la boucle devient la négation de <condition>.

Dans le cas où le symbole <id_or_int> apparaît dans l'instruction PERFORM, nous générons des variables C de type *int* que nous utilisons dans la structure **for** qui nous permet de représenter cette instruction.

rcs (< PERFORMStmt > = rcs (< stmt >)

rcs (< stmt >) =
progr_name (< id_div >)_< procedure_name >**(ptralloc);**
si < PERFORMStmt > ::= < procedure_name >
| rcs (< stmtlist >)
si < PERFORMStmt > ::= < stmtlist >

```
| cnvar2 ( < stmt > ) = ntoi( attr_of_id ( < identifier > ) );
  for ( cnvar ( < stmt > ) = 1; cnvar ( < stmt > ) <= cnvar2 ( < stmt >;
  ++cnvar
  ( < stmt > ) )
  {
  progr_name ( < id_div > )_< procedure_name >( ptralloc );
  }
        si < PERFORMStmt > ::= < procedure_name > < id_or_int >
        avec < id_or_int > ::= < identifier >
| for ( cnvar ( < stmt > ) = 1; cnvar ( < stmt > ) <= < integer >; ++ cnvar
  ( < stmt > ) )
  {
  progr_name ( < id_div > )_< procedure_name >( ptralloc );
  }
        si < PERFORMStmt > ::= < procedure_name > < id_or_int >
        avec < id_or_int > ::= < integer >
| cnvar2 ( < stmt > ) = ntoi( attr_of_id ( < identifier > ) );
  for ( cnvar ( < stmt > ) = 1; cnvar ( < stmt > ) <=
  cnvar2 ( < stmt > ); ++ cnvar ( < stmt > ) )
  {
  rcs ( < stmtlist > )
  }
        si < PERFORMStmt > ::= < stmtlist > < id_or_int >
        avec < id_or_int > ::= < identifier >
| for ( cnvar ( < stmt > ) = 1; cnvar ( < stmt > ) <=
  < integer >; ++ cnvar ( < stmt > ) )
  {
  rcs ( < stmtlist > )
  }
        si < PERFORMStmt > ::= < stmtlist > < id_or_int >
        avec < id_or_int > ::= < integer >
| init_of_variables_for_a_condition ( < condition > )
  while !( repr_of_a_condition ( < condition > ) )
  {
  progr_name ( < id_div > )_< procedure_name >( ptralloc );
  }
        si < PERFORMStmt > ::= < procedure_name > < condition >
| init_of_variables_for_a_condition ( < condition > )
  while !( repr_of_a_condition ( < condition > ) )
  {
  rcs ( < stmtlist > )
  }
        si < PERFORMStmt > ::= < stmtlist > < condition >

nvar2 ( < stmt > ) ::= nvar ( < stmt > ) + 1
```

Si < PERFORMStmt > ::= < procedure_name >,
rnvar (< stmt >) = nvar (< stmt >)
dvs (< stmt >) = nil

Si < PERFORMStmt > ::= < stmtlist >,
nvar (< stmtlist >) = nvar (< stmt >)
rnvar (< stmt >) = rnvar (< stmtlist >)
dvs (< stmt >) = dvs (< stmtlist >)

Si < PERFORMStmt > ::= < procedure_name > < id_or_int >
avec < id_or_int > ::= < identifier >,
rnvar (< stmt >) = nvar (< stmt >) + 2
dvs (< stmt >) = **int** cnvar (< stmt >);
int cnvar2 (< stmt >);

Si < PERFORMStmt > ::= < procedure_name > < id_or_int >
avec < id_or_int > ::= < integer >,
rnvar (< stmt >) = nvar (< stmt >) + 1
dvs (< stmt >) = **int** cnvar (< stmt >);

Si < PERFORMStmt > ::= < stmtlist > < id_or_int >
avec < id_or_int > ::= < identifier >,
nvar (< stmtlist >) = nvar (< stmt >) + 2
rnvar (< stmt >) = rnvar (< stmtlist >)
dvs (< stmt >) = **int** cnvar (< stmt >);
int cnvar2 (< stmt >);
dvs (< stmtlist >)

Si < PERFORMStmt > ::= < stmtlist > < id_or_int >
avec < id_or_int > ::= < integer >,
nvar (< stmtlist >) = nvar (< stmt >) + 1
rnvar (< stmt >) = rnvar (< stmtlist >)
dvs (< stmt >) = **int** cnvar (< stmt >);
dvs (< stmtlist >)

Si < PERFORMStmt > ::= < procedure_name > < condition >,
nvar (< condition >) = nvar (< stmt >)
rnvar (< stmt >) = rnvar (< condition >)
dvs (< stmt >) = dvs (< condition >)

Si < PERFORMStmt > ::= < stmtlist > < condition >,
nvar (< condition >) = nvar (< stmt >)
nvar (< stmtlist >) = rnvar (< condition >)
rnvar (< stmt >) = rnvar (< stmtlist >)

```
dvs ( < stmt > )          = dvs ( < condition > )
                            dvs ( < stmtlist > )
```

7.7.11. Représentation de l'instruction MOVE et de la clause VALUE des données de la working-storage section

Afin de représenter celles-ci, nous utilisons les fonctions d'assignation d'une valeur à une donnée décrites au paragraphe 7.6.1.

7.7.11.1. Représentation de la clause VALUE

7.7.11.1.1. Fonctions de déclaration de variables et de représentation des clauses VALUE

La fonction **decl_of_variables_for_value_clauses** sert à déclarer les variables nécessitées afin de représenter les diverses clauses **VALUE** figurant dans les déclarations de données de la working storage section.

La fonction **repr_c_of_value_clauses** génère les fonctions C représentant ces clauses.

```
decl_of_var_for_value_clauses ( < data_div > ) =
            nil    si < data_div > ::= nil
          | dvvc ( < ws_sec > )
                   si < data_div > ::= < file_sec >
                                       < ws_sec >
                                       < lk_sec >

repr_c_of_value_clauses ( < data_div > ) =
            nil    si < data_div > ::= nil
          | rcvc ( < ws_sec > )
                   si < data_div > ::= < file_sec >
                                       < ws_sec >
                                       < lk_sec >

dvvc ( < ws_sec > ) =
            nil   si < ws_sec > ::= nil
          | dvvc ( < des_list > ) si < ws_sec > ::= < des_list >

rcvc ( < ws_sec > ) =
            nil   si < ws_sec > ::= nil
          | rcvc ( < des_list > ) si < ws_sec > ::= < des_list >

dvvc ( < des_list >1 ) =
            nil si < des_list >1 ::= nil
```

| dvvc (< recdes_entry >)
dvvc (< des_list >$_2$)
si < des_list >$_1$ ::= < recdes_entry >< des_list >$_2$

dvvc (< record_body >$_1$) =
nil si < record_body >$_1$ ::= nil
| dvvc (< recdes_entry >)
dvvc (< record_body >$_2$)
si < record_body >$_1$ ::= < recdes_entry >
< record_body >$_2$

rcvc (< des_list >$_1$) =
nil si < des_list >$_1$ ::= nil
| rcvc (< recdes_entry >)
rcvc (< des_list >$_2$)
si < des_list >$_1$ ::= < recdes_entry >< des_list >$_2$

rcvc (< record_body >$_1$) =
nil si < record_body >$_1$ ::= nil
| rcvc (< recdes_entry >)
rcvc (< record_body >$_2$)
si < record_body >$_1$ ::= < recdes_entry >
< record_body >$_2$

7.7.11.1.2. Principe de numérotation des variables générées

Le nom des variables utilisées dans la représentation des clauses **VALUE** est toujours composé du caractère "**c**" et d'un numéro. Cette numérotation locale à la fonction C représentant un programme COBOL, commence à partir de 0 et se poursuit par incrément de 1 lorsqu'une variable doit être générée. Nous utiliserons également ici la fonction **nvar** et **rnvar** afin d'attribuer un numéro à chacune de ces variables.

Ces fonctions sont définies ici comme suit :

nvar (< des_list >) = 0 si < ws_sec > ::= < des_list >

Si < des_list >$_1$ ::= < recdes_entry > < des_list >$_2$,
nvar (< recdes_entry >) = nvar (< des_list >$_1$)
nvar (< des_list >$_2$) = rnvar (< recdes_entry >)

Si < record_body >$_1$ ::= < recdes_entry > < record_body >$_2$,
nvar (< recdes_entry >) = nvar (< record_body >$_1$)
nvar (< record_body >$_2$) = rnvar (< recdes_entry >)
rnvar (< record_body >$_1$) = rnvar (< record_body >$_2$)

Si < record_body >$_1$ ::= nil,
rnvar (< record_body >$_1$) = nvar (< record_body >$_1$)

7.7.11.1.3. Fonctions de déclaration de variables et de représentation d'une clause value

La fonction **rcvc(<recdes_entry>)** génère les fonctions C permettant de réaliser la clause **VALUE** spécifiée dans la déclaration d'une donnée.

