CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Centre agréé de GRENOBLE (CUEFA)

MEMOIRE

présenté en vue d'obtenir le

DIPLOME D'INGENIEUR CNAM

en INFORMATIQUE

par

Bruno VIVIEN

Etude et réalisation d'un compilateur E-LOTOS à l'aide du générateur de compilateurs SYNTAX/FNC-2

Soutenu le 22 décembre 1997

JURY Président Véronique DONZEAU-GOUGE

Membres Christian CARREZ
Jacques COURTIN
Hubert GARAVEL
Mihaela SIGHIREANU

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Centre agréé de GRENOBLE (CUEFA)

MEMOIRE

présenté en vue d'obtenir le

DIPLOME D'INGENIEUR CNAM

en INFORMATIQUE

par

Bruno VIVIEN

Etude et réalisation d'un compilateur E-LOTOS à l'aide du générateur de compilateurs SYNTAX/FNC-2

Les travaux relatifs à ce mémoire ont été effectués au sein de l'action VASY de l'INRIA sous la direction de M. Hubert Garavel et Melle Mihaela Sighireanu.

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 mémoire ;
- 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 Christian Carrez pour sa participation au jury ;
- 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 son équipe ;
- Mademoiselle Mihaela Sighireanu, étudiante en thèse à l'Inria Rhône–Alpes, pour l'aide qu'elle m'a apportée tout au long de ce travail, et pour ses nombreux conseils en matière de programmation fonctionnelle ;
- Monsieur Alain Landelle, Responsable du Département Formations Supérieures au Centre Universitaire d'Education et de Formation des Adultes de Grenoble, pour la confiance qu'il m'a accordée dans la poursuite du cycle ingénieur Cnam ;
- Monsieur Maurice Ozil, Directeur du laboratoire Exploitation des Réseaux d'Accès au Centre National d'Etudes des Télécommunications de Grenoble, pour m'avoir accordé un congé de formation d'un an afin de pouvoir préparer ce mémoire ;
- Monsieur Pierre Boullier, Directeur de Recherche à l'Inria Rocquencourt, pour son aide dans l'utilisation de l'outil Paradis ;
- Monsieur Didier Parigot, Chargé de Recherche à l'Inria Rocquencourt et Professeur à l'école Polytechnique, pour l'aide qu'il nous a apportée en nous rendant visite et en échangeant avec nous, plus de 200 messages électroniques et pour la version Solaris ainsi que les différentes améliorations du système Syntax/Fnc-2, qu'il a bien voulu réaliser pour répondre à nos besoins ;
- Monsieur Philippe Deschamp, de l'Inria Rocquencourt, pour son aide dans l'utilisation de l'outil Paradis ;
- L'ensemble de l'équipe Vasy avec laquelle j'ai eu de nombreux échanges fructueux ;
- 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.

Table des matières

Chapitre 1

Introduction

1.1 Situation technique et scientifique

Dans le domaine du développement des systèmes informatiques complexes (notamment les systèmes distribués et temps-réel), on appelle *méthodes formelles*, un ensemble de techniques d'ingénierie, souvent à base mathématique, permettant de spécifier un comportement à vérifier et de vérifier son bon fonctionnement. Elles offrent un cadre rationnel comportant des outils de modélisation et de raisonnement, reposant souvent sur l'utilisation d'un langage formel de spécification.

Progressivement, les méthodes formelles diffusent de la recherche universitaire vers les applications industrielles. La part, sans cesse croissante, d'informatique dans les systèmes embarqués et l'essor actuel des télécommunications avec l'avènement prochain des autoroutes de l'information, vont se traduire par des exigences fortes en termes de logiciels certifiés, pour des applications sécuritaires. Dans ces domaines, où le zéro défaut est nécessaire, notamment lorsque des vies ou des équipements coûteux sont en jeu, l'utilisation de méthodes formelles pour spécifier les applications informatiques vitales, est recommandée.

