CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Centre agréé de GRENOBLE (CUEFA)

Examen probatoire

en

INFORMATIQUE

par

Bruno VIVIEN

Etude du système Syntax/Fnc–2 pour la génération de compilateurs

Soutenu le 10 juin 1996

JURY	Président :	Véronique DONZEAU-GOUGE
	Membres :	Jacques COURTIN Alain LANDELLE Hubert GARAVEL Christian CARREZ

Remerciements

Je tiens à remercier :

- Madame Véronique Donzeau–Gouge, Professeur au Conservatoire National des Arts et Métiers, pour avoir bien voulu présider le jury de ce probatoire ;
- Monsieur Jacques Courtin, Professeur à l'Université Pierre Mendès–France de Grenoble et responsable du cycle ingénieur CNAM en informatique de Grenoble, pour m'avoir proposé ce sujet qui m'a permis de découvrir les techniques de compilation ;
- Monsieur Alain Landelle, Responsable du Département Formations Supérieures au Centre Universitaire d'Education et de Formation des Adultes de Grenoble, pour avoir accepté de juger ce travail ;
- Monsieur Pierre–Antoine Badoz, Responsable du département Gestion de la Distribution Optique au Centre National d'Etudes des Télécommunications de Grenoble pour ses encouragements ;
- Monsieur Christian Carrez pour sa participation au jury ;
- Monsieur Didier Parigot, Chargé de Recherche à l'INRIA Rocquencourt pour ses conseils lors de l'installation du système SYNTAX/FNC–2 sur une machine du Centre National d'Etudes des Télécommunications de Grenoble ;
- Monsieur Hubert Garavel, Chargé de recherche à l'Institut National de Recherche en Informatique et en Automatique, qui n'a pas compté son temps pour diriger mon travail et qui m'a enseigné les principes de la rédaction scientifique. Je le remercie également pour le choix de ce sujet et pour son accueil sympathique au sein de l'INRIA ;
- Mademoiselle Michaela Sighireanu en thèse à l'INRIA Rhône–Alpes pour l'aide qu'elle m'a apportée tout au long de la rédaction de ce document, notamment dans l'utilisation de LaTeX [1] ;
- Monsieur Alain Cougoulic en thèse au Centre National d'Etudes des Télécommunications de Grenoble pour ses conseils sur les grammaires attribuées ;
- Monsieur Laurent Souchet, du Service National d'Informatique de France Telecom, actuellement en préparation du mémoire d'ingénieur informatique CNAM au Centre National d'Etudes des Télécommunications de Grenoble, pour ses encouragements et son soutien ;
- enfin Claudia mon épouse, ainsi que mes enfants Céline et Vincent, qui ne m'ont pas beaucoup vu ces derniers temps mais qui ont toujours su faire preuve de patience.

1. Ce texte a été composé avec la distribution GUTenberg de LaTeX

Sommaire

Chapitre 1

Introduction

L'écriture de compilateurs et de traducteurs est un besoin essentiel de l'activité informatique. Elle est rendue nécessaire par l'apparition de nouveaux langages et l'amélioration de langages existants. Dans le domaine des télécommunications et du multimédia, on peut citer notamment les langages JAVA, IDL, ainsi que les langages de description de protocoles normalisés par l'ISO et l'ITU–T.

Le développement d'un compilateur est une entreprise longue et délicate, qui nécessite une organisation adaptée. Elle peut être facilitée par l'emploi d'outils de génie logiciel qui permettent de remédier au manque de convivialité qu'offrent les langages comme C ou l'assembleur et à la difficulté à maintenir le produit final.

Dans ce contexte, l'intérêt de la « méta–compilation » est évident. On aimerait pouvoir spécifier un compilateur avec un langage dans lequel on ne dicterait que les règles nécessaires et rien de plus.

Plusieurs systèmes existent pour la génération de compilateurs : on peut mentionner GAG développé par U. Kastens à l'Université de Karlsruhe, LINGUIST développé par R. Farrow d'abord chez INTEL puis ensuite à l'Université de Columbia, ou encore le SYNTHESIZER GENERATOR développé par T. Reps et T. Teitelbaum à l'Université de Cornel...

La liste précédente n'est pas exhaustive mais montre que ce sujet est toujours d'actualité, la visite du «ÑewsGroup comp.compilers» en est la preuve : on y trouve une rubrique « compiler generators and related tools » dans laquelle de nombreux participants font part de leurs travaux sur les générateurs de compilateurs, presque tous basés sur les grammaires attribuées introduites par Knuth [Knu68].

Ce mémoire présente le système SYNTAX/FNC–2. Il s'appuie sur les documents [ASU91, Jou88, JBP90, JPJ+90, JP90, JP91, Jou91, Rou93, JP94]. Sa structure est la suivante :

- la section 2.1 présente quelques définitions relatives aux grammaires attribuées ;
- la section 2.2 traite des principes et de la composition du système SYNTAX/FNC–2 ;
- la section 2.3 décrit la notion de syntaxe abstraite, qui joue un très grand rôle dans le formalisme des grammaires attribuées.

  Elle décrit également l'outil permettant de les analyser et de les représenter sous une forme interne compréhensible par le reste du système ;
- la section 2.4 présente les outils générant des analyseurs lexicaux et syntaxiques ;
- la section 2.5 donne les détails des outils et langages servant à la description et à l'analyse de la sémantique statique ;
- la section 2.6 décrit l'outil qui produit des évaluateurs à partir des grammaires attribuées ;
- la section 2.7 donne un aperçu de l'interface homme–machine du système FNC–2 ;
- dans les annexes, on trouvera un exemple complet de description de compilateur, il figure dans la distribution du système, j'y ai ajouté quelques commentaires ;
- enfin, on trouvera en fin de document la bibliographie sur laquelle je me suis appuyé ainsi qu'un index.

Chapitre 2

Le système Syntax/Fnc–2

SYNTAX/FNC–2 est un système de génération de compilateurs, basé sur les grammaires attribuées, qui a été développé dans le projet CHLOÉ de l'INRIA Rocquencourt[1] depuis 1986. Citons quelques unes de ses caractéristiques :

- SYNTAX permet de générer des analyseurs lexicaux et syntaxiques capables de corriger automatiquement certaines erreurs du texte source ;
- FNC–2 est un outil spécialisé dans le traitement des grammaires attribuées. Il possède une grande puissance d'expression : il accepte une large classe de grammaires attribuées qui sont les grammaires attribuées fortement non circulaires[2]. Les évaluateurs qu'il génère sont aussi performants en temps d'exécution et en occupation mémoire, que les évaluateurs écrits « à la main ». Les opérations sémantiques sont décrites dans un langage de haut niveau baptisé OLGA, qui s'apparente à un langage de type ML restreint au premier ordre.

2.1 Définitions

Grammaires attribuées

Il est désormais admis que l'écriture d'un compilateur doit être « dirigé par la syntaxe ».

Pour traduire une construction d'un langage de programmation, un compilateur peut avoir besoin d'informations comme le type de la construction, ou le point d'entrée du code généré, ou encore le nombre d'instructions produites... On parlera dans ce cas d'*attributs* associés aux constructions ; ces attributs peuvent être un type, une adresse, un nombre...

Les *grammaires attribuées* [ASU91] constituent un formalisme commode pour décrire les traitements effectués par un compilateur.

1. Adresse des auteurs : INRIA, domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay cedex, France. E-mail : {JOURDAN, PARIGOT, JULIÉ, DURIN, LEBELLEC}@minos.inria.fr
2. D'où le nom du système FNC–2

Une grammaire attribuée utilise une grammaire hors–contexte pour spécifier la structure syntaxique du texte d'entrée. A chaque symbole de la grammaire hors–contexte on associe un ensemble d'attributs et à chaque production un ensemble de règles sémantiques. Le rôle des règles sémantiques est de calculer la valeur des attributs pour les symboles de la production (cf. exemple de la figure 2.1).

FIG. 2.1 - *Ce tableau représente une grammaire attribuée pour la traduction des expressions formées de chiffres séparés par des signes + ou −, en une notation postfixée. A chaque non–terminal est associé un attribut a, dont la valeur est une chaîne représentant la notation postfixée de l'expression engendrée par ce non–terminal. E et T sont des non–terminaux, E représente une expression et T un terme.*

PRODUCTIONS	REGLES SEMANTIQUES	COMMENTAIRES
$E \rightarrow E_1 + T$	$E.a := E_1.a \parallel T.a \parallel' +'$	L'attribut $E.a$ est la concaténation des attributs $E_1.a$, $T.a$ et du signe +.
$E \rightarrow E_1 - T$	$E.a := E_1.a \parallel T.a \parallel' -'$	
$E \rightarrow T$	$E.a := T.a$	
$T \rightarrow 0$	$T.a :=' 0'$	La notation postfixée d'un chiffre est ce chiffre lui-même.
$\ddots$	$\ddots$	$\ddots$
$T \rightarrow 9$	$T.a :=' 9'$	La notation postfixée d'un chiffre est ce chiffre lui-même.

Citons deux qualités importantes des grammaires attribuées :

- elles permettent très naturellement, une décomposition structurelle correspondant à la structure du langage ;
- elles sont *déclaratives* dans le sens où l'on ne spécifie que les règles à utiliser pour calculer la valeur des attributs, sans indiquer l'ordre dans lequel elles interviennent [PRDJ].

Arbres abstraits et phases de compilation

La compilation est divisée en deux parties, l'analyse et la synthèse [ASU91]. Pendant la phase d'analyse, les opérations spécifiées par le programme source sont conservées dans une structure hiérarchique appelée *arbre*. Un *arbre abstrait* est un arbre particulier dans lequel chaque nœud représente une opération ; les fils de ce nœud représentent les arguments de cette opération (cf. figure 2.2). La phase de synthèse consiste à construire le programme cible désiré à partir des informations contenues dans les arbres abstraits.

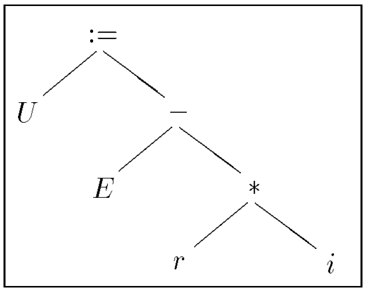

FIG. 2.2 - *Arbre abstrait pour l'expression* $U := E - ri$

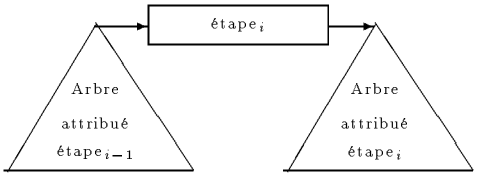

FIG. 2.3 - *Une étape* FNC–*2*

Arbre attribué

Une grammaire attribuée est donc une généralisation des grammaires hors contexte dans laquelle chaque symbole possède un ensemble d'attributs. Si l'on se représente l'arbre syntaxique d'une grammaire attribuée, un nœud étiqueté par un symbole peut être vu comme une structure et les attributs comme des champs de cette structure. Un tel arbre est appelé *arbre attribué*.

Arbre décoré ou arbre annoté

Le processus consistant à donner des valeurs aux attributs par le biais des règles sémantiques associées aux productions s'appelle *décoration* ou *annotation* de l'arbre attribué. Un arbre *décoré* ou *annoté* est un arbre dont les attributs de chacun des nœuds possèdent une valeur.

2.2 Principe et composition du système

Le système SYNTAX/FNC–2 est basé sur le concept de transformation d'un arbre attribué en un autre arbre attribué[3]. Les applications opérant ces transformations sont décrites par des grammaires attribuées. Le système FNC–2 fournit, par l'intermédiaire du langage OLGA décrit dans la section 2.5.2, un mécanisme de construction d'arbres.

Cette notion de transformation est très intéressante pour produire des compilateurs et obtenir des langages intermédiaires qui seront représentés par la production successive d'arbres décorés. Elle permet de découper la génération d'un compilateur en plusieurs étapes[4], chacune prenant en entrée l'arbre attribué fourni par la précédente (cf. figure 2.3). Chaque étape étant spécifiée par une grammaire attribuée ad hoc, la spécification du compilateur à produire est rendue beaucoup plus simple, plus modulaire et plus évolutive.