La fonction **dvvc(<recdes_entry>)** sert à déclarer les variables C utilisées pour réaliser cette clause.

Si < recdes_entry > ::= < recdes_rec > et < valueclause > ::= nil,
nvar (< record_body >) = nvar (< recdes_entry >)

rcvc (< recdes_entry >) =
rcvc (< record_body >)
si < recdes_entry > ::= < recdes_rec >
et < valueclause > ::= nil
| **strcpy (** cnvar (< recdes_entry >)**, replacedoublequotes (** "alvalue (< recdes_entry >)"));
move (cnvar (< recdes_entry >), **strlen (** cnvar (< recdes_entry)), **ptralloc +** adr (< recdes_entry >), longueur (< recdes_entry >));
si < valueclause > ::= < alphanum_literal >
| **initzero(ptralloc +** adr (< recdes_entry >), longueur (<recdes_entry>), **0**);
si < recdes_entry > ::= < recdes_rec >
et < valueclause > ::= < fig_constant >
avec < fig_constant > ::= ZER0
| **initzero(ptralloc +** adr (< recdes_entry >), longueur (<recdes_entry>), s(< recdes_entry >));
si < recdes_entry > ::= < recdes_data >
et < valueclause > ::= < fig_constant >
avec < fig_constant > ::= **ZERO**
| **initspace(ptralloc +** adr < recdes_entry >), longueur (<recdes_entry>));
si < valueclause > ::= < fig_constant >
avec < fig_constant > ::= **SPACE**
| **inithighvalue(ptralloc +** adr (< recdes_entry >), longueur (<recdes_entry>));

si < valueclause > ::= < fig_constant >
avec < fig_constant > ::= **HIGH-VALUE**
| **initlowvalue(ptralloc +** adr (< recdes_entry >)**,** longueur
(< recdes_entry >) **);**
si < valueclause > ::= < fig_constant >
avec < fig_constant > ::= **LOW-VALUE**
| **initquote(ptralloc +** adr (< recdes_entry >)**,** longueur
(< recdes_entry >) **);**
si < valueclause > ::= < fig_constant >
avec < fig_constant > ::= **QUOTE**
| **dton(ptralloc +** adr (< recdes_entry >)**,** longueur (< recdes_entry >)**,**
p(< recdes_entry >)**,** v (< recdes_entry >)**,** s (< recdes_entry >)**, 0,**
&size_error, codelitnumtodouble (**"**nlvalue (< numeric_literal >)**"**) **);**
si < valueclause > ::= < numeric_literal >

rnvar (< recdes_entry >) =
nvar (< recdes_entry >) + 1
si < valueclause > ::= < alphanum_literal >
| rnvar (< record_body >)
si < recdes_entry > ::= < recdes_rec >
et < valueclause > ::= nil
| nvar (< recdes_entry >)

dvvc (< recdes_entry >) =
char cnvar (< recdes_entry >)[allongueur (< alphanum_literal >) **];**
si < valueclause > ::= < alphanum_literal >
| dvvc (< record_body >)
si < recdes_entry > ::= < recdes_rec >
et < valueclause > ::= nil
| nil

7.7.11.2. Représentation de l'instruction MOVE

rcs (< MOVEStmt >) = rcs (< stmt >)

rcs (< stmt >) =
dton_functions (< id_list >)
si < id_or_lit > ::= < numeric_literal >
| < identifier >
avec pic (< identifier >) = "**9**"
| **strcpy(** cnvar (< stmt >)**, replacedoublequotes** (
"alvalue (< alphanum_literal)**"**) **);**
move_functions (< id_list >)
si < id_or_lit > ::= < alphanum_literal >
| move_functions (< id_list >)

si < id_or_lit > ::= < identifier >
avec pic (< identifier >) = "**X**"
| init_functions (< id_list >)
si < id_or_lit > ::= < fig_constant >

dton_functions (< id_list >$_1$) =
dton_function (< identifier >)
si < id_list >$_1$ ::= < identifier >
| dton_function (< identifier >)
dton_functions (< id_list >$_2$)
si < id_list >$_1$ ::= < identifier >
< id_list >$_2$

move_functions (< id_list >$_1$) =
move_function (< identifier >)
si < id_list >$_1$ ::= < identifier >
| move_function (< identifier >)
move_functions (< id_list >$_2$)
si < id_list >$_1$ ::= < identifier >
< id_list >$_2$

init_functions (< id_list >$_1$) =
init_function (< identifier >)
si < id_list >$_1$ ::= < identifier >
| init_function (< identifier >)
init_functions (< id_list >$_2$)
si < id_list >$_1$ ::= < identifier >
< id_list >$_2$

dton_function (< identifier >) =
dton(attr_of_id (< identifier >)**, 0, &size_error,**
codelitnumtodouble ("nlvalue (< numeric_literal >)" **));**
si < id_or_lit > ::= < numeric_literal >
| **dton(** attr_of_id (< identifier >)**, 0, &size_error,**
ntod(attr_of_id (< id_or_lit >) **));**
si < id_or_lit > ::= < identifier >
avec pic (< identifier >) = "**9**"

attr_of_id (< id_or_lit >) = attr_of_id (< identifier >)
si < id_or_lit > ::= < identifier >

move_function (< identifier >) =
move(cnvar (< stmt >)**, strlen(** cnvar (< stmt >) **),**
adresse (< identifier >)**,** np (< identifier >) **));**
si < id_or_lit > ::= < alphanum_literal >

| **move** (adresse (< id_or_lit >), np (< id_or_lit >), adresse (< identifier >), np (< identifier >));
si < id_or_lit > ::= < identifier >
avec pic (< identifier >) = "**X**"

adresse (< id_or_lit >) = adresse (< identifier >)
si < id_or_lit > ::= < identifier >

np (< id_or_lit >) = np (< identifier >)
si < id_or_lit > ::= < identifier >

init_function (< identifier >) =
initzero(adresse (< identifier >), np (< identifier >), s (< identifier >));
si < fig_constant > ::= **ZERO**
| **initspace**(adresse (< identifier >), np (< identifier >));
si < fig_constant > ::= **SPACE**
| **inithighvalue**(adresse (< identifier >), np (< identifier >));
si < fig_constant > ::= **HIGH-VALUE**
| **initlowvalue**(adresse (< identifier >), np (< identifier >));
si < fig_constant > ::= **LOW-VALUE**
| **initquote**(adresse (< identifier >), np (< identifier >));
si < fig_constant > ::= **QUOTE**

dvvc (< stmt >) =
char cnvar (< stmt >)[allongueur (< alphanum_literal >)];
si < id_or_lit > ::= < alphanum_literal >
| nil

rnvar (< stmt >) =
nvar (< stmt >) + 1 si < id_or_lit > ::= < alphanum_literal >
| nvar (< stmt >)

7.7.12. Représentation des conditions COBOL

Celles-ci sont représentées par les fonctions C décrites dans les paragraphes 7.6.1 et 7.6.2.