Au cours des années 80, les organismes de normalisation internationaux (ISO[1]/IEC[2] et CCITT[3]) ont défini trois langages (ESTELLE[4], LOTOS[5] et SDL[6]) pour la description formelle des protocoles de communication et des systèmes distribués. Ces langages permettent de spécifier avec précision le comportement dynamique de systèmes complexes, composés de processus parallèles communicants avec les avantages suivants :

- un langage formel possède une sémantique sûre s'il repose sur des théories mathématiques éprouvées ;

[1]International Organization for Standardization
[2]International Electrotechnical Commission
[3]Comité Consultatif International pour la Téléphonie et la Télégraphie
[4]Extended Finite State Machine Language
[5]Language Of Temporal Ordering Specification
[6]Specification and Description Language

- un langage formel offre la possibilité de prouver les propriétés attendues d'un système, au niveau de la spécification et du code qui peut en dériver ;
- ils permettent de décrire les fonctionnalités d'un système sans faire état de détails de réalisation qui imposeraient des contraintes inutiles aux implémenteurs ;
- les descriptions formelles offrent aux implémenteurs un point de départ à partir duquel ceux-ci peuvent — par raffinements successifs — produire une implémentation conforme, avec l'avantage que les choix de réalisation sont clairement distingués des caractéristiques des algorithmes ;
- ils permettent de faire baisser considérablement les efforts de test et de maintenance des logiciels.

Ces trois langages ont connu un certain succès dans l'enseignement, la recherche et l'industrie. De nombreux outils logiciels ont été développés pour compiler, analyser et vérifier les descriptions écrites dans ces langages.

En particulier, l'INRIA Rhône–Alpes et le laboratoire VERIMAG ont développé la boîte à outils CÆSAR/ALDEBARAN pour la compilation et la vérification de descriptions LOTOS [FGM+92, FGK+96, Gar96]. Cette boîte à outils a été diffusée dans plus de 125 sites.

Cependant, ces trois langages normalisés montrent actuellement leurs limites, pour plusieurs raisons :

- D'abord, certains choix de conception initiaux doivent être révisés pour tenir compte des critiques des utilisateurs. C'est notamment le cas pour les techniques de description des données : ni les types abstraits algébriques (utilisés en SDL et LOTOS), ni le langage PASCAL (choisi pour ESTELLE) ne donnent vraiment satisfaction à l'usage.
- Ensuite, on assiste à l'émergence de nouveaux protocoles de communication qui, contrairement aux protocoles de la génération précédente, font intervenir une notion de temps quantitatif. C'est le cas pour les protocoles des réseaux à hauts débits (notamment ATM[7]), ainsi que pour les applications multimédias (synchronisation audio/vidéo, par exemple). Or, les trois langages normalisés ne permettent pas d'exprimer les aspects temporels, ou seulement de manière trop rudimentaire par rapport aux besoins exprimés.
- Plus généralement, l'apparition de nouvelles architectures de communication (réseaux intelligents, réseaux à hauts débits, systèmes ouverts distribués comme ODP[8] ou CORBA[9]) remet en cause le modèle OSI[10] et son architecture statique à sept couches qui avaient présidé à la définition des langages ESTELLE, LOTOS et SDL.

[7] Asynchronous Transfer Mode
[8] Open Distributed Processing
[9] Common Object Representation Brooker Architecture
[10] Open System Interconnection

C'est pourquoi, il est nécessaire de faire évoluer ces langages afin de les adapter aux besoins actuels. C'est ce qu'a entrepris l'ISO/IEC en lançant l'initiative d'une révision majeure de la norme LOTOS, qui doit déboucher sur un langage appelé E-LOTOS (pour *Extended* LOTOS). Actuellement, plusieurs pays (Belgique, Canada, Espagne, France, Japon, Pays-Bas, Roumanie, Royaume-Uni) participent à cet effort de normalisation.

1.2 Cadre institutionnel

L'INRIA et l'équipe VASY collaborent activement à ce travail de normalisation, notamment en formulant diverses propositions d'extensions du langage LOTOS [Gar94a, Gar94b, Gar95b, Gar95a, Gar95c, GS95a, SG95a, SG95b, GS95b, SG95c, GS96b, GS96a, SG96, JGL+95].

Le travail présenté dans ce document, a été réalisé au sein de l'action VASY[11] de l'INRIA Rhône–Alpes, sous la direction d'Hubert Garavel et de Mihaela Sighireanu.