FNC–2 produit un évaluateur d'attributs opérant sur un arbre décrit par une syntaxe attribuée abstraite. Il n'y a pas d'exigence particulière sur l'utilisation de tel ou tel outil pour produire

3. En fait la transformation peut déboucher sur zéro, un ou plusieurs arbres attribués
4. On parle aussi de phases

cet arbre, FNC–2 peut ainsi fonctionner avec les outils classiques LEX et YACC. Mais dans le cadre de ce mémoire, nous avons préféré étudier le système SYNTAX [5], qui permet de générer des analyseurs lexico–syntaxiques efficaces, à partir de descriptions grammaticales du langage à reconnaître. SYNTAX poursuit les mêmes buts que les outils LEX et YACC mais est beaucoup plus puissant, notamment en ce qui concerne le traitement des erreurs.

La distribution [6] du système SYNTAX/FNC–2 contient :

- ASX : le vérificateur de *syntaxes attribuées abstraites* ;
- SYNTAX : le générateur d'analyseurs lexicaux et syntaxiques qui comprend les outils suivants :
  - BNF : le vérificateur Backus–Naur Form ;
  - LECL : le constructeur lexical ;
  - CSYNT : le constructeur syntaxique ;
  - PRIO : le résolveur d'ambiguïtés ;
  - RECOR : le correcteur d'erreurs ;
  - TABLES_C : le producteur de tables en langage C.
- ATC [7] : le constructeur d'arbres (c'est le lien entre FNC–2 et SYNTAX) ;
- OLGA : le langage utilisé pour décrire les grammaires attribuées ;
- PPAT : le programme d'impression des arbres attribués [8] ;
- FNC–2 : le compilateur utilisé pour traduire le langage OLGA en langage cible.

5. SYNTAX a été conçu et realisé par l'équipe «Langages et Traducteurs» de l'INRIA.

6. On peut la télécharger à l'adresse : `http://www-rocq.inria.fr.charme/FNC-2/`

7. Une implémentation particulière de l'outil ATC a été adaptée aux outils LEX et YACC afin de pouvoir utiliser ces derniers en lieu et place du système SYNTAX.

8. PPAT ne fait qu'imprimer des arbres, nous mentionnons juste son existence, il ne sera pas davantage détaillé dans ce document

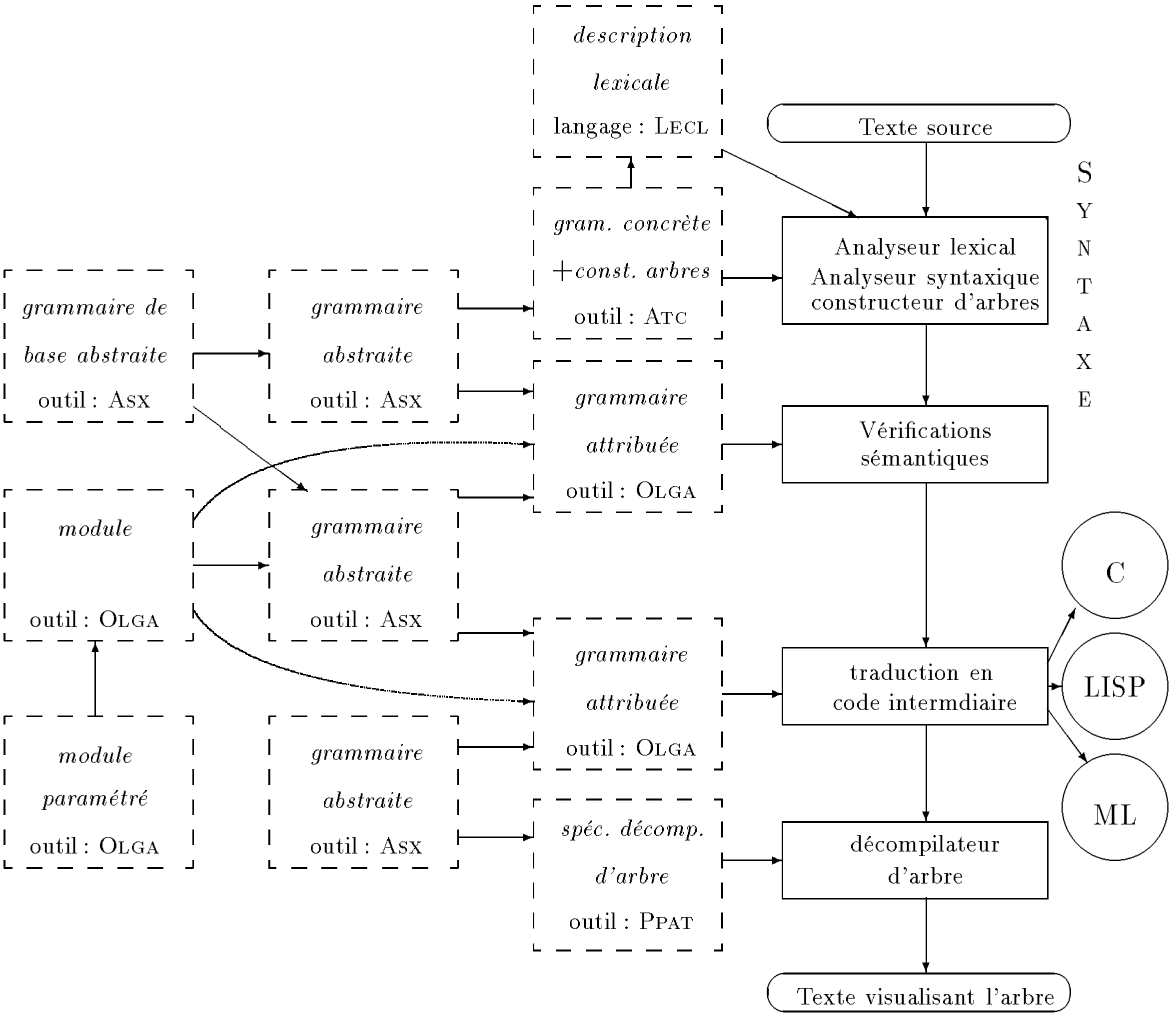

FIG. 2.4 - *Vue d'ensemble du système*

2.3 Description et analyse de la syntaxe abstraite

La description de la syntaxe abstraite constitue la première étape du développement d'un compilateur. Elle permet au concepteur de ne spécifier que les règles nécessaires à la grammaire utilisée, sans avoir à intégrer de contraintes liées à la syntaxe concrète.

2.3.1 Notion de syntaxe attribuée abstraite

Les arbres attribués, qui sont les entrées et sorties des évaluateurs d'attributs, sont décrits par des *syntaxes attribuées abstraites*.

Le formalisme utilisé dans la description des arbres abstraits s'appuie sur les *phyla* (pluriel de *phylum*) et les *opérateurs*, que l'on peut respectivement assimiler aux *non–terminaux* et aux *productions* d'une grammaire hors–contexte. L'énumération suivante décrit ces notions (le lecteur pourra consulter en parallèle, l'exemple de la figure 2.5 page 9) :

- Les opérateurs qui étiquettent les nœuds des arbres abstraits peuvent être assimilés aux productions d'une grammaire hors–contexte. Ils représentent les éléments du langage comme les opérateurs, les expressions... Ils sont divisés en deux classes :
  - *les opérateurs hétérogènes d'arité fixe*, qui peuvent avoir un nombre fixe de phyla comme opérandes (les opérateurs d'arité nulle font partie de cette classe, ce sont les feuilles de l'arbre qui représentent les atomes du langage) ;
  - *les opérateurs homogènes d'arité variable*, qui possèdent un nombre variable d'opérandes appartenant tous au même *phylum*. On les appelle encore *opérateurs de liste* ;
- un phylum est une notion syntaxique du langage que l'on assimile à un ensemble d'opérateurs. Un phyla est un esemble de phylum. A chaque nœud de l'arbre étiqueté par un opérateur, on associe un phylum qui indique quels opérateurs peuvent apparaître à partir de ce nœud. Un même phylum peut être associé à plusieurs descendances de différents opérateurs de l'arbre abstrait. En PASCAL par exemple, le *type* dans une déclaration de type, et le *type* dans une déclaration de variable, peuvent être associés au même phylum `TYPE` car ils correspondent à la même notion et possèdent la même forme syntaxique [JP94] ;

Pour définir une syntaxe abstraite, il faut :

- déterminer les opérateurs ;
- déterminer l'arité des opérateurs ;
- déterminer les phyla associés aux fils des opérateurs ;
- déterminer les phyla en listant les opérateurs qu'ils contiennent ;
- déterminer les attributs en listant les phyla auxquels ils sont attachés ;
- déterminer le type des attributs.

C'est le langage ASX qui permet de décrire de telles syntaxes abstraites. L'exemple qui suit décrit les syntaxes abstraites des arbres en entrée et en sortie du programme PPAT. Pour cela trois modules sont utilisés : le premier décrit ce qui est commun aux arbres d'entrée et de sortie ; le deuxième importe les descriptions du premier et ajoute la déclaration des attributs de l'arbre en entrée ; quant au troisième, il importe les descriptions du deuxième et ajoute la déclaration d'un nouvel attribut sur l'arbre de sortie.

En premier lieu, nous présentons la syntaxe abstraite (pour le moment non encore attribuée) utilisée par PPAT le visualiseur d'arbres attribués (cf. figure 2.5). PPAT utilise la notion de *boîte* telle qu'on la connaît dans le formateur de texte LaTeX. Ppat effectue la mise en page des arbres en s'appuyant lui–même sur un arbre de boîtes. Une boîte est soit terminale (auquel cas elle représente une chaîne de caractères), soit une liste de boîtes, soit une boîte vide. Cette structure est attachée au phylum PPAT.

```
grammar boxes-base is    { Le nom de la grammaire est boxe-base }
    root is PPAT ; { PPAT est le phylum contenant l'opérateur représentant
                     la racine de l'arbre }
    PPAT      =  ppat ; { PPAT est un phylum contenant un unique opérateur }
    ppat      -> BOX ;  { ppat est un opérateur d'arité fixe }
    BOX       =  box terminal empty-box ; { BOX est un phylum contenant
                                           plusieurs opérateurs }
    box       -> BOXES ;
    terminal  -> STRING-BOX ;
    empty-box -> ; { empty-box est un opérateur d'arité nulle }
    BOXES     = boxes;
    boxes     -> BOX * ; { boxes : opérateur homogène d'arité variable }
    STRING-BOX = string-box ;
    string-box -> ;
end grammar ;
```

Fig. 2.5 - *Syntaxe abstraite, tirée de [Jou91]*

Pour que cette syntaxe abstraite soit utilisable, il faut pouvoir décrire les attributs associés aux phyla. Comme on le verra dans la section 2.5.2, c'est à partir du langage Olga que l'on spécifiera les grammaires attribuées chargées de décorer les arbres. Pour notre exemple, les attributs relatifs aux arbres de Ppat sont décrits dans les grammaires des figures 2.6 et 2.7.

```
grammar boxes-in is { Grammaire associée à l'arbre attribué en entrée }
    from ppat-predef import all ;  { inclusion des déclarations de types }
    import grammar boxes-base ;    { importation des phyla et des opérateurs
                                     définis dans la grammaire de base }

    root is PPAT ;
    $separator  (BOX) : sep-info ; { $separator : attribut du phylum BOX }
    $string-box (STRING-BOX) : string ;
end grammar ;
```

Fig. 2.6 - *Syntaxe attribuée abstraite décrivant l'arbre attribué en entrée*

```
grammar boxes-out is { grammaire associée à l'arbre attribué en sortie }
    import grammar boxes-in ; { Importation des phyla et des opérateurs
                                définis dans la grammaire de l'arbre
                                d'entrée }
```

```
    root is PPAT ;
    $coord (PPAT, BOX) : coord-box ; { ajout de l'attribut représentant
                                      les coordonnées des boîtes
                                      $coord est un attribut du phylum PPAT
                                      et du phylum BOX }
end grammar ;
```

FIG. 2.7 - *Syntaxe attribuée abstraite décrivant l'arbre attribué en sortie*

Les syntaxes abstraites de FNC–2 présentent plusieurs avantages :

- elles peuvent figurer dans des unités de compilation séparées, ce qui permet d'augmenter la modularité ;
- les phyla et les attributs peuvent être écrits dans l'ordre qui convient le mieux à la lisibilité ;
- elles offrent une vision plus claire des constructions sémantiques ;
- les arbres abstraits générés sont plus petits que les arbres concrets, car ils ne contiennent aucun mot-clef ni aucune autre information contextuelle [Jou91].