a) La comparaison de données

repr_of_a_condition (< condition >) =
| rc1 (< id_or_lit >$_1$) signe (< condition >) rc2 (<id_or_lit >$_2$)

si < id_or_lit >$_1$ ::= < identifier >$_1$
avec pic (< identifier >$_1$) = "**9**"
et < id_or_lit >$_2$ ::= < identifier >$_2$
avec pic (< identifier >$_2$) = "**9**"
| < id_or_lit >$_1$ ::= < identifier >
avec pic (< identifier >) = "**9**"
et < id_or_lit >$_2$ ::= < numeric_literal >
| < id_or_lit >$_1$ ::= < numeric_literal >
et < id_or_lit >$_2$ ::= < identifier >
avec pic (< identifier >) = "**9**"

| **ntod(** attr_of_id (< id_or_lit >$_1$)) **= = 0**
si < id_or_lit >$_1$ ::= < identifier >
avec pic (< identifier >) = "**9**"
et < id_or_lit >$_2$ ::= < fig_constant >
avec < fig_constant > ::= **ZERO**

| **ntod(** attr_of_id (< id_or_lit >$_2$)) **= = 0**
si < id_or_lit >$_1$ ::= < fig_constant >
avec < fig_constant > ::= **ZERO**
et < id_or_lit >$_2$ ::= < identifier >
avec pic (< identifier >) = "**9**"

| **zerocomp(** adresse (< id_or_lit >$_1$), np (< id_or_lit >$_1$))
si < id_or_lit >$_1$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_2$ ::= < fig_constant >
avec < fig_constant > ::= **ZERO**

| **zerocomp(** adresse (< id_or_lit >$_2$), np (< id_or_lit >$_2$))
si < id_or_lit >$_2$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_1$ ::= < fig_constant >
avec < fig_constant > ::= **ZERO**

| **spacecomp(** adresse (< id_or_lit >$_1$), np (< id_or_lit >$_1$))
si < id_or_lit >$_1$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_2$ ::= < fig_constant >
avec < fig_constant > ::= **SPACE**

| **spacecomp(** adresse (< id_or_lit >$_2$), np (< id_or_lit >$_2$))
si < id_or_lit >$_2$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_1$ ::= < fig_constant >
avec < fig_constant > ::= **SPACE**

| **highcomp(** adresse (< id_or_lit >$_1$), np (< id_or_lit >$_1$))
si < id_or_lit >$_1$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_2$ ::= < fig_constant >
avec < fig_constant > ::= **HIGH-VALUE**

| **highcomp(** adresse (< id_or_lit >$_2$), np (< id_or_lit >$_2$))

si < id_or_lit >$_2$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_1$ ::= < fig_constant >
avec < fig_constant > ::= **HIGH-VALUE**

| **lowcomp**(adresse (< id_or_lit >$_1$), np (< id_or_lit >$_1$))
si < id_or_lit >$_1$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_2$ ::= < fig_constant >
avec < fig_constant > ::= **LOW-VALUE**

| **lowcomp**(adresse (< id_or_Lit >$_2$), np (< id_or_lit >$_2$))
si < id_or_lit >$_2$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_1$ ::= < fig_constant >
avec < fig_constant > ::= **LOW-VALUE**

| **quotecomp**(adresse (< id_or_lit >$_1$), np (< id_or_lit >$_1$))
si < id_or_lit >$_1$ ::= < identifier >
avec pic (< identifier >) = "**X**"
et < id_or_lit >$_2$ ::= < fig_constant >
avec < fig_constant > ::= **QUOTE**

| **quotecomp**(adresse (< id_or_lit >$_2$), np (< id_or_lit >$_2$))
si < id_or_lit >$_2$ ::= < identifier >
avec pic (< identifier >) = "**9**"
et < id_or_lit >$_1$ ::= < fig_constant >
et < fig_constant > ::= **QUOTE**

| **compareX9**(adr2 (< condition >), l2 (< condition >),
attr_of_id (< id_or_lit >$_1$), "signe (< condition >)")
si < id_or_lit >$_1$ ::= < identifier >$_1$
avec pic (< identifier >$_1$) = "**9**"
et < id_or_lit >$_2$ ::= < identifier >$_2$
avec pic (< identifier >$_2$) = "**X**"

| **compareX9**(adr1 (< condition >), l1 (< condition >),
attr_of_id (< id_or_lit >$_2$), "signe (< condition >)")
si < id_or_lit >$_1$ ::= < identifier >$_1$
avec pic (< identifier >$_1$) = "**X**"
et < id_or_lit >$_2$ ::= < identifier >$_2$
avec pic (< identifier >$_2$) = "**9**"

| **compare**(adr1 (< condition >), l1 (< condition >),
adr2 (< condition >), l2 (< condition >), "signe (<
condition >)")

attr_of_id (< id_or_lit >$_1$) = attr_of_id (< identifier >)
si < id_or_lit >$_1$ ::= < identifier >

attr_of_id (< id_or_lit >$_2$) = attr_of_id (< identifier >)
si < id_or_lit >$_2$ ::= < identifier >

```
adr1 ( < condition > ) = adresse ( < id_or_lit >1 )
                              si  < id_or_lit >1 ::= < identifier >
                      | cnvar ( < condition > )

adr2 ( < condition > ) = adresse ( < id_or_lit >2 )
                              si < id_or_lit >2 ::= < identifier >
                      | cnvar ( < condition > )

l1 ( < condition > ) =  np ( < id_or_lit >1 )
                              si < id_or_lit >1 ::= < identifier >
                      | cnvar ( < condition > )

l2 ( < condition > ) = np ( < id_or_lit >2 )
                              si < id_or_lit >2 ::= < identifier >
                      | cnvar ( < condition > )

rc1 ( < condition > ) =
        ntod( attr_of_id ( < id_or_lit >1 ) )
              si < id_or_lit >1 ::= < identifier >
      | codelitnumtodouble( "nlvalue ( < numeric_literal > )" )
              si < id_or_lit >1 ::= < numeric_literal >

rc2 ( < condition > ) =
        ntod( attr_of_id ( < id_or_lit >2 ) )
              si < id_or_lit >2 ::= < identifier >
      | codelitnumtodouble( "nlvalue ( < numeric_literal > )" )
              si < id_or_lit >2 ::= < numeric_literal >

signe ( < condition > ) =
        "> " si < condition >  ::= < id_or_lit >1 > < id_or_lit >2
      | "<=" si < condition > ::= < id_or_lit >1 <= < id_or_lit >2
                              |    < id_or_lit >1 NOT > < id_or_lit >2
      | "< " si < condition >  ::= < id_or_lit >1 < < id_or_lit >2
      | ">=" si < condition > ::= < id_or_lit >1 NOT < < id_or_lit >2
                              |    < id_or_lit >1 >= < id_or_lit >2
      | "==" si < condition >  ::= < id_or_lit >1 = < id_or_lit >2
      | "! =" si < condition >  ::= < id_or_lit >1 NOT = < id_or_lit >2

rnvar ( < condition  > ) =
        nvar ( < condition > ) + 1
              si < id_or_lit >1 ::= < alphanum_literal >
               | < id_or_lit >2 ::= < alphanum_literal >
      | nvar ( < condition > )

dvs ( < condition > ) =
        char cnvar ( < condition > )[ allongueur ( < alphanum_literal > ) ];
```

si < id_or_lit >$_1$ ::= < alphanum_literal >
| < id_or_lit >$_2$ ::= < alphanum_literal >

| nil

init_of_variables_for_a_condition (< condition >) =
strcpy(cnvar (< condition >), **replacedoublequotes** (
"alvalue (< alphanum_literal >)") **);**
si < id_or_lit >$_1$ ::= < alphanum_literal >
| < id_or_lit >$_2$ ::= < alphanum_literal >

b) Cas où <condition>$_1$ ::= <condition>$_2$ AND <condition>$_3$
et <condition>$_1$ ::= <condition>$_2$ OR <condition>$_3$

signe (< condition >$_1$) =
"**&&**" si < condition >$_1$ ::= < condition >$_2$ **AND** < condition >$_3$
| "| |" si < condition >$_1$ ::= < condition >$_2$ **OR** < condition >$_3$

dvs (< condition >$_1$) = dvs (< condition >$_2$)
dvs (< condition >$_3$)

nvar (< condition >$_2$) = nvar (< condition >$_1$)
nvar (< condition >3) = rnvar (< condition >2)
rnvar (< condition >$_1$) = rnvar (< condition >$_3$)

init_of_variables_for_a_condition (< condition >$_1$) =
init_of_variables_for_a_condition (< condition >$_2$)
init_of_variables_for_a_condition (< condition >$_3$)

repr_of_a_condition (< condition >$_1$) =
(repr_of_a_condition (< condition >$_2$) signe (< condition >$_1$)
repr_of_a_condition (< condition >$_3$))

c) Cas où <condition>$_1$ ::= NOT <condition>$_2$

dvs (< condition >$_1$) = dvs (< condition >$_2$)

nvar (< condition >$_2$) = nvar (< condition >$_1$)
rnvar (< condition >$_1$) = rnvar (< condition >$_2$)

init_of_variables_for_a_condition (< condition >$_1$) =
init_of_variables_for_a_condition (< condition >$_2$)

repr_of_a_condition (< condition >$_1$) =
!(repr_of_a_condition (< condition >$_2$))

d) Cas où <condition> ::= <identifier> IS NUMERIC et <condition> ::= <identifier> IS NOT NUMERIC

dvs (< condition >) = nil

rnvar (< condition >) = nvar (< condition >)

init_of_variables_for_a_condition (< condition >) = nil

repr_of_a_condition (< condition >) =
numericid(adresse (< identifier >), np (< identifier >))
si < condition > ::= < identifier > **IS NUMERIC**
| **!numericid**(adresse(< identifier >), np (< identifier >))
si < condition > ::= < identifier > **IS NOT NUMERIC**

e) Cas où <condition> ::= <identifier> IS ALPHABETIC et <condition> ::= <identifier> IS NOT ALPHABETIC

dvs (< condition >) = nil

rnvar (< condition >) = nvar (< condition >)

init_of_variables_for_a_condition (< condition >) = nil

repr_of_a_condition (< condition >) =
alphabeticid(adresse (< identifier >), np (< identifier >))
si < condition > ::= < identifier > **IS ALPHABETIC**
| **!alphabeticid**(adresse (< identifier >), np (< identifier >))
si < condition > ::= < identifier > **IS NOT ALPHABETIC**