L'action VASY a débuté en 1995, c'est à la fois, une action de recherche INRIA Rhône–Alpes et une action du GIE[12] BULL-INRIA (DYADE). L'objectif général de l'action VASY porte sur :

- les outils et méthodes pour le développement de systèmes distribués asynchrones basés sur :
  - l'application des méthodes formelles, notamment la norme LOTOS ;
  - des compilateurs de langages formels avec notamment, la boîte à outils CADP (CÆSAR/ALDEBARAN Development Package) développée à l'INRIA Rhône–Alpes ;
  - des outils de vérification, comme XTL.
- des applications industrielles dans le cadre d'une coopération avec BULL, avec par exemple, l'étude de l'architecture POWERSCALE développée par BULL et utilisée dans les stations de travail et les serveurs de la gamme ESCALA. Un sous-ensemble significatif de cette architecture a été modélisé formellement dans le langage LOTOS.

La cible d'application de l'action VASY est l'architecture multiprocesseurs POLYKID de BULL, développée à BULL Italie et notamment le protocole de cohérence de caches de POLYKID.

Dans une démarche à plus long terme, les doctorants de l'équipe VASY travaillent sur l'amélioration des techniques de spécification et de vérification :

- Radu Mateescu étudie et implémente une extension du μcalcul modal par des variables typées, ce qui devrait permettre de vérifier plus aisément des propriétés portant sur des signaux avec valeurs, comme celles vérifiées pour l'architecture POWERSCALE [CGM+96] ;

[11] Validation des systèmes
[12] Groupement d'Intérêt Economique

- Mihalea Sighireanu contribue à la définition de la future norme E-LOTOS, qui améliore significativement le langage LOTOS, notamment dans la perspective d'une utilisation industrielle. C'est avec elle que ce mémoire a été réalisé. Le sujet de ce travail consiste à réaliser une première implémentation du langage E-LOTOS, actuellement en cours de développement. Pour ce faire, compte-tenu que la définition de E-LOTOS n'est pas figée, mais évolue, nous avons décidé d'utiliser des outils de haut niveau, pour la génération de compilateurs. Notre choix s'est porté sur SYNTAX/FNC-2 développé à l'INRIA Rocquencourt.

1.3 Travail effectué

L'essentiel du travail effectué dans le cadre de ce mémoire, porte sur la conception et le développement d'un prototype de compilateur, pour le langage E-LOTOS. Il s'inscrit dans la continuité du sujet de probatoire CNAM que nous avons effectué en mai-juin 1996 [Viv96]. Ce compilateur, nommé TRAIAN[13], doit devenir la base d'un futur atelier logiciel pour E-LOTOS, en offrant des fonctionnalités de simulation et de vérification pour le langage E-LOTOS.

D'autres outils (comme des paragrapheurs pour les langages LOTOS et E-LOTOS) ont été réalisés en tant que sous-produits de notre compilateur.

1.4 Organisation du document

Ce mémoire est organisé comme suit :

- le chapitre 2 (*Objectifs et contraintes*) présente nos objectifs et l'environnement dans lequel nous avons travaillé pour les atteindre ;
- le chapitre 3 (*Solutions et choix*) détaille plus en profondeur chacun des sous-objectifs et donne les éléments de choix qui nous ont fait opter pour telle ou telle solution ;
- le chapitre 4 (*Réalisation*), plus technique que les précédents, donne une vision des méthodes que nous avons employées et du travail réalisé ;
- enfin, le chapitre 5 (*Conclusion*), est un retour sur les chapitres précédents, en termes de bilan et de perspectives.

[13] Pour faire suite à CÆSAR développé par l'INRIA Rhône–Alpes, nous avons choisi le nom de Trajan (Marcus Ulpius Trajanus). Né à Italica en 53, il succéda à Nerva et fut Empereur Romain de 98 jusqu'à sa mort en 117. Il se montra excellent administrateur et fut un grand bâtisseur. Par les guerres de Dacie (101-107), il assura la sécurité des frontières sur le Danube et par son action en Orient (114-117), il étendit l'empire jusqu'à l'Arabie Pétrée, l'Arménie et la Mésopotamie.

Chapitre 2

Objectifs et contraintes

Ce chapitre présente les objectifs à atteindre (section 2.1) ainsi que l'existant et les contraintes à respecter (section 2.2).