2.3.2 Le compilateur Asx

ASX joue un grand rôle puisqu'il est chargé de lire, vérifier et compiler dans une forme interne, les descriptions des syntaxes abstraites [JP94].

Un exemple d'écriture d'une telle grammaire est donné dans le fichier `simproc-base.asx` figurant en annexe (page 31) (les exemples donnés s'appuient sur le langage SIMPROC). On pourra constater que l'écriture de la grammaire est très proche de l'écriture d'une grammaire hors–contexte.

Le compilateur ASX joue les rôles suivants :

- il effectue une analyse lexicale et syntaxique des directives du fichier « `.asx` » ;
- il vérifie que les phylum et opérateurs utilisés sont définis ;
- il analyse les fichiers importés via la clause **import grammar** et cherche à cette fin les fichiers « `.ast` » correspondants. Si un de ces fichiers est absent, le processus s'arrête sur une erreur fatale ;
- il détecte certaines ambiguïtés (e.g. les doubles déclarations) ;
- il fournit en sortie un ensemble de tables contenues dans un fichier « `.ast` » (cf. figure 2.9 page 11) ;
- il fournit également un fichier de résultat de compilation `simproc.asx.l`.

2.4 Description et analyse de la lexicographie et de la syntaxe

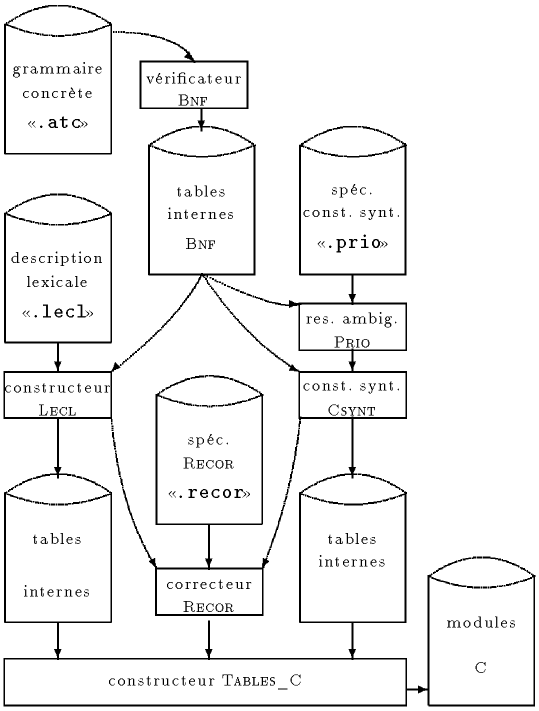

FIG. 2.8 - SYNTAX

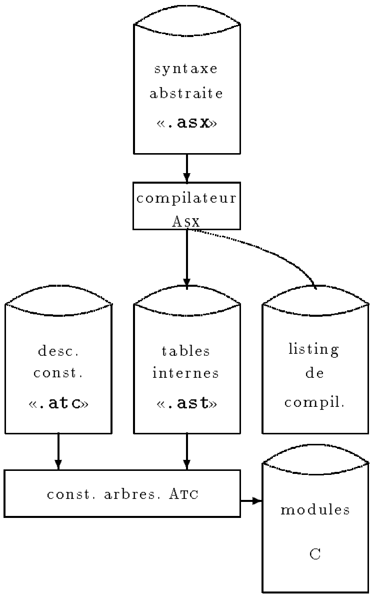

FIG. 2.9 - ATC

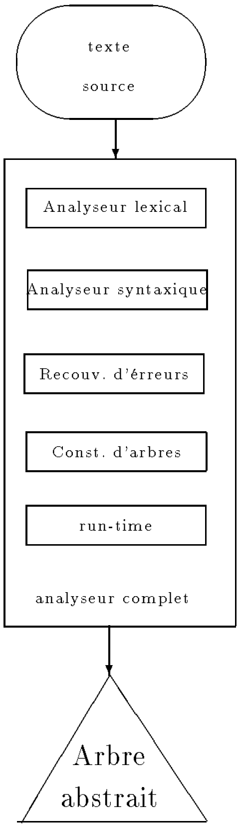

FIG. 2.10 - *Analyseur complet*

2.4.1 Le vérificateur Bnf

BNF est un processeur vérifiant les grammaires hors contexte résidant dans des fichiers de type « `.atc` ». Un fichier « `.atc` » contient la syntaxe concrète du langage à reconnaître[9]. A titre d'exemple, on pourra consulter la syntaxe concrète du langage SIMPROC dans le fichier `simproc.atc` (cf. annexe B.3 page 26). Le programme de l'annexe B.1 page 26 est un exemple de fichier source reconnu par le langage SIMPROC. L'arbre d'analyse syntaxique lui correspondant est donné page 29.

9. Ce fichier contient aussi des directives pour ATC, elles sont ignorées par BNF.

Le processeur BNF effectue les fonctions suivantes :

- il détecte les erreurs syntaxiques dans les directives du fichier « `.atc` » ;
- il lit les grammaires hors–contexte écrites dans une syntaxe proche de la notation «Backus Naur Form » ;
- il vérifie que chaque terminal est accessible à partir de l'axiome ;
- il vérifie que chaque non–terminal est productif, c'est à dire capable de dériver une chaîne de caractères (même vide) ;
- il détecte les ambiguïtés et erreurs diverses notamment les productions identiques, les terminaux qui se dérivent eux-mêmes, l'utilisation de l'axiome en partie droite d'une production etc ;

  BNF accepte les grammaires ambiguës, à condition qu'elles soient résolues par des niveaux de priorité donnés au processeur PRIO présenté plus loin. Cette possibilité intéressante, jointe à la possibilité d'utiliser des prédicats et des actions programmées, permet d'accepter des langages non déterministes ;
- BNF construit des tables de transition donnant la suite des terminaux pouvant apparaître derrière un terminal donné ;
- il fournit en sortie, un ensemble de tables internes utilisées par les autres processeurs, notamment PRIO, CSYNT et LECL ;

2.4.2 Le constructeur lexical Lecl

C'est le constructeur lexical de SYNTAX. Un exemple d'utilisation est donné par le fichier **simproc.lecl** (cf. annexes page 26). Il offre les caractéristiques suivantes :

- il prend ses entrées dans les fichiers « `.lecl` » contenant les spécifications des unités lexicales du langage à analyser. Il lit également les tables internes produites par BNF, notamment celles donnant la liste des terminaux pouvant apparaître derrière un terminal donné ;
- il fournit en sortie des tables décrivant l'analyseur lexical engendré ;
- la description des terminaux de la grammaire est faite par des expressions régulières (comme avec LEX) dont les opérandes sont des classes de caractères ;

  LECL offre la possibilité de nommer des «morceaux » d'expressions régulières et de les réutiliser dans la construction d'autres expressions. Il offre également la possibilité de créer des synonymes pour des mots-clefs, par exemple **proc** est un synonyme de **procedure**.

– Il autorise la définition d'*actions* pour permettre à l'utilisateur d'effectuer au cours de l'analyse, des traitements qui ne peuvent pas se spécifier avec des expressions régulières, par exemple la transformation d'une unité lexicale en majuscules/minuscules. LECL permet aussi de définir des prédicats qui permettent à l'utilisateur de vérifier que certains caractères apparaissent bien dans une colonne, ce qui peut être utile pour des langages tels que FORTRAN ou COBOL.

2.4.3 Le constructeur syntaxique Csynt

Le processeur CSYNT travaille à partir des tables internes BNF. Il construit un analyseur syntaxique ascendant selon la méthode LALR(1). Si la grammaire n'est pas reconnue comme étant de classe LALR(1), plusieurs tentatives de résolution du problème sont essayées. En présence d'un conflit l'utilisateur a quatre possibilités : soit l'utilisateur accepte les règles internes appliquées par CSYNT, soit il modifie la grammaire pour la rendre LALR(1), soit il écrit une spécification indiquant au constructeur les choix possibles face à un tel conflit (cf. la description du processeur PRIO ci–dessous), soit il insère dans la grammaire des prédicats et/ou des actions permettant de résoudre le conflit.

2.4.4 Le résolveur d'ambiguïté Prio

Les spécifications de priorité sont écrites dans les fichiers « `.prio` » en termes de priorités sur les opérateurs ou de priorités sur les règles de grammaire. La syntaxe et la sémantique des spécifications sont proches de celles de YACC.

2.4.5 Le correcteur d'erreurs Recor

Le traitement des erreurs est la partie la plus novatrice de SYNTAX. Elle lui confère un avantage considérable sur le couple d'outils LEX/YACC. Lorsqu'une erreur est détectée, l'analyseur produit virtuellement les parties syntaxiquement correctes et les compare à une liste ordonnée de modèles de correction. Ces modèles sont fournis dans des fichier « `.recor` » écrits par l'auteur de la grammaire. Ils spécifient un nombre quelconque de suppressions, insertions ou remplacements dans le source du texte. De nombreuses options permettent à l'utilisateur de contrôler très finement le mécanisme de correction.

Le lecteur intéressé pourra se reporter à l'annexe B.4 page 28 pour visualiser un exemple commenté de ce qu'il est possible de faire dans ce domaine.

2.4.6 Le constructeur Tables_C

Les tables construites par les processeurs précédents sont vérifiées par le processeur TABLES_C et traduites en un ensemble de structures C, destinées à être compilées et liées avec le noyau de SYNTAX [10], pour former un analyseur complet.

10. (Le run-time de SYNTAX est écrit en C)

2.4.7 Le constructeur d'arbres Atc

ATC est le constructeur d'arbres qui permet de faire la liaison entre SYNTAX et FNC–2. Nous avons vu que le principe du système FNC–2 est basé sur la notion d'étape prenant un arbre abstrait en entrée et fournissant zéro, un ou plusieurs arbres abstraits en sortie : il est donc nécessaire de disposer d'un constructeur qui fournira le premier arbre à FNC–2.

Les caractéristiques du constructeur ATC sont les suivantes :

– il travaille sur une syntaxe concrète, qu'il lit dans le même fichier que le processeur BNF, à savoir un fichier « `.atc` » ;

– les arbres générés sont décrits par des syntaxes attribuées abstraites contenues dans les fichiers « `.asx` » ;

– ATC ne manipule que des attributs de type arbre.

2.5 Description et analyse de la sémantique statique

2.5.1 Les grammaires attribuées

Nous allons présenter ci-dessous les types de grammaires utilisées dans FNC–2, ainsi que les classes et catégories d'attributs.

Pour le système FNC–2, il existe deux types de grammaires attribuées :

– les grammaires attribuées *fonctionnelles*, qui sont des grammaires pouvant avoir plusieurs arbres de sortie et qui possèdent donc des syntaxes abstraites de sortie différentes des syntaxes abstraites d'entrée. Pour ces grammaires, la structure de l'arbre de sortie est différente de celle de l'arbre d'entrée ;

– les grammaires attribuées *procédurales*, qui sont des grammaires fournissant un arbre de sortie ayant la même structure que l'arbre d'entrée, mais décoré différemment.

Il existe trois catégories d'attributs :

– les attributs *importés*, qui décorent l'arbre d'entrée de la grammaire ; on les trouve dans la syntaxe attribuée abstraite d'entrée ; ils ne peuvent en aucun cas être modifiés ;

– les attributs *exportés*, qui décorent le ou les arbres de sortie ; ils sont définis dans la syntaxe abstraite de sortie ;

– les attributs *de travail*, qui jouent le rôle d'attributs locaux servant à effectuer des calculs ; ils sont invisibles de l'utilisateur et ne sont pas accrochés aux arbres de sortie.