7.8. Conclusion

Afin de rendre nos règles de traduction agréables à lire, nous avons parlé dans ce chapitre de fonctions associées aux symboles de la grammaire COBOL et d'environnements constitués sous forme de tables. Le chapitre suivant montrera comment implémenter le générateur de code que nous avons conçu de cette manière, à l'aide du Cornell Synthesizer Generator et de son langage de spécification. Nous avons donc exposé le "quoi" générer, il reste à montrer le "comment".

8. Implémentation du générateur de code C

8.1. Introduction

Ce chapitre expose la manière d'implémenter le générateur de code C que nous avons conçu selon une certaine notation. Il décrit les étapes à suivre pour passer de cette notation à celle du langage de spécification SSL. Il montrera comment spécifier attributs et équations d'attributs. Il développera également des techniques de construction d'environnements, ainsi que certaines particularités propres au CSG afin d'améliorer l'efficacité de l'évaluation incrémentale de ces attributs.

8.2. Attributs et équations d'attributs

8.2.1. Formalisme du SSL

Comme nous l'avons déjà dit précédemment, une grammaire attribuée étend une grammaire context-free en attachant des attributs aux symboles de la grammaire. Associé à chaque production de la grammaire se trouve un ensemble d'équations d'attributs; chaque équation définit un attribut par une expression définie en termes d'autres attributs de la production. Ces attributs sont divisés en deux classes disjointes : les attributs synthétisés et les attributs hérités.

Ceux-ci sont attachés à un phylum par une déclaration spécifiant le nom du phylum, le type de chacun de ses attributs (eux-mêmes des phyla) et leur caractère synthétisé ou hérité.

$\mathbf{phylum_0}$ {
    synthesized $\mathbf{phylum_1}$ $\mathbf{attribute_1}$;
    ...
    inherited $\mathbf{phylum_n}$ $\mathbf{attribute_n}$;
}

Dans une telle déclaration, l'ordre des attributs d'un phylum est quelconque; les mots-clé **synthesized** et **inherited** peuvent être abrégés par **syn** et **inh** respectivement. Le type d'un attribut peut être soit un phylum prédéfini du langage SSL, soit un phylum défini par l'utilisateur; un attribut peut donc être un arbre ou un sous-arbre de la syntaxe abstraite représentant un objet éditable.

Exemple : La déclaration suivante associe au phylum **exp** un attribut synthétisé de nom **code** et de type STR (chaîne de caractères).

exp { syn STR code; }

Des attributs peuvent également être attachés à des productions (plutôt qu'à des phyla) par des déclarations d'attributs locaux.

```
local phylum attribute;
```

Pour chaque production, le concepteur d'un éditeur spécifie les équations des attributs des phyla d'une production. Les attributs synthétisés du phylum apparaissant du côté gauche d'une production, les attributs hérités des phyla de son côté droit et les attributs locaux à une production sont appelés attributs d'*output* .

```
phylum : operator
        {
        local phylum attribute;
            ...
        local phylum attribute;
        output-attribute = expression;
            ...
        output-attribute = expression;
        };
```

Un attribut b d'un phylum X dans une production est dénoté par "X.b". Lorsque la production contient plusieurs occurrences d'un phylum X, les différentes occurrences de b sont dénotées de gauche à droite par "X$1.b", "X$2.b", etc. Si X est le phylum du côté gauche d'une production, l'attribut b peut être dénoté par "$$.b". Un attribut local ne peut être dénoté que par son nom "b".

Comme à une production on peut associer différents opérateurs, la syntaxe des équations d'attributs est la suivante :

```
phylum : operator, ... , operator { equations }
        | operator, ... , operator { equations }
            ...
        | operator, ... , operator { equations }
        ;
```

Les équations d'attributs d'une production suivent donc la déclaration de la production à laquelle elles sont associées.

8.2.2. Comment passer de nos règles de traduction au formalisme SSL

Après avoir spécifié les déclarations de phyla de la syntaxe abstraite, le passage de la notation utilisée dans le chapitre précédent à la spécification SSL des attributs et équations d'attributs s'effectue de la manière suivante :

- Les fonctions attachées au membre droit d'une production de la grammaire COBOL deviennent des attributs synthétisés soit locaux.

- Les fonctions attachées aux symboles non-terminaux du membre gauche d'une production deviennent des attributs hérités.

- Leurs équations doivent être définies pour le phylum du membre gauche de cette production.

- Le symbole de concaténation de chaînes de caractères est "#". Une chaîne de caractères gras doit figurer entre guillemets.

- La fonction SSL de conversion d'une valeur entière en une chaîne de caractères s'appelle "INTtoSTR".

- Le symbole "\n" est utilisé pour la présentation à l'écran de la valeur d'un attribut. Il permet de réaliser un saut de ligne.

- La notation indicée utilisée s'applique de la même manière.

- Le type des fonctions dans lesquelles apparaissent des caractères gras est STR, celui des fonctions numériques est INT.

- Les attributs non locaux à une production sont hérités et/ou synthétisés le long de l'arbre de dérivation et deviennent des attributs de celle-ci.

- De plus, afin de vérifier les conditions apparaissant dans la définition des fonctions :

1° Nous définissons des attributs (de type INT) attachés aux productions lorsque celles-ci apparaissent dans la partie "si ... " et leur attribuons une valeur 1 ou 0 testée dans la condition.

2° Les conditions définies en termes de valeurs d'autres fonctions sont rédigées en SSL en utilisant la structure if-then-else propre à celui-ci. Elle se présente de la manière suivante :

(condition) ? expression$_1$: expression$_2$.

Le symbole "?" représente le "then", ":" le "else".

3° Les symboles apparaissant dans les conditions sont ceux du langage C.

8.2.3. Elements méthodologiques

La démarche que nous utilisons afin de passer à la spécification SSL est la suivante :

1° Nous déclarons les attributs en les attachant aux phyla correspondant aux productions de la grammaire en caractérisant leur caractère synthétisé ou hérité.

2° Si une fonction n'est associée à aucun phylum d'une production dont cette fonction définit la valeur d'un attribut, nous utilisons des attributs le long de l'arbre de dérivation, c'est-à-dire à chaque noeud de cet arbre, permettant de passer la valeur de celle-ci de manière à la rendre disponible au niveau de cette production.

3° Nous déclarons ensuite des attributs afin de représenter les conditions exprimées dans la définition d'une fonction.

4° Nous définissons finalement les équations de chacun des attributs déclarés.

Exemple :

La valeur de la fonction repr_of_size_error_stmts (< stmt >) dans le cas où < size_errorstmt> ::= < stmtlist > et < notsize_errorstmt > ::= nil est :

```
if ( size_error = 1 )
{
rcs ( < stmtlist > )
}
```

Pour cette fonction, la démarche que nous suivons est :

1° Nous déclarons d'abord l'attribut synthétisé **rcses** (attaché au phylum stmt) de type STR permettant de représenter de manière abrégée la fonction repr_of_size_error_stmts, un attribut de type STR appelé **rcs** pour la fonction rcs attaché aux phyla size_errorstmt et notsize_errorstmt.

2° Nous déclarons un attribut synthétisé **sp** attaché aux phyla size_errorstmt et notsize_errorstmt dont la valeur est 1 quand la partie droite de la production correspondante est < stmtlist >, 0 quand elle est nil. Cette valeur *nil* est représentée par un opérateur sans arguments (completing term) auquel nous attachons cet attribut et lui donnons la valeur 0.