2.1 Objectifs à atteindre

Dans cette section, nous présenterons tour à tour, les quatre objectifs suivants :

- la réalisation d'une souche de compilateur pour le langage E-LOTOS ;
- la réalisation d'un traducteur de E-LOTOS vers LOTOS ;
- la réalisation de paragrapheurs pour les langages E-LOTOS et LOTOS ;
- la réalisation de jeux de tests.

2.1.1 Réalisation d'une souche de compilateur pour le langage E-LOTOS

L'objectif principal est de réaliser des outils logiciels pour le langage E-LOTOS et de réduire le laps de temps entre la parution d'une nouvelle norme et la création d'un prototype de compilateur.

La première partie du travail consiste à définir la partie avant (*front-end*) du compilateur. Cette partie regroupe :

- l'analyse lexicale ;
- l'analyse syntaxique ;
- l'analyse de sémantique statique (liaison des identificateurs, calculs de types. . .).

2.1.2 Réalisation d'un traducteur de E-Lotos vers Lotos

Le second objectif est de réaliser une partie de l'outil de simulation et de vérification pour E-Lotos. Aussi, le compilateur à réaliser (nommé Traian), doit prendre en entrée une spécification E-Lotos et produire en sortie une spécification Lotos qui est la traduction de la spécification E-Lotos en faisant abstraction des aspects temps réel. La spécification Lotos peut être ensuite compilée en utilisant des outils déjà existants pour Lotos (notamment la boîte à outils Cadp développée à l'Inria Rhône–Alpes).

La traduction repose sur les travaux de Mihaela Sighireanu qui a spécifié des algorithmes sous forme de grammaires attribuées (12 pages).

2.1.3 Réalisation de paragrapheurs

Le troisième objectif consiste à réaliser des outils de paragraphage pour les langages E-Lotos et Lotos.

2.1.4 Réalisation de jeux de tests

Le quatrième objectif consiste à créer une base de tests pour le compilateur à réaliser. Il faut également réaliser un outil permettant de détecter automatiquement la non-conformité d'un test avec une référence établie.

2.2 Existant et contraintes

Dans cette section, nous présentons l'ensemble des éléments qui existaient au début du projet, ainsi que l'ensemble des contraintes à respecter.

2.2.1 Cadp

Cadp (Cæsar/Aldebaran Development Package), est une boîte à outils pour la compilation et la vérification de programmes Lotos [GJM+97, Gar96, FGK+96].

Pour la production de notre compilateur, ainsi que pour la production des paragrapheurs, nous devons réutiliser au maximum les descriptions de la partie avant (front-end) du compilateur Cæsar pour Lotos.

2.2.2 Travaux du « Committee Draft »

La première révision du standard Lotos a eu lieu dans le groupe de travail de l'Iso nommé « Enhancement to Lotos » (en abrégé : E-Lotos). Elle a donné lieu à la publication, en février 1997, du *Committee Draft* [Queer] qui a été soumis à un vote international en juin 1997 et qui a reçu l'adhésion de 11 pays.

Le langage E-LOTOS défini dans ce document, est un langage de spécification formel pour décrire des protocoles et des systèmes complexes. Il repose sur la plupart des principes qui ont été à la base du développement du langage LOTOS :

- une définition mathématique : ce document contient la grammaire E-LOTOS ainsi que les règles de sémantique statique et dynamique sous forme de règles de Plotkin ;
- des constructions aptes à la description de systèmes complexes : abstraction (possibilité de cacher des informations), indépendance de l'implémentation...

Le langage E-LOTOS tel qu'il est défini dans le Committee Draft, s'intéresse seulement au pouvoir d'expression du langage, mais pas à la possibilité pratique d'une implémentation du langage, ni aux performances de l'implémentation éventuelle.

Dans le but d'avoir un langage qui se compile facilement tout en gardant intacte le pouvoir d'expression du langage, les participants grenoblois (Hubert Garavel et Mihaela Sighireanu) ont proposé une variante du langage qui peut être compilée avec de bonnes performances [SG96]. Cette variante se nomme E-LOTOS User Language.

2.2.3 Environnement matériel et logiciel multi-architectures

Les outils à réaliser doivent fonctionner sous UNIX, dans les environnements SUN-OS et SOLARIS. A terme, on prévoit aussi un portage sous d'autres systèmes d'exploitation comme LINUX.