Les attributs exportés et les attributs de travail sont classés en quatre sous–catégories :

– les *attributs synthétisés*, qui servent à propager de l'information, des feuilles de l'arbre vers la racine (un attribut synthétisé attaché à un phylum est calculé en fonction des attributs des descendants de ce phylum) ;

– les *attributs hérités*, qui servent à propager de l'information, de la racine de l'arbre vers les feuilles. La valeur d'un attribut hérité associé à un phylum est transmise par le père ou un frère de ce phylum ;

– les *attributs mixtes*, qui représentent une association d'un attribut synthétisé et d'un attribut hérité ;

– les *attributs globaux*, qui sont visibles dans l'ensemble du sous-arbre ayant pour racine l'opérateur appartenant à ce phylum.

2.5.2 Le langage Olga pour le calcul des attributs

OLGA est un langage spécialisé dans le traitement des grammaires attribuées. Il permet la spécification de tout ce qui est nécessaire pour définir une grammaire attribuée [Jou91] et facilite grandement la définition des attributs et règles sémantiques. Les points forts de ce langage sont la *facilité* d'utilisation, la *puissance d'expression*, l'*efficacité*, la *modularité* et la *lisibilité*.

Les programmes OLGA sont écrits dans des fichier « `.olga` » et sont destinés à spécifier la décoration des arbres abstraits, qui sont par ailleurs décrits avec les syntaxes attribuées abstraites précédemment définies.

Les caractéristiques essentielles du langage OLGA sont les suivantes :

– il est *indépendant* des méthodes d'évaluation des attributs ;

– c'est un langage à vocation générale que l'on peut envisager d'utiliser dans un contexte différent de FNC–2 ;

– c'est un langage *applicatif*, c.-à-d. un langage pour lequel il n'y a ni affectation ni effet de bord[11], mais seulement des expressions et des fonctions ;

– il est *fortement typé*, c'est une caractéristique importante du langage. OLGA offre la vérification de type pour chaque expression, ce qui permet de détecter beaucoup d'inconsistances lors de la compilation. Les types prédéfinis sont : **int**, **real**, **bool**, **char**, **string**, **token** (type correspondant aux unités lexicales du langage), **source–index** (type permettant d'implémenter la position d'une unité lexicale du texte source ; la position comprend le nom du fichier, le numéro de ligne et la position dans la ligne) et le type « vide ».

  L'*inférence de type* permet à l'auteur de la grammaire de ne pas systématiquement déclarer le type des attributs : c'est OLGA qui se chargera de le déduire du contexte d'utilisation. OLGA possède plusieurs constructeurs pour les types *énumération*, *ensemble*, *sous-ensemble*, *enregistrement*, *union* et *liste* ;

11. La connexion avec des fonctions externes étant possible, le programmeur pourra contourner ce beau principe...

- il permet la définition de fonctions supportant le *polymorphisme* (les fonctions acceptent des paramètres de différents types) ainsi que la surchage d'opérateurs. Les définitions de fonctions appartiennent à des blocs pouvant être imbriqués. Comme en ADA et C++, les fonctions peuvent être déclarées **inline**. Le lecteur pourra se reporter à la page 23 pour visualiser un exemple simple de définition et d'utilisation de fonction ;

- c'est un langage modulaire : les unités de compilation sont des modules de déclarations et des modules de définitions de fonctions. Il est possible de paramétrer les modules, ce qui autorise la création de modules génériques ;

- il existe des instructions spéciales destinées à la manipulation des listes d'attributs attachés aux phyla. L'opérateur **map list** peut effectuer le calcul d'une valeur sur une liste. Il suffit de lui fournir une valeur initiale, une fonction et une liste. Il visitera chacun des éléments de la liste en leur appliquant la fonction ; si la liste est vide il rend la valeur initiale, sinon il rend le résultat de la fonction. Le parcours peut se faire de gauche à droite ou de droite à gauche ;

- il existe des clauses d'importation de grammaires, ce qui est particulièrement intéressant dans le cas des *grammaires attribuées procédurales* (cf. définition page 14), pour lesquelles les arbres d'entrée et de sortie ont la même structure mais sont décorés différemment. Dans un tel cas, l'utilisateur spécifie une syntaxe abstraite de base sans attribut, puis l'importe dans les syntaxes abstraites d'entrée et de sortie qui ne contiennent que des spécifications d'attributs. Ceci permet une factorisation non négligeable des spécifications (cf. annexes pages 31–38) ;

- OLGA permet de gérer les erreurs d'exécution au moyen d'*exceptions*. Plusieurs exceptions sont prédéfinies dans le langage, mais l'utilisateur peut définir les siennes, ainsi que des fonctions de reprise destinées à « capturer » et éventuellement à corriger les erreurs. Signaler à l'utilisateur la présence d'erreurs dans le programme source est un des rôles les plus importants du compilateur [ASU91]. Le langage OLGA comporte à cet effet toutes les facilités de traitement nécessaires. Il est capable de renseigner l'utilisateur à tout moment sur l'origine et la position d'un symbole. Lorsqu'une erreur est détectée au niveau de l'analyse sémantique, le programmeur peut utiliser l'expression **message** pour afficher le contexte dans lequel elle est apparue, ainsi que le niveau de sévérité ;

- OLGA possède un grand nombre d'opérateurs prédéfinis :
  - les opérateurs classiques (qui ne sont que des appels de fonctions avec des noms spéciaux) tels que : $+, -, \cdots, >=$;
  - les deux opérateurs booléen **and** et **or** qui ont la particularité d'être optimisés, par exemple le **and** est un « et court–circuit » : dans l'expression **if A and B and C ...** l'évaluation s'arrêtera à l'évaluation de **A** si **A** s'avère faux ;
  - l'opérateur **let** permet d'introduire un nouveau nom pour une expression ;
  - l'expression conditionnelle permet de sélectionner une valeur parmi d'autres en fonction d'une condition booléenne : c'est le **if–then–else** avec **elsif** comme en

ADA. Dans une expression conditionnelle, toutes les expressions retournées doivent impérativement avoir le même type ;

- l'expression de sélection **case ... of** permet la sélection d'une valeur parmi d'autres en fonction d'une combinaison de tests qui peuvent être de plusieurs type :
  - *test d'égalité* entre une constante et une valeur scalaire ;
  - *test d'appartenance d'une valeur à un intervalle* ;
  - *test de concordance de motif* (pattern matching) : cette facilité permet de vérifier qu'une valeur correspond à une certaine structure. Supposons que la valeur dont le motif nous intéresse soit du type RT suivant :

```
type RT = record
    F1 : T1 ;
    F2 : T2 ;
      ...
    Fn : Tn
end record ;
```

    alors cette valeur ne pourra correspondre qu'à un motif de la forme $RT(P_1, P_2, \cdots, P_n)$ où chaque P_i est un sous-motif correspondant aux types T_i.

- OLGA permet de définir les règles sémantiques associées aux productions, pour calculer la valeur des attributs. C'est en fait l'objectif principal d'une spécification de grammaire attribuée : chaque production consiste en un bloc dans lequel il est possible de définir des valeurs (*attributs locaux de travail*) , des fonctions... Ces définitions ne sont pas visibles de l'extérieur. Les règles sémantiques consistent en l'affectation d'expressions à une occurence d'attribut.

  L'affectation de valeurs aux attributs peut se faire de deux façons :

  - soit explicitement en écrivant les instructions adéquates dans la grammaire OLGA par exemple :

```
attribute synthesized $correct1 (BLOCK, DECLS, DECL) ;

where program -> BLOCK use
    { $correct reçoit la valeur de $correct1 qui est un
      attribut du phylum BLOCK }
    $correct := $correct1(BLOCK) ;
end where
```

  - soit implicitement par l'utilisation de *règles de copies par défaut.* Il s'agit d'une facilité permettant d'omettre l'écriture de règles sémantiques. Ces règles entrent en action lorsque le compilateur OLGA s'aperçoit, après avoir évalué toutes les règles sémantiques attachées à une production, qu'il reste des attributs de sortie à évaluer. La clarté des spécifications OLGA est due en partie à ces règles. L'expérience prouve en effet qu'une grande partie des règles sémantiques ne sont que des règles de copie du style : `$a1 (sym1) := $a2 (sym2)` où la valeur de l'attribut `$a2` du phylum `sym2` est copié dans l'attribut `$a1` du phylum `sym1`. Les règles de copie par défaut qui sont adoptées par le système FNC–2 furent introduites par Bernard Lorho [Lor77] et leur utilité a été largement démontrée depuis.

Donnons un exemple d'une de ces règles : supposons qu'un attribut synthétisé `$s` d'un père (c'est à dire le symbole de gauche d'une production) ne soit pas explicitement défini, la règle adoptée sera alors de lui donner la valeur du même attribut `$s` de son fils le plus à droite (un fils étant un symbole de partie droite de production cf. figure 2.11).

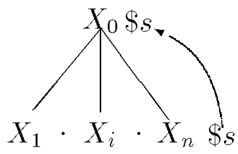

FIG. 2.11 - *Exemple de règle de copie par défaut*

OLGA fournit des attributs *prédéclarés*, notamment **$arity**, qui permet de connaître l'arité d'un opérateur, **$position**, qui donne la position d'un phylum dans une production et **$scx**, qui nous renseigne sur la position d'un phylum dans le texte source.

2.5.3 Les opérations de construction d'arbres

OLGA fournit un mécanisme de construction d'arbres basé sur les descriptions de syntaxes abstraites. Tous les arbres décorés sont décrits par des syntaxes abstraites, une syntaxe abstraite de sortie spécifiera les phyla comme des types d'arbres et les opérateurs comme des fonctions de construction d'arbre.

Fonctions de construction d'arbres

Un opérateur d'arité fixe appartenant au phylum `PH` et déclaré ainsi :

$$\mathtt{op} \rightarrow \mathtt{PH}_1\ \mathtt{PH}_2 \cdots \mathtt{PH}_n$$

correspond à la fonction de construction **function** $\mathtt{op}\ (\mathtt{PH}_1\ ;\ \mathtt{PH}_2\ ;\cdots;\ \mathtt{PH}_n)\ :\ \mathtt{PH}$.

Un opérateur liste appartenant au phylum `PH` et déclaré ainsi :

$$\mathtt{op} \rightarrow \mathtt{PH}_1\texttt{*}\ \mathtt{ou}\ \mathtt{op} \rightarrow \mathtt{PH}_1\texttt{+}$$

correspond aux fonctions de construction : **function** `op () : PH ou`

function $\mathtt{op}\ (\mathtt{PH}_1)\ :\ \mathtt{PH}$

qui sont des fonctions construisant zéro et un descendant respectivement.

Les fonctions suivantes sont prédéfinies dans le langage OLGA .

- **function** `op-post (PH ; PH`$_1$`) : PH` **raise** `TREE-ERROR`
  ajoute un fils à la fin de la liste de nœuds dont `op` est le père.

- **function** `op-pre (PH ; PH`$_1$`) : PH` **raise** `TREE-ERROR`
  ajoute un fils au début de la liste de nœuds dont `op` est le père.

- **function** `op-merge (list of PH`$_1$` ; list of PH1) : PH` **raise** `ARITY-ERROR`
  fusionne deux listes de descendants dont `op` est le père.

- **function** `op-all (list of PH`$_1$`) : PH` **raise** `ARITY-ERROR`
  permet de créer une liste de nœuds dont `op` sera le père.

Décoration d'arbres : le calcul des attributs

Cette partie correspond à la phase de *traduction* d'un programme en un autre. Lorsque la syntaxe abstraite prévoit des attributs pour les phyla, ce qui est la majorité des cas, il est nécessaire de calculer et d'attacher ces attributs aux arbres en même temps qu'on les construit.

La construction d'un nœud de l'arbre de sortie et le calcul des valeurs attachées à ses attributs doivent être effectués dans une seule expression **with...end with** (cf. le contenu du fichier `simproc-terme.olga` en annexe page 46).

2.6 Le compilateur Fnc–2

Le rôle du compilateur FNC–2 est de lire les unités de compilation OLGA pour produire des *évaluateurs d'attributs* efficaces, c'est à dire des programmes dont la fonction est de décorer les arbres en fonction des grammaires spécifiées avec le langage OLGA.