3° Son équation devient ensuite :

```
$$.rcses = ( ( size_errorstmt.sp == 1 ) && ( notsize_errorstmt.sp == 0 ) ) ?
        "if ( size_error == 1 )\n{\n"#size_errorstmt.rcs#"\n}\n" :  ... ;
```

8.3. Implémentation des environnements de compilation

8.3.1. Représentation d'un environnement

Chaque environnement que nous avons considéré comme une table dans le chapitre précédent est implémenté en SSL sous la forme d'un phylum liste.

Exemple :

Supposons que nous ayons spécifié un environnement de nom ENV composé de lignes contenant le nom et le type d'une donnée. Si le nom est un phylum de type ID et le type d'une donnée un phylum STR, sa représentation devient en SSL:

```
list ENV;
ENV : NullEnv ()
        | EnvConcat ( BINDING ENV )
        ;
BINDING : Binding ( ID STR );
```

Chaque ligne devient un phylum élément de la liste ENV contenant les informations de ses colonnes. Dans l'exemple, BINDING représente un phylum élément de liste et les arguments (eux-mêmes des phyla) de son opérateur Binding sont respectivement ID et STR.

8.3.2. Construction d'un environnement

Un environnement peut être construit de deux manières. Examinons cela au travers d'un exemple.

Exemple :

Soit une liste de déclarations de données dont la spécification SSL est la suivante :

```
list DECLLISTE;
DECLLISTE : NullListe ()
          | ListeConcat ( ELEMENT DECLLISTE );

ELEMENT :   ElementNil ()
          | Element ( name type )
          ;
```

La construction de l'environnement ENV défini plus haut s'effectue de la manière suivante :

1° Nous associons un attribut hérité **inhenv** et synthétisé **synenv** de type ENV au phylum DECLLISTE, ainsi qu'un attribut synthétisé **bind** de type BINDING au phylum ELEMENT.

2° Sachant que "::" représente en SSL le symbole de concaténation d'un élément de liste au sommet de cette liste, que "@" est le symbole de concaténation de deux listes et que le

premier noeud correspondant au phylum DECLLISTE dans la représentation arborescente de celui-ci reçoit comme attribut inhenv la liste vide NullEnv,

a) Si **value** est un attribut synthétisé associé aux phyla name et type représentant la valeur à mettre dans cet environnement, les équations suivantes construisent l'environnement ENV en synthétisant celui-ci à partir du bas de l'arbre de dérivation correspondant à DECLLISTE.

```
DECLLISTE : NullListe
        {
        $$.synenv = NullEnv();
        }
                | ListeConcat
        {
        DECLLISTE$1.synenv = ELEMENT.bind ::
                                        DECLLISTE$2.synenv;
        };

ELEMENT : Element
        {
        $$.bind = Binding( name.value, type.value );
        };
```

b) les équations suivantes construisent l'environnement ENV en le faisant hériter à chacun des éléments de la liste DECLLISTE et en le synthétisant ensuite quand le completing term de cette liste est rencontré :

```
DECLLISTE : NullListe
{
$$.synenv = $$.inhenv;
}
| ListeConcat
{
local ENV env1;
env1 = ELEMENT.bind :: NullEnv;
DECLLISTE$1.synenv = DECLLISTE$2.synenv;
DECLLISTE$2.inhenv = DECLLISTE$1.inhenv @ env1;
};
```

L'utilisation d'un attribut local **env1** dans ce cas est due à la contrainte selon laquelle une liste doit toujours être terminée par son completing term (ici NullEnv).

L'avantage de cette seconde méthode de construction est que l'environnement constitué par les éléments précédant un élément de cette liste peut être hérité de telle manière que l'on peut vérifier à partir de celui-ci si une donnée par exemple a déjà été déclarée ou trouver dans notre cas l'adresse d'une donnée redéfinie par une autre (cfr chapitre 7, 5.2.4), ainsi que toute autre caractristique des éléments de cet environnement.

Son principe de construction se résume donc à ceci :

1° Tout élément de la liste des déclarations de données hérite d'un environnement.
2° Les informations relatives à cet élément sont ajoutées à celui-ci.
3° L'environnement ainsi augmenté est hérité par le reste de la liste de déclarations et le processus de construction continue de la même manière jusqu'à la fin de cette liste.

4° L'environnement ainsi composé est ensuite synthétisé jusqu'au sommet de l'arbre représentant ces déclarations.

Un environnement construit selon ces deux méthodes peut être ensuite hérité vers tout noeud de l'arbre de dérivation d'un programme qui le requiert.

8.3.3. La consultation d'un environnement

L'accès aux éléments d'un environnement s'effectue en utilisant des fonctions rédigées par le concepteur d'un éditeur en utilisant le langage de spécification SSL.

Exemple :

```
BINDING lookup ( ID id, ENV env )
      {
      with ( env ) (
            NullEnv : Binding ( "?", "?" ),
            EnvConcat ( b, e ) :
                  with ( b )
                  ( Binding ( s, *) : (id = = s ? b : lookup ( id, e )) )
                  )
      };
```

Cette fonction de nom **lookup** cherche dans un objet de type ENV le premier Binding(s, *) tel que son argument id est égal à s et le renvoie, soit retourne Binding("?", "?").

L'accès aux arguments de Binding s'effectue en utilisant à nouveau la structure "with".

Exemple : Si inhenv est un attribut de type ENV, l'attribut b d'un phylum X prend la valeur d'un argument du Binding renvoyé par la fonction lookup de la manière suivante :

```
X.b = with ( lookup ( "a", inhenv ) )
      {
      Binding( *, t ) : t,
      Default : ""
      };
```

L'attribut b reçoit ainsi la valeur du type de la donnée de nom "a" figurant dans l'environnement inhenv.

Quand le nombre d'arguments de l'opérateur Binding est élevé, il est préférable afin d'éviter de multiples accès à l'environnement ENV, d'utiliser un attribut de type BINDING local au phylum X, d'y copier le Binding renvoyé par la fonction lookup et d'accéder ensuite aux arguments de cet attribut.

Exemple :

Si la déclaration de BINDING est :

BINDING : Binding (ID STR STR INT);

Si **bind** est un attribut de type BINDING d'un phylum X, il reçoit un élément Binding de la manière suivante :

X.bind = lookup ("a", inhenv);

Soient les attributs a, b, c de X de même type que les arguments de Binding, ils reçoivent la valeur des éléments de l'attribut bind comme suit :

```
X.a  = with ( bind )
        {
        Binding( *, t, *,* ) : t,
        Default : ""
        };

X.b = with ( bind )
        {
        Binding( *, *, s, * ) : s,
        Default : ""
        };

X.c = with ( bind )
        {
        Binding( *, *, *, i ) : i,
        Default : 0
        };
```

8.4. Les attributs à la demande

Les attributs peuvent être spécifiés à la demande. Cela signifie qu'ils ne reçoivent une valeur que lorsqu'une demande est exprimée par l'utilisateur d'un éditeur afin d'en avoir la valeur plutôt que d'être maintenus automatiquement lorsque leurs valeurs ne sont pas nécessaires.

Les attributs à la demande sont spécifiés en préfaçant leur déclaration avec le mot-clé **demand.**

demand synthesized phylum attribute;
demand inherited phylum attribute;
demand local phylum attribute;

Une demande d'attribut est effectuée soit lors d'un besoin de faire apparaître cet attribut à l'écran, soit indirectement d'un besoin d'évaluer un autre attribut dépendant de celui-ci. Le qualificatif "demand" est une directive pour l'éditeur généré qui affecte seulement l'efficacité de l'évaluation incrémentale. La valeur d'un attribut est inaffectée selon qu'il est ou non évalué à la demande.

Cette évaluation à la demande est nécessaire dans la mesure où elle permet de ne pas maintenir l'ensemble des attributs constituant dans notre cas le code généré pour un programme. Cette génération n'a en effet de sens que lorsqu'on dispose d'un programme correct; il est donc inutile pour l'utilisateur de voir apparaître la valeur de ces attributs tant que la rédaction du programme n'est pas entièrement terminée. Cela provoque à la fois une perte de temps (pour leur évaluation) et une grande utilisation de place mémoire pour des attributs dont la valeur n'est pas nécessaire.

8.5. Les attributs "remote"

A côté des attributs d'une production p, il est possible de se référer directement aux attributs non locaux à p et appartenant à une production différente devant apparaître nécessairement au dessus de chaque instance de cette production p. Par "au dessus", nous voulons dire "entre chaque instance de la production p et le symbole **root**".