2.2.4 Flexibilité des outils

L'implémentation du compilateur TRAIAN dépend de la définition du langage E-LOTOS. Or, les deux définitions du langage sont instables, aussi bien celle du Committee Draft, que celle de la variante développée dans l'action VASY RA. Aussi, il est souhaitable d'utiliser un système de haut niveau pour la génération de compilateurs. Il est également souhaitable d'adopter les principes des langages déclaratifs [PRDJ95]. Grâce à ces principes, le concepteur d'un compilateur spécifie des règles à utiliser sans se soucier de l'ordre dans lequel elles sont évaluées. De même, dans un but d'évolutivité, on cherchera à utiliser des outils nous débarrassant de tout détail d'implémentation.

Pour ces raisons, il est exclu, dès le début, de programmer le compilateur en C/C++.

Chapitre 3

Solutions et choix

Ce chapitre présente d'abord, un éventail de solutions existantes en matière d'outils dédiés à la production de compilateurs et de paragrapheurs, puis il décrit les choix techniques effectués.

De tels outils, s'ils sont bien conçus et robustes, représentent une aide appréciable dans la production de compilateurs. Par exemple, il est beaucoup plus facile d'obtenir un analyseur syntaxique correct, en applicant un constructeur d'analyseur à la description grammaticale du langage, que de l'implémenter directement à la main [ASU91].

La recherche d'outils adéquats, est l'objet principal de ce chapitre. Elle se fera en tenant compte des critères suivants :

Rapidité de développement : dans le cas d'un langage en cours de définition, il est avantageux de disposer rapidement d'un compilateur pour vérifier le bien fondé des constructions ;

Evolutivité : l'évolutivité des divers composants du compilateur pour répondre aux ajustements de la norme E-Lotos est également un critère important, car nous devons pouvoir prendre en compte ces changements sans grand bouleversement dans le code déjà écrit ;

Sûreté des développements : il s'agira de produire des composants fiables, ce qui implique l'utilisation d'outils robustes ;

Compatibilité : les outils utilisés devront être compatibles entre eux.

La structure générale d'un compilateur est donnée sur la figure 3.1. La partie *analyse lexicale* correspond à la section 3.2, la partie *analyse syntaxique* à la section 3.3, la partie *analyse sémantique* à la section 3.4 et la partie *génération de code cible* à la section 3.5.

Les définitions nécessaires à la compréhension de ces différentes parties, sont regroupées dans la section suivante.