FNC–2 accepte en entrée la spécification d'une grammaire attribuée à partir de laquelle il construit un évaluateur. L'évaluateur est exécutable, il calcule la valeur sémantique spécifiée par la grammaire OLGA.

Les évaluateurs produits par FNC–2 sont des programmes écrits dans l'un des trois langages–cible actuellement supportés par FNC–2 : C, LISP et ML.

Le principe de Ganzinger et Giegerich [GG84] a été retenu par FNC–2 : « Une grammaire attribuée spécifie, un évaluateur d'attributs implémente. ».

Le compilateur FNC–2 est divisé en deux parties distinctes qui sont :

- la partie compilation des modules OLGA;

- la partie création d'évaluateurs exécutables pour la grammaire OLGA. Cette partie s'appelle EVALGEN.

Le choix d'un évaluateur d'attributs est le plus important des choix à faire lorsque l'on construit un système de traitement des grammaires attribuées, car il détermine l'efficacité et la puissance d'expression (les grammaires acceptées).

Pour obtenir un évaluateur efficace, il faut effectuer le maximum de calculs au moment de sa construction afin de minimiser les calculs effectués au moment de l'exécution. De nombreuses recherches ont eu lieu dans ce domaine, elles ont donné naissance à trois familles d'évaluateurs :

- la famille des évaluateurs *dynamiques*, [Har79, Fan72, War76, Lor74, Lor77] : ils déduisent dynamiquement l'ordre d'évaluation des attributs et sont peu efficaces en temps ;

- la famille des évaluateurs *par phases successives* : ils fixent arbitrairement la stratégie de parcours des arbres, sont efficaces en temps, mais limitent fortement la classe des grammaires attribuées acceptées.

- La famille des évaluateurs *statiques* est celle retenue pour implémenter les évaluateurs Fnc–2. Elle réalise un bon compromis entre l'efficacité en temps et la classe des grammaires acceptées. Cette famille est basée sur la déduction d'un ordre statique d'évaluation à partir des graphes de dépendance des attributs. Contrairement aux deux autres familles, c'est la grammaire attribuée qui induit l'ordre et par conséquent le parcours de l'arbre.

Fnc–2 détermine l'ordre d'évaluation des attributs en construisant un graphe des dépendances impliquées par les règles sémantiques. Ceci lui permet de déterminer statiquement si une grammaire est ou n'est pas circulaire, c'est–à–dire s'il existe ou non des dépendances circulaires entre les attributs.

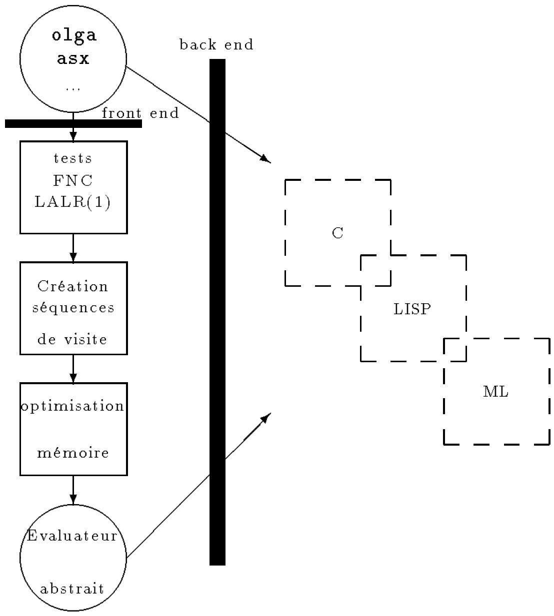

Fig. 2.12 - Fnc–2

Lorsque la grammaire est acceptée par Fnc–2, ce qui veut dire qu'elle est *fortement non circulaire* [JP94], Fnc–2 génère un évaluateur basé sur le paradigme des séquences de visite [JPJ+90] (cf. figure 2.12).

2.7 L'interface X–Windows

Gérer la production d'un compilateur sous Fnc–2 peut s'avérer être une tâche délicate. Elle est facilitée par Xfnc2, un environnement graphique fonctionnant sous X–Windows. Xfnc2 offre les services suivants :

- choix du langage cible C, LISP, FSDL) [JP94] ;
- création de l'ensemble des répertoires de travail indispensables au développement du compilateur ;
- génération automatique d'un squelette de fichier (baptisé `NomCompilateur.MKFNC2`) contenant les directives de production des composants du compilateur. Il s'agit en fait de l'équivalent du `Makefile` cher au monde Unix ;
- création des constructeurs d'arbres ;
- production de l'évaluateur dans le langage cible choisi. L'utilisateur retournera au cycle de construction jusqu'à ce que la compilation du moteur ne comporte plus d'erreur et que le produit final soit conforme à ce qu'il attend.

Chapitre 3

Conclusion

Nous avons étudié le système SYNTAX/FNC–2. Il s'agit d'un système complet qui couvre les différentes phases d'écriture de compilateurs, allant de l'analyse lexico–syntaxique jusqu'à la production et l'optimisation de code objet.

Les premiers développements datent de 1986 mais le système continue à évoluer. Il a fait la preuve de son efficacité lors du développement de plusieurs systèmes et de compilateurs comme PARLOG ou PASCAL ISO. L'application la plus importante réside sans doute dans le développement de FNC–2 par lui–même (notion d'amorçage).

Ce qui fait son intérêt et sa puissance est principalement :

- la production automatique de compilateurs à partir de descriptions de haut niveau du langage source ;
- la puissance du système SYNTAX dans le traitement des erreurs ;
- la présence du langage applicatif OLGA, comparable en puissance à un langage de type ML du premier ordre et dont la spécialité est le traitement des grammaires attribuées ;
- l'efficacité des évaluateurs d'attributs produits par FNC–2 qui peuvent être générés en C, LISP ou ML.

Malgré l'existence d'outils améliorant le confort d'utilisation, la manipulation du système reste complexe :

- il reste encore quelques fichiers C à écrire « à la main », notamment pour l'appel des différents constructeurs ;
- l'utilisation du système nécessite une bonne connaissance des grammaires LALR et du domaine de la compilation.

Je pense que ce système mériterait comme l'indique le rapport d'activité de l'INRIA (cf. [Inr94] pp. 47), une validation dans des applications de qualité industrielle. C'est à mon avis un des outils les plus modernes pour l'écriture de compilateurs. Il serait intéressant de l'utiliser pour réaliser un compilateur dédié à un nouveau langage.

Annexe A

Un exemple de programme Olga

A.1 Fonction polymorphe testant l'appartenance d'un élément à une liste

La fonction suivante est un exemple illustrant les aspects suivants :

- la déclaration et l'utilisation de fonctions ;
- le polymorphisme ;
- la gestion des exceptions : lever et capturer une exception ;
- l'écriture de code **inline** ;
- l'utilisation de l'opérateur **map list** ;
- une expression de test **if–then–else** ;

```
{ La fonction est polymorphe, @T est un type "générique" qui identifie
  n'importe quel type. Cette fonction rendra un booléen indiquant
  l'appartenance ou non de l'élément X à la liste L }

function APPARTENIR-A-LISTE (X : @T ; L : list of @T) : bool is
  declare
      { On lèvera une exception lorsqu'on aura trouvé l'élément }
      exception TROUVE ;

      { La sous-fonction "CHERCHER-X-DANS-LISTE" déclarée dans la
        fonction "APPARTENIR-A-LISTE" est privée.
        Le mot-clef "inline" implique que le code de la fonction sera
        développé en chaque point où elle sera appelée (optimisation du
        temps d'exécution) }
```

```
      inline
          function CHERCHER-X-DANS-LISTE (FACTICE : bool ; Y : @T) :
  bool is
              if Y = X then  { Notons que "=" doit être polymorphe... }
                raise TROUVE { On lève l'exception "trouve" }
              else
                false
              end if
          end function ;

  { point d'entrée de la fonction}
  in
        { Scrutation de la liste L de gauche à droite ; au départ on
          on fixe la valeur calculée à "false" ; par la suite L
          prendra à chaque visite d'un élément de la liste,
          la valeur retournée par la fonction "CHERCHER-X-DANS-LISTE" }

        map  list L left CHERCHER-X-DANS-LISTE value false
        end map

        { pour l'exception "TROUVE", on rend la valeur true }
        catch
          TROUVE : true ;
    end catch
end function ;
```

Annexe B

La découverte de Fnc–2 sur un exemple

B.1 Introduction

Le but de cette annexe est de montrer concrètement comment réaliser un compilateur à l'aide du système SYNTAX/FNC–2. A titre d'exemple, j'ai choisi le langage SIMPROC figurant dans la distribution du système SYNTAX/FNC–2 (cf. manuel technique [JP94]). J'ai complété cet exemple avec les commentaires appropriés.

SIMPROC possède les caractéristiques suivantes :

- C'est un langage structuré basé sur la notion de bloc. L'unité de compilation est un bloc. Chaque bloc est composé d'une liste de déclarations (éventuellement vide) suivie d'une liste d'instructions (jamais vide). La liste de déclarations du bloc principal ne doit pas être vide pour des raisons sémantiques (car les instructions autorisées dans SIMPROC sont telles qu'elles nécessitent au moins une variable). Les règles appliquées sont issues du langage Algol, ce qui signifie qu'un identificateur est visible dans la totalité du bloc où il est déclaré, ainsi que dans les blocs inclus, sauf s'il est redéclaré dans l'un d'entre eux. Un identificateur ne peut être déclaré qu'une seule fois à l'intérieur d'un même bloc.

- SIMPROC possède un unique type, le type *entier* et un seul type structuré, le *tableau unidimensionnel d'entiers*. L'indice inférieur du tableau est « 1 », sa déclaration se fait en indiquant l'indice supérieur (constante entière strictement positive), ce qui revient en fait à déclarer son nombre d'éléments. Une déclaration implique donc soit une variable entière, soit une variable de type tableau d'entiers, soit une procédure.

- Les variables sont soit simples, soit des tableaux de variables indexés par une expression entière, soit des paramètres formels. Chaque utilisation d'identificateur doit être conforme à sa déclaration.

- Les expressions sont formées de variables, de constantes entières, d'opérateurs arithmétiques ($+, -, *, /$) et de parenthèses. Tous les opérateurs ont la même priorité et sont évalués de gauche à droite.

- Les instructions valides sont l'affectation d'un entier, l'appel de procédure et un simple test conditionnel. Ce dernier consiste en un test d'égalité de deux expressions entières qui déclenchera l'exécution d'une des deux listes d'instructions associées (jamais vides).

- SIMPROC possède la notion de *procédure*. Dans une déclaration de procédure on spécifie son nom et les noms des paramètres formels. Les procédures peuvent être récursives. Une procédure ne peut avoir que deux paramètres, un d'entrée et l'autre de sortie. Le premier paramètre formel est passé par valeur et doit être une expression, tandis que le second est passé par référence et doit être une variable. A l'intérieur du corps de la procédure, le premier paramètre prend la forme d'une variable locale initialisée avec la valeur de l'expression tandis que toute référence au second paramètre agit sur une variable externe à la procédure.