La valeur de ceux-ci est alors référencée de la manière suivante :

{ identifier.attribute }.

Chaque *identifier* est soit le nom d'un phylum, soit celui d'un opérateur. Si *identifier* est le nom d'un phylum, *attribute* doit être un attribut synthétisé ou hérité de ce phylum; si *identifier* est le nom d'un opérateur, *attribute* est un attribut local d'une production qui a cet opérateur dans le côté droit de sa spécification.

La valeur d'un tel attribut dans une instance particulière de la production p est celle qui apparaît la première en remontant l'arbre de dérivation à partir de cette instance vers le symbole **root** .

L'avantage de tels attributs est qu'il n'est pas nécessaire d'en faire hériter la valeur tout au long de l'arbre qui le sépare de l'instance d'une production qui désire en obtenir la valeur, et de diminuer ainsi le nombre d'attributs par lesquels devrait passer cette valeur. Cela constitue donc un gain de temps à la fois pour le concepteur d'un éditeur et pour l'évaluation de ce type d'attribut.

8.6. Conclusion

La conception et l'implémentation d'un compilateur incrémental ne constituent pas une tâche simple. Le lecteur a pu remarquer que les attributs représentant le code généré doivent être conçus de manière très locale (à chaque symbole non-terminal d'une grammaire) et que l'implémentation quant à elle nécessite l'apprentissage d'un langage, le SSL, auquel il n'est pas évident de s'habituer; il est de plus souvent difficile de s'imaginer le parcours (dans un arbre de dérivation) que doit faire la valeur d'un attribut pour être rendue disponible à un phylum, ainsi que les manipulations de structures arborescentes représentant des environnements. Nous conseillons donc aux personnes désireuses de réaliser un tel ouvrage de procéder avec méthode.

Conclusion

9. Conclusion

Afin de pouvoir porter un regard critique sur un outil de génération automatique tel que le Cornell Synthesizer Generator, nous avons réalisé un Environnement de Programmation Interactif pour COBOL intégrant un éditeur syntaxique et un compilateur incrémental.

Nous nous sommes choisi un sous-ensemble du COBOL avec le désir de nous attacher aux *principes* de la spécification des aspects syntaxiques et sémantiques (statique et dynamique) de ce langage. Nous avons essayé de ne supprimer aucun élément typique de la philosophie du COBOL afin que les principes de conception que nous avons dégagés permettent de spécifier ce langage dans sa totalité.

Ce mémoire présente comment concevoir et réaliser un environnement de programmation avec le CSG. Il essaye de mettre en exergue des éléments sur lesquels la littérature n'insiste pas assez, ou bien qu'elle n'évoque pas du tout.

Nous avons essayé d'apporter une réponse à la question suivante : «Utiliser un générateur automatique comme le CSG ne revient-il pas à vouloir cueillir des fraises des bois avec un bulldozer ?». En d'autres termes : "est-ce vraiment plus simple d'écrire la spécification d'un environnement de programmation pour le CSG, plutôt que de le programmer directement dans un langage tradiditonnel ?".

Nous avons distingué deux aspects pour formuler notre réponse : la conception et la réalisation.

Concevoir un environnement de programmation en termes de grammaires attribuées demande une démarche qui sort des sentiers battus : il faut étudier les différentes relations qui existent entre les composantes d'un langage et les exprimer de façon déclarative, trouver les "bons" attributs, donner une équation sémantique à chacun. Chaque élément d'une production de grammaire doit être considéré comme une boîte noire : une production ne connaît que ses propres éléments et ignore comment ils sont décomposés. Elle a connaissance de son environnement par l'intermédiaire des valeurs d'attribut dont elle hérite ou des valeurs d'attribut synthétisées par ses éléments. Une équation sémantique d'attribut attachée à une production représente la contribution de cette production au contexte général : *ce* qu'elle doit réaliser, connaissant ce qui a déjà été fait. C'est là un mode de spécification déclaratif dont il faut tirer parti.

Le second aspect dont nous voulons parler concerne la réalisation d'un environnement de programmation avec le CSG. Un tel environnement est construit avec la structure abstraite du langage cible pour fondation. Le langage de spécification SSL que met le CSG à notre disposition est très puissant. Ce qui fait sa force nous paraît déroutant au début : tout objet est un arbre attribuable, tout attribut est un objet structuré... on manipule donc un attribut comme on manipule un objet. Une fois passé le stade d'adaptation à ce langage, réaliser des outils qui utilisent une structure abstraite existante est chose relativement aisée

car le SSL dispose de mécanismes très utiles comme le "pattern-matching", et est déclaratif. Cependant le bât blesse lorsqu'il s'agit d'utiliser les aspects du SSL qui sont typiquement liés à l'édition textuelle d'un programme : la construction d'une structure abstraite à partir d'une structure textuelle concrète impose au concepteur de tenir compte des idiosyncrasies de Lex et de YACC, les "générateurs" utilisés pour produire, respectivement, analyseur lexical et analyseur syntaxique. Il faut donc inclure Lex et YACC, aux côtés des grammaires attribuées et des langages fonctionnels, dans le bagage culturel que demande le CSG, pour pouvoir spécifier un environnement de programmation sans trop de mal.

La spécification que l'on donne au CSG est *une* des briques, qui permettent à celui-ci de construire un environnement de programmation : s'y ajoutent un noyau d'édition syntaxique et un évaluateur incrémental d'attributs, propres au CSG.

Après avoir assimilé les notions nécessaires pour savoir utiliser le CSG, nous pouvons dire que celui-ci offre un moyen fort commode et très puissant de produire des environnements de programmation : nous aurions éprouvé beaucoup plus de difficultés à programmer nous-mêmes notre environnement COBOL, qu'à le spécifier en SSL. En effet, le SSL est taillé sur mesure pour décrire de tels environnements alors qu'un langage conventionnel ne l'est pas.

Le CSG est donc utile si l'on désire créer des environnements de programmation pour des langages syntaxiquement riches. Il rend la tâche plus aisée dans des travaux d'une certaine ampleur : le "monstre" CSG n'est intéressant que pour se mesurer à des "monstres" comme COBOL[1].

L'analyse incrémentale disponible avec des éditeurs générés par le CSG rend possible un grand nombre de vérifications syntaxique et de sémantique statique. L'intérêt est de pouvoir se tailler un éditeur syntaxique "sur mesure", plus ou moins sophistiqué. Cependant il faudra trouver un compromis entre une détection d'erreurs pertinente et complète, et la surcharge occasionnée par de tels mécanismes. Maintenir un grand nombre d'attributs se paye en espace mémoire et en temps calcul. Bien que le coût de la mémoire diminue et que la puissance des processeurs augmente, il faut rester les pieds sur terre : une édition syntaxique avec toutes les vérifications possibles et imaginables est-elle vraiment réaliste si elle nécessite un CRAY pour donner un temps de réponse acceptable à l'utilisateur ?

Un éditeur syntaxique évite à l'utilisateur de devoir répéter de multiples fois le cycle de "mise au point syntaxique" d'un programme. Nous pensons donc que le coût d'une édition syntaxique est acceptable s'il ne surpasse pas le coût total des invocations de ce cycle "traditionnel". L'avantage d'un éditeur syntaxique pourrait alors être que la charge demandée au système est répartie plus uniformément dans le temps, tandis qu'un cycle édition - compilation (vérifications sémantico-syntaxique seulement) utilise les ressources de façon beaucoup plus inégale.

[1] L'interprétation de cette phrase est laissée à la discrétion du lecteur et n'engage en rien l'opinion des auteurs.

Nous ne pensons pas que l'édition syntaxique empêche un programmeur de commettre des fautes : un programme doit être bien conçu avant d'être bien écrit; un éditeur syntaxique devrait agir comme "aide à la typographie" d'un programme, permettre d'éviter la recompilation de celui-ci à cause d'une simple faute de frappe, mais ne devrait pas remettre en cause sa conception. Dans un Environnement de Programmation Interactif, un éditeur syntaxique trouve sa place en tant que "constructeur" de la structure de programme commune que les différents outils de développement d'un tel environnement utilisent.