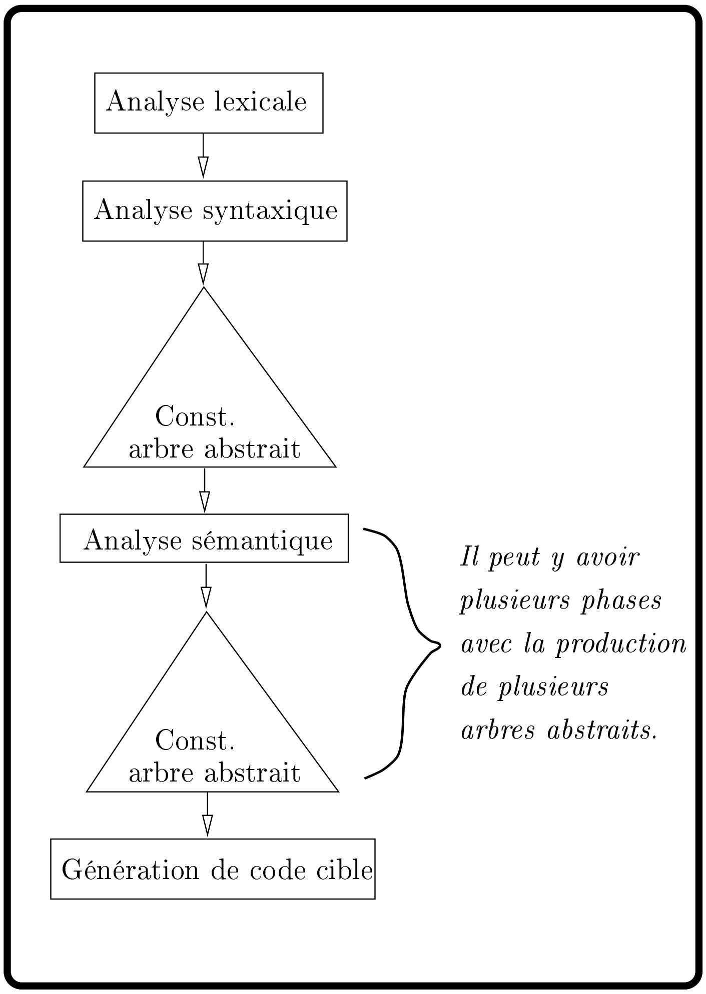

Figure 3.1: Structure d'un compilateur

3.1 Définitions

3.1.1 Définitions relatives à l'analyse lexicale

Classe de caractères : une classe ou *alphabet*, dénote tout ensemble fini de symboles. Par exemple, $0, 1$ est l'alphabet binaire et $[A - Za - z]$ est une expression régulière représentant la classe *lettre*.

Unité lexicale : une unité lexicale reconnaît un ensemble de chaînes en entrée. Les unités lexicales reconnues dépendent du langage.

Modèle d'unité lexicale : c'est une règle décrivant une unité lexicale. Un modèle filtre chaque chaîne d'une unité lexicale.

Lexèmes ou terminaux : ce sont des suites de caractères du programme source qui concordent avec le modèle d'une unité lexicale. On peut les classer en deux catégories :

- les terminaux *génériques* tels que les identificateurs et les littéraux, qui peuvent recouvrir un ensemble théoriquement infini de possibilités pour le programme et qui doivent obligatoirement être définis au niveau lexical ;
- les autres terminaux, tels que les mots clés, les symboles spéciaux...

Analyseur lexical : L'analyseur lexical (on dit aussi « scanner » ou « lexer ») constitue la première phase d'un compilateur. Son rôle est de lire un programme source sous forme d'une suite de caractères et de décomposer cette suite en une succession d'unités lexicales.

Les unités lexicales reconnues sont ensuite mises à disposition de l'analyseur syntaxique décrit dans la section suivante et dont le rôle est de découvrir une unité structurelle dans un groupe d'unités lexicales. L'analyseur lexical joue le rôle de réservoir à symboles pour l'analyseur syntaxique.

3.1.2 Définitions relatives à l'analyse syntaxique

Grammaire hors-contexte ou BNF : elles sont apparues en 1956 [Cho56] et furent utilisées comme spécifications *déclaratives* du langage ALGOL 60. Une grammaire hors-contexte (en abrégé *grammaire*), comprend quatre composants :

- Un ensemble d'unités lexicales appelées symboles terminaux ;
- Un ensemble de non-terminaux ;
- Un ensemble de productions où chaque production est syntaxiquement composée d'un non-terminal, suivi d'une flèche, suivie d'une suite d'unités lexicales et de non-terminaux. Le non-terminal situé à gauche de la flèche est appelé *partie gauche* et ce qui se trouve à droite de la flèche est appelé *partie droite* (voir l'exemple ci-dessous) ;

– Un axiome, qui désigne un des non-terminaux comme symbole de départ de la grammaire.

L'exemple suivant permet de décrire les expressions composées de chiffres et des signes + et −, comme $1 + 2 - 3$, $1 - 2$, 7. L'axiome est le non terminal « `EXPRESSION` ».

Dans la suite du document, nous prendrons comme convention d'écrire les non-terminaux en majuscules et les terminaux en minuscules.

```
EXPRESSION -> LISTE ;

LISTE -> LISTE + CHIFFRE ;
LISTE -> LISTE - CHIFFRE ;
LISTE -> CHIFFRE ;

CHIFFRE -> 0 ;
CHIFFRE -> 1 ;
CHIFFRE -> 2 ;
...
CHIFFRE -> 9 ;
```

Arbre syntaxique : un arbre syntaxique représente le regroupement des unités lexicales du programme source en structures grammaticales qui seront employées par le compilateur pour synthétiser son résultat [ASU91]. Dans un arbre syntaxique, on garde la trace des terminaux du langage.

Analyseur syntaxique : l'analyseur syntaxique (on dit aussi « parser ») est chargé de découvrir la structure du programme à partir d'une liste d'éléments lexicaux fournie par l'analyseur lexical.

Il sait comment sont construites les *unités syntaxiques*, comme les expressions, les instructions et les listes de telles constructions.