L'exemple de programme SIMPROC donné ci–dessous, conduit à l'arbre de syntaxe concrète représenté page 29.

```
integer J ;

proc compute (val I , var K )
begin
    integer L ;
    array T[10] ;

    L := 1 ;
    T[L] := 5
end ;
J := 0
```

B.2 Le contenu du fichier simproc.lecl

```
Classes
  SPACE   = SP + HT + NL + FF ;

Tokens
  Comments = -{ SPACE | "%" "%" {^EOL}* EOL }+ ;
  %ID      = LETTER {["_"] (LETTER | DIGIT)}* ;
  %INT     = {DIGIT}+ ;
```

B.3 Le contenu du fichier simproc.atc

```
import grammar simproc-in ;
```

```
{ program & blocks }
<PROGRAM> = <BLOCK> ;
          program (<BLOCK>)
<BLOCK>   = <DECL *> <STMT +> ;
          block (<DECL *>, <STMT +>)

{ declarations }
<DECL *>  = <DECL *> <DECL> ;
          decls-post (<DECL *>, <DECL>)
<DECL *>  = ;
          decls ()
<DECL>    = integer <id> ";" ;
          int-decl (<id>)
<DECL>    = array <id> "[" <int> "]" ";" ;
          array-decl (<id>, <int>)
<DECL>    = proc <id> ( val <id> , var <id> )  begin <BLOCK> end ";" ;
          proc-decl (<id>.1, <id>.2, <id>.3, <BLOCK>)

{ déclarations }
<STMT +>  = <STMT +> ";" <STMT> ;
          stmt-list-post (<STMT +>, <STMT>)
<STMT +>  = <STMT> ;
          stmt-list (<STMT>)
<STMT>    = <VAR> ":=" <EXPR> ;
          assign (<VAR>, <EXPR>)
<STMT>    = call <id> "(" <EXPR> "," <VAR> ")" ;
          call (<id>, <EXPR>, <VAR>)
<STMT>    = if <EXPR> eq <EXPR> then <STMT +> else <STMT +> fi ;
          ifthenelse (<EXPR>.1, <EXPR>.2, <STMT +>.1, <STMT +>.2)

{ expressions et operateurs }
<EXPR>    = <EXPR> <OP> <TERM> ;
          bin-expr (<OP>, <EXPR>, <TERM>)
<EXPR>    = <TERM> ;
<TERM>    = <VAR> ;
          used-var (<VAR>)
<TERM>    = "(" <EXPR> ")" ;
          <EXPR>
<TERM>    = <int> ;
          constant (<int>)
<OP>      = "+" ;
          plus ()
<OP>      = "-" ;
          minus ()
<OP>      = "*" ;
```

```
          mul ()
<OP>      = "/" ;
          div ()

{ variables }
<VAR>     = <id> ;
          simple-var (<id>)
<VAR>     = <id> "[" <EXPR> "]" ;
          indexed-var (<id>, <EXPR>)

{ terminals }
<id>      = %ID ;
          id () with $id := %ID end with
<int>     = %INT ;
          number () with $text := %INT end with
```

B.4 Le contenu du fichier `simproc.recor`

```
Titles
    "",
    "Warning:\t",
    "Error:\t" ;

Scanner
    Local
        1 2 3 4          ; "The invalid character \"" $0 "\" is
  deleted." ;
        X 1 2 3 4        ; "The invalid character \"" $0
                           "\" is replaced by \"" %0 "\"." ;
        X 0 1 2 3        ; "The character \"" %0 "\" is inserted before
                           \"" $0 "\"." ;

        Dont_Delete = {} ;
        Dont_Insert = {} ;

    Global
        Detection        : "\"%s\" is deleted." ;
        -- parameter     : character in error
        Keyword          : "This unknown keyword is erased." ;
        Eol              : "End Of Line" ;
        Eof              : "End Of File" ;
        -- The "character"
        Halt             : "Scanning stops on End Of File." ;
```

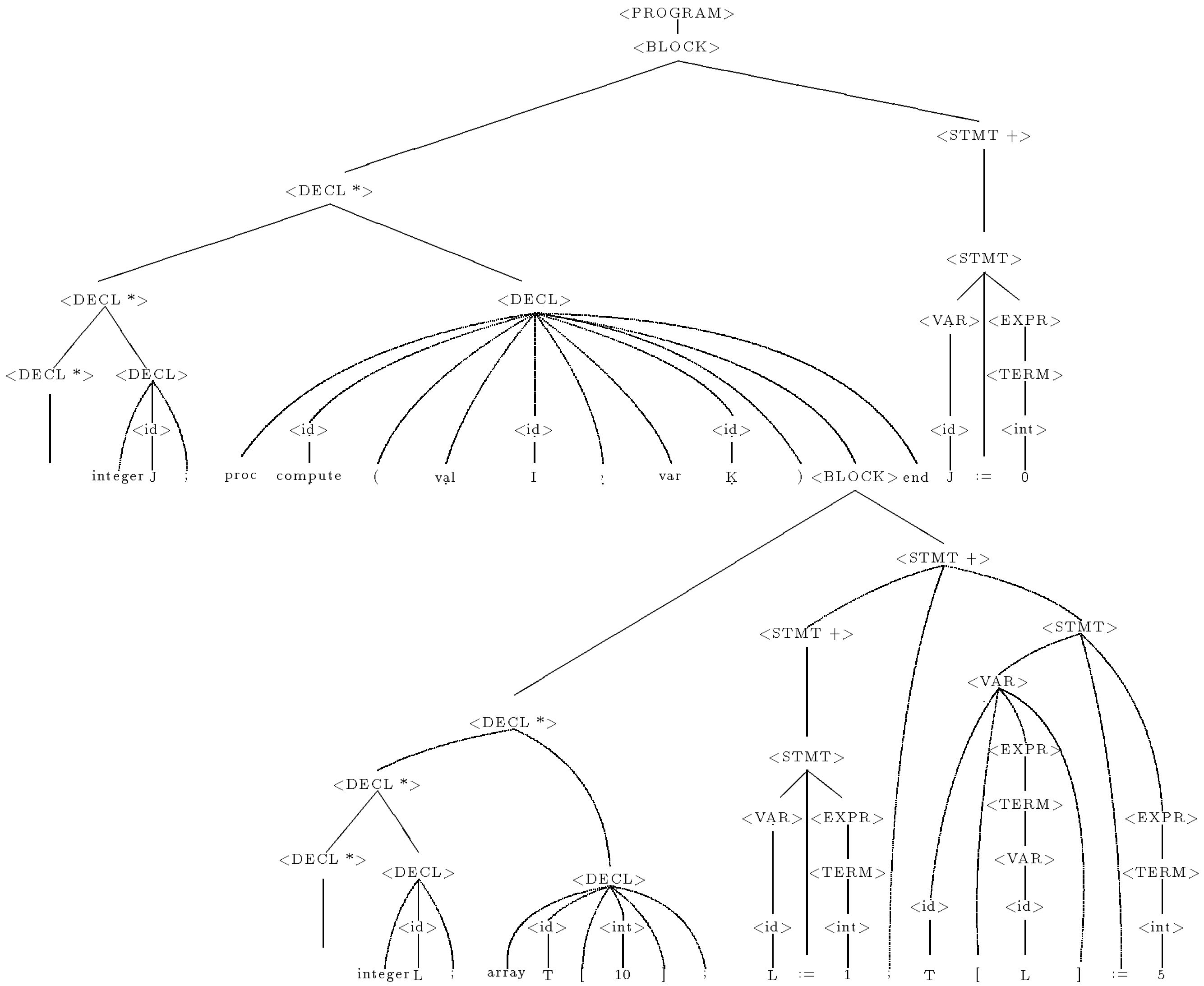

FIG. B.1 - *Syntaxe concrète de l'exemple page 26*

```
Parser
    Local
        0 S 2               ; "Misspelling of \"" $1 "\" which is replaced
                              by the keyword \"" %1 "\"." ;
        S 1                 ; "Misspelling of \"" $0 "\" before \"" $1
                              "\" which is replaced by the keyword
                              \"" %0 "\"." ;
        0 X 1 2 3           ; "\"" %1 "\" is inserted before \"" $1 "\"." ;
        0 X 2 3 4           ; "\"" $1 "\" is replaced by \"" %1 "\"." ;
        0 2 3 4             ; "\"" $1 "\" is deleted." ;
        0 X X 1 2 3 4       ; "\"" %1 " " %2 "\" is inserted before \"" $1
                              "\"." ;
        X 0 1 2 3           ; "\"" %0 "\" is inserted before \"" $0 " " $1
                              "\"." ;
        X 1 2 3 4           ; "\"" $0 "\" before \"" $1 "\" is replaced by
                              \"" %0 "\"." ;
        1 2 3 4             ; "\"" $0 "\" before \"" $1 "\" is deleted." ;
        X 2 3 4             ; "\"" $0 " " $1 "\" is replaced by \"" %0
                              "\"." ;
        X X 1 2 3           ; "\"" $0 "\" before \"" $1 "\" is replaced by
                              \"" %0 " " %1 "\"." ;

        Dont_Delete = {} ;
        Dont_Insert = {} ;

    Forced_Insertion
        "\"" %0 "\" is forced before \"" $1 "\"." ;

    Global
        Key_Terminals     = {} ;
        Validation_Length = 2 ;
        Followers_Number <= 5
                          : "\"%s\"^(, \"%s\"^) is expected" ;
        -- parameters : array (1:Followers_Number) of valid followers at
        -- detection point
        Detection         : "Global recovery." ;
        -- parameters: none
        Restarting        : "Parsing resumes on \"%s\"" ;
        -- parameters : array (1:Validation_Length) of valid followers
        -- at restarting point
        Eof               : "End Of File" ;
        -- The "token"
        Halt              : "Parsing stops on End Of File." ;
        -- parameters: none
```

```
Abstract
    "%d errors and %d warnings are reported." ;
    -- parameters: array (1:Titles_No) of number of messages
```

Le programme SIMPROC ci–dessous est lexicalement et syntaxiquement incorrect.

```
proc fact(val I, var O)
    begin
        integre L ;
        array T[10] ;
        L := 1 ;
        integer M ;
        T[L] := 5 ;
    end ;

    J := 0
```

L'analyseur engendré par SYNTAX à partir de la grammaire concrète donnée dans l'annexe B.3 page 26 produira, sur cet exemple, les messages d'erreur suivants :

```
line 3 : column 17 : Warning : Misspelling of "integre" before "L"
         wich is replaced by the keyword "integer".
line 6 : column 9 : Error : "end", "call", "if", "%ID" is expected
line 6 : column 9 : Error : Global recovery.
line 6 : column 9 : Warning : Parsing resumes on "integer"
```

La première ligne de message montre que SYNTAX ne s'arrête pas mais corrige une erreur de frappe.

Les messages relatifs à la ligne 6 indiquent une erreur de syntaxe, une déclaration d'entier figure après une instruction.

B.5 Le contenu du fichier simproc-base.asx

Une consultation en parallèle de la syntaxe concrète du fichier `simproc.atc` cf. page 26, peut amener un éclairage supplémentaire.

```
grammar simproc-base is

{ Une syntaxe abstraite non attribuée pour le  langage simproc
  avec un exemple de commentaire { imbriqué } }

root is PROGRAM ;

{ program & blocks }
```

```
PROGRAM     =  program ; { L'opérateur program du phylum PROGRAM }
program     -> BLOCK ; { BLOCK : phylum  unique de l'opérateur program }
BLOCK       =  block ;
block       -> DECLS STMTS ; { block est un opérateur d'arité fixe }

{ declarations }

DECLS       =  decls ;
decls       -> DECL* ; { decls est un opérateur liste }
DECL        =  int-decl array-decl proc-decl ; { Plusieurs opér. associés }
int-decl    -> ID { Identificateur de variable }  ;
array-decl  -> ID { Identificateur de tableau }
               NUMBER { La dimension }  ;
proc-decl   -> ID { Identificateur de la procédure }
               ID { L'identificateur du paramètre par valeur }
               ID { L'identificateur du paramètre par référence }
               BLOCK ;

{ instructions }

STMTS       =  stmt-list ;
stmt-list   -> STMT+ ;
STMT        =  assign call ifthenelse ;

assign      -> VAR EXPR ;
call        -> ID { Identificateur de la procédure }
               EXPR { Le paramètre par valeur }
               VAR { Le paramètre par reference }  ;
ifthenelse  -> EXPR EXPR { L'unique opérateur de comparaison est l'égalité }
               STMTS { partie vraie } STMTS { partie fausse }  ;

{ expressions }

EXPR        =  bin-expr constant used-var ;
bin-expr    -> OP EXPR EXPR ;
constant    -> NUMBER ;
used-var    -> VAR ;
OP          =  plus minus mul div ;
plus        -> ;
minus       -> ;
mul         -> ;
div         -> ;

{ variables }
```

```
VAR         =  simple-var indexed-var ;
simple-var  -> ID { identificateur de variable }  ;
indexed-var -> ID { identificateur de tableau } EXPR { l'index }  ;

{ terminals }

NUMBER      =  number ;
number      -> ;
ID          =  id;
id          -> ; { id -> token ; }

end grammar ; { simproc-base }
```