Si vraiment l'utilisation d'éditeurs syntaxiques augmente la productivité des programmeurs, le rapport entre efficacité - coût - et satisfaction de l'utilisateur peut devenir avantageux : dans ce cas il est temps de sortir de l'ère du développement de programmes selon le "paradigme de la boîte à outils" pour entrer dans celle des Environnements de Programmation Interactifs. Ceux-ci ne sont pas légion à l'heure actuelle. Utiliser des "générateurs automatiques" tels que le CSG ouvre la voie à la multiplication du nombre de ces environnements. Par conséquent, de nouveaux champs d'investigation apparaissent : étudier les avantages que les Environnements de Programmation Interactifs offrent par rapport au paradigme de la "boîte à outils", et tirer parti du rôle pédagogique qu'ils pourraient jouer dans l'apprentissage de la programmation.

La puissance d'expression des Grammaires Attribuées et du langage SSL nous a étonnés. Cependant les Grammaires attribuées ne permettent de décrire qu'une sous-classe d'outils de développement de programmes : elles décrivent bien les propriétés statiques d'un langage, mais sont mal adaptées pour décrire des propriétés dynamiques, à cause de la nature *dérivée* des attributs ([KAISER 1986]). Pourquoi ne pas imaginer de nouvelles applications qui utilisent ce formalisme et étendre la puissance d'expression de celui-ci aux caractéristiques dynamiques d'un programme, à son *histoire* ?

En utilisant le CSG nous avons cruellement ressenti l'absence de méthodologies bien établies pour aider à la spécification de grammaires, qu'elles soient abstraites ou concrètes, ainsi que pour guider la conception de grammaires attribuées, en particulier le choix des "bons" attributs et de leur mode de propagation (héritage ou synthèse). Pour profiter au maximum de la puissance d'expression de ce formalisme, il serait intéressant de penser à des méthodes de construction pour celui-ci.

Annexes

A.Détermination du sous-ensemble COBOL.

Nous avons choisi un sous-ensemble du langage COBOL décrit dans la norme ANSI X3.23-1985. Ce sous-ensemble se base sur le "LEVEL 1 minimum COBOL subset" dont nous avons supprimé tous les éléments considérés comme "obsolètes". Ceux-ci ne seront de toute façon pas repris dans la prochaine norme.

Une description exhaustive de tous les éléments que nous avons repris se trouve dans la syntaxe BNF de notre sous-ensemble COBOL qui se trouve au chapitre 7 de ce mémoire. La sémantique de ces éléments est celle qui est décrite dans la norme. La terminologie utilisée pour décrire des concepts COBOL tels que identificateur, procédure, etc ... est propre au langage COBOL. Nous supposons que le lecteur est familiarisé avec le COBOL; celui qui voudrait plus de précision au sujet de la terminologie propre à ce langage est invité à consulter, dans la norme, le glossaire situé à la section III.

Nous avons préféré choisir un "petit" sous-ensemble de COBOL, avec l'intégralité de sa sémantique, plutôt que la totalité de la norme expurgée de ses caractéristiques difficiles. Nous voulons, en considérant l'intégralité de la sémantique de notre sous-ensemble, montrer que nous avons limité notre champ d'action non pas par facilité, mais pour nous attacher mieux aux principes de la compilation incrémentale, pour montrer qu'avec un langage riche mais peu structuré comme COBOL, il est possible de spécifier sa syntaxe, sa sémantique statique et sa sémantique translationnelle dans le formalisme des grammaires attribuées. Nous avons également voulu montrer qu'il est possible de générer pour ce langage, à partir de spécifications, un environnement de programmation interactif.

B. Etat de notre Réalisation

1. Installation du CSG

Nous n'avons pas pu, durant notre stage, utiliser le CSG car nous disposions d'un ordinateur dont les systèmes d'exploitation (UNIX V et SINIX) étaient incompatibles avec UNIX 4.2 Bsd, indispensable pour pouvoir installer le CSG.

Nous avons pu installer le CSG sur ALMA, un VAX 11/750 de l'institut d'informatique, après de nombreuses tribulations :
- nous avons dû modifier TERMCAP, la base de données décrivant les caractéristiques des terminaux et spécifier que, pour un VT100, NEWLINE correspond à CARRIAGE-RETURN suivi de LINEFEED,
- nous avons créé une commande de compilation "sur mesure" : la commande originale mémorise ses fichiers intermédiaires sur un disque rapide dont la capacité est trop faible dans certains cas. Nous nous sommes donc concocté une commande propre qui place ses fichiers temporaires ailleurs, sur un disque de capacité plus grande.

2. L'environnement de programmation COBOL

Notre but fut, au départ, de fournir un Environnement logiciel composé d'un éditeur syntaxique et d'un compilateur incrémental pour le sous-ensemble COBOL que nous nous sommes fixés.

Du fait des difficultés entraînées par l'installation du CSG, et du temps demandé par cette dernière, nous avons limité nos prétentions. Notre éditeur ne reconnaît pas encore les commentaires, certains conflits liés à l'analyse syntaxique sont en cours de résolution. Une édition strictement textuelle n'est pas encore possible. L'édition structurelle fonctionne par contre très bien, quoique nous pourrions ajouter encore quelques commandes supplémentaires afin d'améliorer le confort d'utilisation. Quelques détails dans les schémas de décompilation doivent être revus afin d'améliorer la représentation visuelle d'un programme. Les spécifications SSL pour la compilation incrémentale sont entièrement rédigées, elles doivent encore être mises au point.

En bref, nous avons réussi à spécifier en SSL la grammaire de notre sous-ensemble COBOL, à produire un éditeur syntaxique textuel-structurel expérimental et opérationnel. Par manque de temps, les vérifications de sémantique statique et la compilation incrémentale n'ont pas pu être incorporées, mais c'est en cours de réalisation.

Pour rédiger nos spécifications SSL, nous avons, en partie, utilisé un éditeur syntaxique pour langage SSL fourni comme exemple avec le CSG.

C. Modification du CSG

Le compilateur SSL doit être modifié afin de reconnaître les états finaux pour le lexème WHITESPACE. Dans le répertoire *syn/src/spec_lang*, il faut modifier la fonction *c_tokdef* du fichier scan.c.

La fonction originale est :

```
c_tokdef(s, p)
        STRING s;
        PRODUCTION p;
        {
        register int c;
        register int lexlength = 0;
        char lexbuf[MAX_LEXEME_LENGTH];

        /* skip whitespace */
        c = inchar(infile);
        while( WHITE(c) ) c = inchar(infile);

        /* process the (optional) start conditions */
        if (c == '<') {
           prefix_conds();
           c = inchar(infile);

           /* skip whitespace */
           while( WHITE(c) ) c = inchar(infile);
           }

        /* copy all until whitespace */
        while ( c != ' ' && c != '\t' && c != '\n' && c!= EOF ) {
           if ( lexlength < MAX_LEXEME_LENGTH ) {
                lexbuf[lexlength] = c;
                lexlength ++;
                }
           else yyerror("lexeme regular expression is too long");
           putc( c, lexfile );
           if ( c == '\\' ) {
                c = inchar( infile );
                if ( c != EOF ) yyerror ( "unexpected end of file" );
                     if ( lexlength < MAX_LEXEME_LENGTH ) {
                     lexbuf[lexlength] = c;
                     lexlength ++;
                     }
```

```
                    else yyerror("lexeme regular expression is too long");
            putc( c, lexfile );
            }
      c = inchar( infile );
      }
lexbuf[lexlength] = '\000';
lex_expr_str = stralloc(lexbuf);

/* skip whitespace */
while( WHITE(c) )  c = inchar(infile);

/* put out the action associated with this lexeme */
if (strcmp (sym_id(lhs(p)), "WHITESPACE") == 0) fprintf(lexfile, "\t;\n");
else {
      fputs ("\t{", lexfile);
      if (c == '<') {
            postfix_cond();
            c = inchar(infile);

            /* skip whitespace */
            while( WHITE(c) ) c = inchar(infile);
            }
      fprintf(lexfile,
            "yylval= newlexeme(%d, str_to_str0(yytext) );
                                                return(%s_TOKEN) ;}\n",
            p,
            s
            );
      }

/* parse '>' */
if (c != '>') yyerror ( "syntax error : missing >");
}
```

La fonction modifiée devient :

```
c_tokdef(s, p)
        STRING s;
        PRODUCTION p;
        {
        register int c;
        register int lexlength = 0;
        char lexbuf[MAX_LEXEME_LENGTH];
        int sw_WHITESPACE;
        int sw_postcond;

        /* skip whitespace */
        c = inchar(infile);
        while( WHITE(c) ) c = inchar(infile);

        /* process the (optional) start conditions */
        if (c == '<') {
           prefix_conds();
           c = inchar(infile);

           /* skip whitespace */
           while( WHITE(c) ) c = inchar(infile);
           }

        /* copy all until whitespace */
        while ( c != ' ' && c != '\t' && c != '\n' && c!= EOF ) {
           if ( lexlength < MAX_LEXEME_LENGTH ) {
                   lexbuf[lexlength] = c;
                   lexlength ++;
                   }
           else yyerror("lexeme regular expression is too long");
           putc( c, lexfile );
           if ( c == '\\' ) {
                   c = inchar( infile );
                   if ( c != EOF ) yyerror ( "unexpected end of file" );
                           if ( lexlength < MAX_LEXEME_LENGTH ) {
                           lexbuf[lexlength] = c;
                           lexlength ++;
                           }
                           else yyerror("lexeme regular expression is too long");
                   putc( c, lexfile );
                   }
           c = inchar( infile );
           }
        lexbuf[lexlength] = '\000';
        lex_expr_str = stralloc(lexbuf);
```

```
/* skip whitespace */
while( WHITE(c) )  c = inchar(infile);

/* put out the action associated with this lexeme */
sw_WHITESPACE = (strcmp (sym_id(lhs(p)), "WHITESPACE") == 0);
sw_postcond = (c == '<');
if  (sw_postcond || (! sa_WHITESPACE)) fputs("\t{", lexfile);
if(sw_postcond) {
          postfix_cond();
          c = inchar(infile);

          /* skip whitespace */
          while( WHITE(c) ) c = inchar(infile);
          }
if (sw_WHITESPACE) fprintf(lexfile, "%s", (sw_postcond)?"}\n":"\t;\n");
else {
   fprintf(lexfile,
          "yylval= newlexeme(%d, str_to_str0(yytext) );
                                             return(%s_TOKEN) ;}\n",
          p,
          s
          );
   }

/* parse '>' */
if (c != '>') yyerror ( "syntax error : missing >");
}
```

D. Spécification SSL de notre Environnement de Programmation pour COBOL

Cette spécification se trouve dans un document séparé, sous forme d'un listing.

Bibliographie

Bibliographie

[AHO & ULLMAN 1986]
Aho, A., Ullman, J. Compilers, principles, techniques and tools. Addison Wesley Series in Computer Science, ISBN 0-201-10088-6.

[Ansi 1985]
American National Standard for Information Systems - Programming language-COBOL; ANSI X3.23-1985; ISO 1989-1985. Revision of ANSI X3.23-1974. American National Standards Institute, inc. 1030 Broadway, New York 10078; 10 septembre 1985.

[ARCHER & CONWAY 1981]
Archer, J., Conway, R. COPE : a Cooperative Programming Environment, Cornell University TR 81-459, Juin 1981.

[BAHLKE & SNELTING 1986]
Bahlke, Snelting. The PSG System : From Formal language Definitions to Interactive Programming Environments. ACM Transactions on Programming Languages and Systems, vol. 8, 4, octobre 1986, pp 547-576.

[CLARINVAL 1981]
Clarinval, A. Comprendre, connaître et maîtriser le COBOL. Norme ANSI 1974; Travaux de l'institut d'informatique n°6, Presses Universitaires de Namur, première édition, 1981.

[DELISLE & al. 1984]
Delisle, N., Menicosy, D., Schwartz, M. Viewing a programming Environment as a single tool in proceeding of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments Avril1984.

[DEMERS & al. 1981]
Demers, A., Reps, T., Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In conference Record of the 8th ACM Symposium on Principles of Programming Languages, Williamsburg, Va., jan 26-28, 1987, pp. 105-116.

[DONZEAU-GOUGE & al. 1975]
Donzeau-Gouge, V., Huet, G., Lang, B., et Levy J.J. A structure-oriented program editor. Tech. Rep. INRIA, Rocquencourt, France, avril 1975.

[FISCHER & al. 1984]
Fischer, C., Johnson, G., Mauney, J., Pal, A. et Stock, D. The POE language-based Editor Project, in proceeding of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments Avril 1984.

[GANSNER & al. 1983]
Gansner, E., Horgan, J., More, D., Sarko, P., Swartwout, D., Reppy, J. SYNED : A language-based editor for an Interactive Programming Environment. IEEE trans-on Software Engineering.

[GHEZZI & MANDRIOLI1979]
Ghezzi, Mandrioli. Incremental Parsing. ACM transactions on Programming languages & systems 1 (1), pp. 58-70 (1979).

[HORWITZ & TEITELBAUM 1986]
Horwitz, S., teitelbaum, T. Generating Editiong Environments Based on Relations and Attributes. ACM. Transactions on Programming Languages and Systems, vol. 8., 4, octobre 1986. pp 577_608.

[JOHNSON 1978]
Johnson, S. YACC : Yet Another Compiler Compiler, Bell Laboratories, Murray Hill, N. J. 07974, juillet 1978.

[KAISER 1986]
Kaiser, E. Generation of run-time environments. ACM 0-89791-197-0/86/0600-0051

[KASTENS 1980]
Kastens, U. Ordered Attribute Grammars. Acta Informatica 13, 3 (1980), pp. 229-256.

[KENNEY 1983]
Kenney, G. For Buiding and running programs quickly, combine interpreter and compiler. Electronic Design, 10 novembre 1983.

[KERNIGHAN & RITCHIE 1984]
Kernighan, B., et Ritchie, D. Le langage C; Masson, Paris 1984 (traduction de "The C programming language" paru chez Prentice Hall, inc.).

[KNUTH 1968]
Knuth, D. Semantics of Context-free Languages in Math. Syst. Theory
2,2 (juin 1968) pp. 127-145.

[LESK & SCHMIDT]
Lesk, M., Schmidt, E. LEX - A lexical analyzer generator. Bell Laboratories, Murray Hill, N. J. 07974.

[MEDINA-MORA & FEILER 1981]
Medina-Mora, R. et Feiler, P. An incremental programming environment. IEEE Tans. Softw. Eng. SE-7, 5, Septembre 1981, pp 472-482.

[MORRIS & SCHWARTZ 1981]
Morris, J., Schwartz, M. The design of a Language Directed Editor for Block-structured languages. Proceedings SIGPLAN/SIGOA Conference on Text Manipulation, Portland, OR, pp. 28-33.

[REISS 1984]
Reiss, S. Graphical Program Development with PECAN Program Development Systems, in proceedings in proceeding of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments Avril 1984.

[REPS & al. 1983]
Reps, T., Teitelbaum, T., et Demers, A. Incremental Context dependent Analysis. ACM Transactions on Programming Language and Systems, Vol 5, n°3, Juillet 1983.

[REPS & TEITELBAUM 1984]
Reps, T., et Teitelbaum, T. The Synthesizer Generator (avril 84) ACM, n°4, pp. 42-48.

[REPS & TEITELBAUM 1987]
Reps, T., et Teitelbaum, T. The Synthesizer Generator Reference Manual. Cornell University, Ithaca, New York, 1987.

[REPS 1984]
Reps, T. Generating Language-Based Environments. 1983 ACM Doctoral Dissertation Award. The MIT Press, Mars 1984.

[REPS & MARCEAU 1986]
Reps, T., Marceau, C. Remote attribute updating for langage-based editors. ACM 0-89791-175-X-1/86-0001.

[SEBESTA 1985]
Sebesta, R. Conversational programming Systems. Journal of Pascal, Ada, & Modula-2, Mai-Juin 1985.

[SHAPIRO & al. 1980]
Shapiro, E., Collins, G., Johnson, L., Ruttenberg, J. PASES : a programming environment for PASCAL.

[STALLMAN 1981]
Stallman, R. EMACS : the Extensible Customizable, Self-Documenting Display Editor. Proc. of the ACM SIGPLAN/SIGOA S ymposium on Text Manipulation. Portland, Oregon, 8-10 juin 1981; apparu dans SIGPLAN Notices 16, 6 (juin 1981) pp. 147-156.

[TEITELBAUM, REPS 1981]
Teitelbaum, T., Reps, T. The Cornell Program Synthesizer : A syntax-directed programming environment. Comm. ACM, 24, 9 (septembre 1981) pp. 563-573.

[TEITELBAUM & al. 1981]
T. Teitelbaum & al. The why and wherefore of the Cornell Program synthesizer; Proceedings SIGPLAN/SIGOA conference on Text Manipulation, Portland, OR, pp. 8-16 (1981).