B.6 Le contenu du fichier simproc-in.asx

Dans cette syntaxe abstraite, on affecte des attributs de type **token** aux phyla NUMBER et ID définis dans la syntaxe abstraite importée simproc-base. C'est ce qu'on appelle une syntaxe abstraite attribuée.

```
grammar simproc-in is
    import grammar simproc-base ;

    root is PROGRAM ;

    { attributs }
    $text (NUMBER) : token ; { attribut text de type token
                               affecté au phylum  NUMBER }
    $id (ID)       : token ;
end grammar ;
```

B.7 Le contenu du fichier symbol-table.olga

Le module ci-dessous déclare une table des symboles. Il s'agit d'un module générique paramétré par deux types et une fonction.

```
declaration module symbol-table (type  key-type ;
                                 function < (x, y : key-type) : bool ;
                                 type info-type) is

{ Définition d'une table de symboles pour un langage à structure de blocs }
    type symbol-table ;
    exception  key-not-found () ; { Déclaration d'une nouvelle exception }
```

```
    value
       empty-table : symbol-table ;

    { Cette fonction est susceptible de lever l'exception key-not-found }
    function lookup (key : key-type ; st : symbol-table) : info-type
                                                        raise key-not-found ;
    function lookup-in-block (key : key-type ; st : symbol-table) : bool ;

    function insert (key : key-type ; info : info-type ; st :
                                           symbol-table) : symbol-table ;
    function enter-block (st : symbol-table) : symbol-table ;
end module ;
```

B.8 Le contenu du fichier linear-list.olga

```
definition module symbol-table is

{ Implémentation de la table des symboles utilisant une liste linéaire }
{ pour un langage structuré }

    type
        true-element = record
            key : key-type ;
            info : info-type ;
        end record ;

        symbol-table-element = union
           void { pour séparer les blocs }
           true-element ;
        end union ;

        symbol-table = list of symbol-table-element ;

    value
        empty-table := symbol-table (symbol-table-element (null)) ;

    function lookup (key : key-type ; st : symbol-table) : info-type
                                           raise key-not-found is
        if empty (st) then
            raise key-not-found ()
        else
            case head (st) match
               symbol-table-element (null : void) :
                   lookup (key, tail (st)) ;
```

```
                { sauter au dessus de la limite de bloc }
            symbol-table-element (node : true-element) :
                if node.key = key then
                    node.info
                else
                    lookup (key, tail (st))
                end if ;
         end case
     end if
 end function { lookup } ;

 function lookup-in-block (key : key-type ; st : symbol-table) : bool is
 { Ici nous n'avons pas besoin de tester si la table est vide puisqu'il
   y a toujours un bloc dans la table et la recherche n'ira pas au-delà }
     case head (st) match
       symbol-table-element (null : void) :
         false { Ne pas franchir la fontière de block} ;
       symbol-table-element (node : true-element) :
        if node.key = key then
          true
        else
          lookup-in-block (key, tail (st))
        end if ;
    end case
 end function { lookup-in-block } ;

 function insert (key : key-type ; info : info-type ; st :
                                       symbol-table) : symbol-table is
     symbol-table (symbol-table-element (true-element (key, info))) + st
 end function { insert } ;

 function enter-block (st : symbol-table) : symbol-table is
     symbol-table (symbol-table-element (null)) + st
 end function { enter-block } ;
end module ;
```

B.9 Le contenu du fichier binary-tree.olga

```
definition module symbol-table is

  { Implémentation de la table des symboles pour un programme à structure
    de blocs avec une liste d'arbres binaires }

  type
```

```
  tree-node = record
                key : key-type ;
                info : info-type ;
                left-son, right-son : bin-tree ;
              end record ;
and
  bin-tree = union
               void { for leaves } ;
               tree-node ;
             end union ;
  symbol-table = list of bin-tree ;

value
  leaf := bin-tree (null) { this value is local } ;
  empty-table := symbol-table (leaf) ;

function lookup-in-tree (key : key-type ; node : bin-tree) : info-type
    raise key-not-found is
  { Cette fonction est locale au module }
  case node match
    bin-tree (null : void) :
      raise key-not-found () ;
    bin-tree (tree-node (the-key, the-info, left-s, right-s)) :
      if the-key = key then
        the-info
      elsif the-key < key then
        lookup-in-tree (key, left-s)
      else
        lookup-in-tree (key, right-s)
      end if ;
  end case
end function { lookup-in-tree } ;

function lookup (key : key-type ; st : symbol-table) : info-type raise
    key-not-found is
  if empty (st) then
    raise key-not-found ()
  else
    lookup-in-tree (key, head (st))
    catch
      key-not-found :
        lookup (key, tail (st)) ;
    end catch
  end if ;
end function { end lookup }
```

```
function lookup-in-block (key : key-type ; st : symbol-table) : bool is
  { Utilisation d'une astuce pour éviter la réécriture
    du "lookup-in-tree" }
declare
  function dummy-eval (x : info-type): bool is true
  end function ;
in
  dummy-eval (lookup-in-tree (key, head (st)))
  catch
    key-not-found :
      false ;
  end catch
end function { end lookup-in-block } ;

function insert-in-tree (key : key-type ; info : info-type ; node :
    bin-tree) : bin-tree is
{ Cette fonction est également locale }
  case node match
    bin-tree (null : void) :
      { On insère ici pour partager au mieux }
      bin-tree (tree-node (key, info, node, node)) ;
    bin-tree (tree-node (the-key, the-info, left-s, right-s)) :
      if the-key = key then
        { remplacer l'ancienne entrée }
        bin-tree (tree-node (key, info, left-s, right-s))
      elsif the-key < key then
        bin-tree (tree-node (the-key, the-info, insert-in-tree (key, info,
    left-s), right-s))
      else
        bin-tree (tree-node (the-key, the-info, left-s, insert-in-tree
    (key, info, right-s)))
      end if ;
  end case
end function { end insert-in-tree } ;

function insert (key : key-type ; info : info-type ; st : symbol-table) :
    symbol-table is
  symbol-table (insert-in-tree (key, info, head (st))) + tail (st)
end function { insert } ;

function enter-block (st : symbol-table) : symbol-table is
  symbol-table (leaf) + st
end function { enter-block} ;
```

```
end module ;
```

B.10 Le contenu du fichier simproc-types.olga

Ici, entre autres choses, on adapte le module `simproc-table` cf. page 33 à notre langage Simproc. On peut constater combien il est facile de réutiliser un composant paramétré ! Le type **token** est choisi ainsi qu'un type énuméré pour décrire les différents modes de variables.

```
declaration module  simproc-types  is

  type
      mode   =  (varid,  refid,  arrayid,  procid) ;
      { Pour identifier les variables simples, les paramètres par valeur,
        les paramètres par référence,  les tableaux et procédures }
      operator  =  (wrong-op,  addition,  subtraction,
                    multiplication,  division) ;

      { On appelle un module paramétré !!! }
      from  symbol-table  (token,  <,  mode)  import  all ;

end module ;
```

B.11 Le contenu du fichier simproc-out.asx

```
grammar simproc-out is
    import grammar simproc-base ;
    from simproc-types import  all ; { pour les types symbol-table
                                       et operator}
    root is PROGRAM ;

    $symtab (BLOCK)    : symbol-table ; { $symtab attribut de
                                          type symbol-table }
    $op (OP)           : operator ;
    $value (NUMBER)    : token ; { remplace $text de la syntaxe d'entrée }
    $id (ID)           : token ; { Le seul attribut apparaissant aussi dans
                                   "simproc-in" }
    $correct (PROGRAM) : bool ;
end grammar ;
```

B.12 Le contenu du fichier simproc-check.olga

Les points à vérifier en fonction des contraintes sémantiques de la page 25 sont les suivants :

- unicité de déclaration de chaque identificateur dans chaque bloc ;
- conformité de l'utilisation d'un identificateur avec sa déclaration ;
- validité des paramètres passés par valeur ou par référence ;

```
attribute grammar  simproc-check  (simproc-in,  simproc-out) :   is
    from  simproc-types  import all ;

    attribute
         { attributs de travail }
         compound  $s-temp-st  and  $h-temp-st  (BLOCK,  DECLS,  DECL) :
                                                          symbol-table ;
         synthesized  $correct1  (BLOCK,  DECLS,  DECL) :  bool ;
         synthesized  $correct2  (BLOCK,  DECLS,  DECL,  STMT,  EXPR,  VAR,
                                  NUMBER) :  bool ;
         { les attributs "résultats" }
         synthesized  $value  () ;
         synthesized  $op  () ;
         global  $symtab  () ;
         synthesized  $correct  () ;

        { Les règles sémantiques }
        { program et blocks  }
        where  program  ->  BLOCK  use
            $correct  :=  $correct1(BLOCK)  &  $correct2(BLOCK) ;
            $h-temp-st(BLOCK)  :=  empty-table ;
            $symtab(BLOCK)  :=  $s-temp-st(BLOCK) ;
        end where ;

        where  null-PROGRAM  ->  use
            $correct  :=  false ;
        end where ;

        where  block  ->  DECLS  STMT  use
            $correct1  :=  $correct1(DECLS) ;
            $correct2  :=  $correct2(DECLS)  &  $correct2(STMT) ;
            $s-temp-st  :=  $s-temp-st(DECLS) ;
        end where ;

        where  null-BLOCK  ->  use
            $correct1  :=  false ;
```

```
    $correct2  :=  false ;
end where ;

{ déclarations  }
where  decls  ->  DECL*  use
    $correct1  :=
                    map left  &
                         value  true
                         other  $correct1(DECL)
                    end map ;
    $correct2  :=
                    map left  &
                        value  true
                        other  $correct2(DECL)
                    end map ;
    { Cette définition est donnée uniquement pour fournir un exemple
      d'utilisation du "case position".  Ceci aurait pu être généré
      automatiquement par les règles de copie par défaut }
    $h-temp-st(DECL):= case position is
                           first  :   $h-temp-st ;
                           other  :   $s-temp-st(DECL.left) ;
                       end case ;
end where ;

where  null-DECLS  ->  use
    $correct1  :=  false ;
    $correct2  :=  false ;
end where ;

where  int-decl  ->  ID
    declare value
        correct  :=
            if  $id(ID)  =  error-token  then
                false
            elsif  lookup-in-block  ($id(ID),  $h-temp-st) then
                message  "Identificateur déjà déclaré."
                          position  ID
                          severity  2
                          value  false
            else
                true
            end if ;
    use
        $correct1  :=  correct ;
        $correct2  :=  true ;
```

```
            $s-temp-st := if  correct  then
                              insert  ($id(ID),  varid,  $h-temp-st)
                          else
                              $h-temp-st
                          end if ;
    end where ;

    where  array-decl  ->  ID  NUMBER
        declare value
            correct  :=
                (if  $id(ID)  =  error-token  then
                     false
                 elsif lookup-in-block  ($id(ID), $h-temp-st)  then
                     message  "Identificateur déjà déclaré."
                              position  ID
                              severity  2
                              value  false
                 else
                     true
                 end if)
                &
                ($correct2(NUMBER)  and
                 if  $value(NUMBER)  =  0  then
                     message  "Size  cannot  be  null"
                              position  NUMBER
                              severity  2
                              value  false
                 else
                     true
                 end if) ;
        use
            $correct1  :=  correct ;
            $correct2  :=  true ;
            $s-temp-st  := if  correct  then
                               insert  ($id(ID),  arrayid,  $h-temp-st)
                           else
                               $h-temp-st
                           end if ;
    end where ;

    where  proc-decl  ->  ID1  ID2  ID3  BLOCK
        declare value
            correct  :=
                (if  $id(ID1)  =  error-token  then
                     false
```

```
              elsif  lookup-in-block  ($id(ID1), $h-temp-st)  then
                  message  "Identificateur déjà déclaré."
                           position  ID1
                           severity  2
                           value  false
              else
                  true
              end if)
             &
             (if  $id(ID2)  =  error-token  or
                  $id(ID3)  =  error-token  then
                  false
              elsif  $id(ID2)  =  $id(ID3)  then
                  message "Erreur : paramètres formels identiques !"
                          position  ID3
                          severity  2
                          value  false
              else
                  true
              end if) ;
     use
         $correct1  :=  correct ;
         $correct2  :=  $correct1(BLOCK)  &  $correct2(BLOCK) ;
         $s-temp-st  := if  correct  then
                              insert  ($id(ID1),  procid,  $h-temp-st)
                          else
                              $h-temp-st
                          end if ;
         $h-temp-st(BLOCK)  := insert  ( $id(ID3),  refid,
                                         insert  ( $id(ID2),  varid,
                                                   enter-block
                                                  ( $symtab[BLOCK] )
                                                 )
                                       ) ;
         $symtab(BLOCK) :=  $s-temp-st(BLOCK) ;
end where ;

where  null-DECL  ->  use
   $correct1  :=  false ;
   $correct2  :=  false ;
end where ;

{ instructions  }
where  stmt-list  ->  STMT+  use
   $correct2  := map left  &
```

```
                      value  true
                      other  $correct2(STMT)
                 end map ;
end where ;

where  assign  ->  VAR  EXPR  use
    $correct2  :=  $correct2(VAR)  &  $correct2(EXPR) ;
end where ;

where  ifthenelse  ->  EXPR1  EXPR2  STMT1  STMT2  use
    $correct2  :=  $correct2(EXPR1)  &  $correct2(EXPR2)  &
                   $correct2(STMT1)  &  $correct2(STMT2) ;
end where ;

where  call  ->  ID  EXPR  VAR  use
    $correct2  := (if  $id(ID)  =  error-token  then
                        false
                  else
                        if  lookup  ($id(ID),  $symtab[BLOCK])
                            !=  procid  then
                            message  "Ce n'est pas une procédure."
                            position  ID
                            severity  2
                            value  false
                        else
                            true
                        end if
                        catch
                            key-not-found :
                            message  "Identificateur non déclaré."
                                 position  ID
                                 severity  2
                                 value  false ;
                        end catch
                  end if)  &
                  $correct2(EXPR)  &  $correct2(VAR) ;
end where ;

where  null-STMT  ->  use
    $correct2  :=  false ;
end where ;

{ expressions  et  opérateurs  }
where  bin-expr  ->  OP  EXPR1  EXPR2  use
    $correct2  :=  $correct2(EXPR1)  &  $correct2(EXPR2)  &
```

```
                  $op(OP)!=  wrong-op ;
end where ;

where  constant  ->  NUMBER  use
end where ;

where  EXPR  ->  VAR  use
end where ;

where  null-EXPR  ->  use
   $correct2  :=  false ;
end where ;

where  plus  ->  use
   $op  :=  addition ;
end where ;

where  minus  ->  use
   $op  :=  subtraction ;
end where ;

wheremul  ->  use
   $op  :=  multiplication ;
end where ;

where  div  ->  use
   $op  :=  division ;
end where ;

where  null-OP  ->  use
   $op  :=  wrong-op ;
end where ;

{ variables  }
where  simple-var  ->  ID  use
   $correct2  :=
                 if  $id(ID)  =  error-token  then
                      false
                 else
                      let  mode :=  lookup  ($id(ID),
                                      $symtab[BLOCK])
                      in  if  mode  =  varid  or
                          mode  =  refid  then
                               true
                          else
```

```
                                message  "Ce n'est pas une variable."
                                    position  ID
                                    severity  2
                                    value  false
                            end if
                        catch
                            key-not-found :
                            message  "Identificateur non déclaré."
                                 position  ID
                                 severity  2
                                 value  false ;
                        end catch
                   end if ;
end where ;

where  indexed-var  ->  ID  EXPR  use
    $correct2 := (if  $id(ID)  =  error-token  then
                        false
                   else
                        if  lookup  ($id(ID),  $symtab[BLOCK])
                            !=  arrayid  then
                            message  "Ce n'est pas un tableau."
                                 position  ID
                                 severity  2
                                 value  false
                        else
                            true
                        end if
                        catch
                            key-not-found :
                            message  "Identificateur non déclaré."
                                 position  ID
                                 severity  2
                                 value  false ;
                        end catch
                   end if)  &  $correct2(EXPR) ;
end where ;

{ numbers  }
where  number  ->  use
    $correct2  :=  $text !=  error-token ;
    $value  := if  $text  =  error-token  then
                   0
                else
                   int  (string  ($text))
```

```
                end if ;
    end where ;
   end grammar  { simproc-check  } ;
```

B.13 Le contenu du fichier simproc-term.olga

La grammaire attribuée décrite est destinée à traduire le texte source d'un programme écrit en SIMPROC dans un type de données abstrait décrivant ce langage [MG78]. Cette grammaire sera exécutée après la grammaire simproc-check.olga décrite page 39.

```
attribute grammar  simproc-term  (simproc-out) :   (simproc-adt)  is
    from  simproc-types  import all ;

    attribute
          { Pour le calcul des termes de l'arbre abstrait source }
    synthesized  $Program  (PROGRAM) :  Program ;
    synthesized  $Model  (BLOCK,  DECLS,  DECL,  STMT) :   Source-modif ;
    synthesized  $Var  (VAR) :  Var ;
    synthesized  $Value  (EXP,  NUMBER) :  Int ;
    inherited    $procid  (BLOCK) :  token  { name  of  enclosing  proc}

    { Règles sémantiques }
    { programm et  blocks }

    where  program  ->  BLOCK  use
       $Program  :=  Execute  ($Model(BLOCK)) ;
       $procid(BLOCK)  :=  token  ("Prog")  { On l'invente... }
    end where ;

    where  null-PROGRAM  ->  use
         { Ceci ne devrait jamais se produire puisque le texte source a été
           vérifié afin d'être complètement correct au moment de
           l'exécution de cette grammaire }
         $Program  :=  Execute  (Void  ()) ;
    end where ;

    where  block  ->  DECLS  STMT  use
        $Model :=  { Essayer de construire l'arbre minimum }
                 case  $Model(DECLS)  match
                      Void  ()  { pas de déclaration }
                      $Model(STMT) ;
                      Concat-all  (D)  { plusieurs déclarations }
                      case  $Model(STMT)  match
                           Concat-all  (S)  { plusieurs instructions }
```

```
                        Concat-merge  (D,  S) ;
                        S  { une instruction unique }
                        Concat-post  ($Model(DECLS),  S) ;
                   end case ;
                   D  { une unique déclaration }
                   case  $Model(STMT)  match
                        Concat-all  (S)  { plusieurs instructions }
                        Concat-pre  (D,  $Model(STMT)) ;
                        S  { une seule déclaration }
                        Concat  (D,  S) ;
                   end case ;
                end case ;
 end where ;

where  null-BLOCK  ->  use
    $Model  :=  Void  () ;
end where ;

{ déclarations  }
where  decls  ->  DECL*  use
     $Model  :=
              case arity is
                   0  :   Void  () ;
                   1  :   $Model(DECL) ;
                   other  :   map left  Concat-post
                                 value  Concat  ()
                                 other  $Model(DECL)
                             end map ;
              end case ;
end where ;

where  null-DECLS  ->  use
    $Model  :=  Void  () ;
end where ;

where  int-decl  ->  ID  use
    $Model  :=  Intdecl  (VarId  ()
                       with  $id  :=  $id(ID)
                       end with,
                       ProcId  ()
                       with  $id  :=  $procid[BLOCK]
                       end with) ;
end where ;

where  array-decl  ->  ID  NUMBER  use
```

```
    $Model  :=   Arraydecl  (ArrayId  ()
                        with  $id  :=  $id(ID)
                        end with,
                        $Value(NUMBER),
                        ProcId  ()
                        with  $id  :=  $procid[BLOCK]
                        end with) ;
end where ;

where  proc-decl  ->  ID1  ID2  ID3  BLOCK  use
    $Model  :=   Procdecl  (ProcId  ()
                         with  $id  :=  $id(ID1)
                         end with,
                         VarId  ()
                         with  $id :=  $id(ID2)
                         end with,
                         RefId  ()
                         with  $id  :=  $id(ID3)
                         end with,
                         ProcId  ()
                         with  $id  :=  $procid[BLOCK]
                         end with,
                         $Model(BLOCK)) ;
    $procid(BLOCK)  :=  $id(ID1) ;
end where ;

where  null-DECL  ->  use
    $Model  :=  Void  () ;
end where ;

{ instructions }
where  stmt-list  ->  STMT+  use
    $Model  := case arity is
                 1  :   $Model(STMT) ;
                 other  :   map left  Concat-post
                               value  Concat  ()
                               other  $Model(STMT)
                            end map ;
              end case ;
end where ;

where  assign  ->  VAR  EXPR  use
    $Model  :=
             case  $Address(VAR)  match
                 Refersto  (AR)  :
```

```
                     Assignvar  (AR,  $Value(EXPR)) ;
                other  :
                     Assignint  ($Address(VAR),  $Value(EXPR)) ;
           end case ;
end where ;

where  ifthenelse  ->  EXPR1  EXPR2  STMT1  STMT2  use
    $Model  :=  Condit  ($Value(EXPR1),  $Value(EXPR2),
                         $Model(STMT1),  $Model(STMT2)) ;
end where ;


where  call  ->  ID  EXPR  VAR  use
     $Model  :=  Proccall  (ProcId  ()  { callee  }
                   with  $id  :=  $id(ID)
                   end with,
                   $Value(EXPR),
                   $Address(VAR),
                   ProcId  ()  { caller  }
                   with  $id  :=  $procid[BLOCK]
                   end with) ;
end where ;

where  null-STMT  ->  use
    $Model  :=  Void  () ;
end where ;

{ expressions et opérateurs }
where  bin-expr  ->  OP  EXPR1  EXPR2  use
    $Value  :=
           case  $op(OP)  is
                addition,  wrong-op  :
                     Plus  ($Value(EXPR1),  $Value(EXPR2)) ;
                subtraction  :
                     Minus  ($Value(EXPR1),  $Value(EXPR2)) ;
                multiplication  :
                     Times  ($Value(EXPR1),  $Value(EXPR2)) ;
                division  :
                Quotient  ($Value(EXPR1),  $Value(EXPR2)) ;
           end case ;
end where ;

where  constant  ->  NUMBER  use
end where ;
```

```
where  EXPR  ->  VAR  use
   $Value  :=  Valueof  ($Address(VAR)) ;
end where ;

where  null-EXPR  ->  use
    $Value  :=
             Const  (Number  ()
                  with  $value  :=  0
                  end with) ;
end where ;

{ variables  }
where  simple-var  ->  ID  use
    $Address  :=
             let  id  :=  $id(ID)
             in  case  lookup  (id,  $symtab[BLOCK])  is
                       refid  :
                       Refersto  (Refdesignates  (
                                    RefId  ()
                                    with  $id  :=  id
                                    end with,
                                    Currentcall  ())) ;
                       other  :   { includes  "varid"  }
                       Designates  (VarId  ()
                                with  $id  :=  id
                                end with,
                                Currentcall  ()) ;
                end case ;
end where ;

where  indexed-var  ->  ID  EXPR  use
    $Address  :=
                  Element  (ArrayId  ()
                       with  $id  :=  $id(ID)
                       end with,
                       Currentcall  (),
                       $Value(EXPR)) ;
end where ;

{ numbers  }

where  number  ->  use
    $Value  :=
             Const  (Number  ()
                  with  $Value  :=  $value
```

```
                     end with) ;
   end where ;

end grammar  { simproc-term  }
```

Bibliographie

[ACA91] Jeffrey Ullman Alfred Claudia Aho, Ravi Sethi. *Compilateurs, Principes techniques et outils.* InterEditions, 1991.

[Jou91] Jean-Philippe Jouve. Réalisation du décompilateur d'arbres attribués du système FNC–2 PPAT (a pretty–printer for attributed trees), Juillet 1991. Rapport d'option. Directeur de stage : Monsieur Jourdan INRIA.

[JP94] Martin Jourdan and Didier Parigot. The FNC–2 system user's guide and reference manual. Technical Report Release 1.18, INRIA Rocquencourt (France), November 1994.

Index