Les résultats de l'analyseur syntaxique peuvent se présenter sous plusieurs formes, mais la forme la plus pratique est l'arbre syntaxique. Son but est de fournir aux étapes ultérieures, une représentation syntaxique du texte source du programme.

3.1.3 Définitions relatives à l'analyse sémantique

Définition dirigée par la syntaxe : une définition dirigée par la syntaxe est une généralisation d'une grammaire hors-contexte, dans laquelle chaque symbole de la grammaire possède un ensemble d'attributs, partitionné en deux sous-ensembles : les *attributs synthétisés* et les *attributs hérités* par ce symbole.

Attribut synthétisé : si l'on calcule l'attribut d'un nœud de l'arbre syntaxique, en fonction des valeurs des attributs de ses fils, on dit que cet attribut est *synthétisé*.

Attribut hérité : si l'on calcule l'attribut d'un nœud de l'arbre syntaxique, en fonction des valeurs des attributs de ses frères ou du père, on dit que cet attribut est *hérité*.

Grammaire attribuée : une grammaire attribuée est une extension d'une grammaire hors-contexte pour laquelle on associe à chaque symbole 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.

Les *grammaires attribuées* constituent un formalisme commode pour décrire les traitements effectués par un compilateur. 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 [PRDJ95].

L'exemple ci-dessous, représente une grammaire attribuée pour la traduction des expressions décrites page 12, en une notation postfixée. On utilise un seul attribut (« `$s_A` »), de type synthétisé. Par convention, nous ferons précéder les attributs synthétisés par « `$s_` » et les attributs hérités par « `$h_` ». Lorsqu'un non-terminal apparaît plusieurs fois dans une règle de production, on lui ajoute un indice : ainsi « `$s_A (LISTE_0)` » représente l'attribut synthétisé « `$s_A` », du nom terminal « `LISTE` » de la partie gauche.

Pour effectuer la concaténation des attributs, on suppose, dans notre exemple, qu'il existe une fonction « `Concatener` » qui prend deux chaînes en entrée et retourne une chaîne en sortie, représentant la concaténation des deux chaînes.

```
EXPRESSION -> LISTE ;

LISTE -> LISTE + CHIFFRE ;
      $s_A (LISTE_0) :=
          Concatener (Concatener ($s_A (LISTE_1), $s_A (CHIFFRE)), "+")

LISTE -> LISTE - CHIFFRE ;
      $s_A (LISTE_0) :=
          Concatener (Concatener ($s_A (LISTE_1), $s_A (CHIFFRE)), "-")

LISTE -> CHIFFRE ;
      $s_A (LISTE) := $s_A (CHIFFRE)

CHIFFRE -> 0 ;
      $s_A (CHIFFRE) := "0"

CHIFFRE -> 1 ;
```

```
      $s_A (CHIFFRE) := "1"

CHIFFRE -> 2 ;
     $s_A (CHIFFRE) := "2"
...

CHIFFRE -> 9 ;
     $s_A (CHIFFRE) := "9"
```

Arbres abstraits : un arbre abstrait est une représentation compacte de l'arbre syntaxique dont nous avons donné une définition page 11. Contrairement à l'arbre syntaxique qui représente toutes les dérivations possibles, l'arbre abstrait cherche à donner une représentation plus simple, plus synthétique et mieux adaptée aux traitements sémantiques. Au niveau des traitements sémantiques, les terminaux du niveau syntaxique n'ont plus d'intérêt, c'est pourquoi les mots clés du langage ne sont pas conservés dans l'arbre abstrait. Un exemple d'arbre abstrait est donné sur la figure figure 3.2.

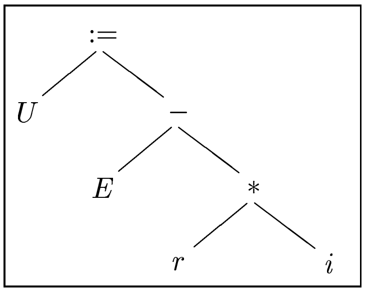

Figure 3.2: Arbre abstrait pour l'expression $U := E - ri$

3.2 Analyse lexicale

3.2.1 Besoins

Même si les techniques d'analyse lexicale sont bien connues, l'écriture manuelle d'un analyseur lexical reste délicate. On préfère se concentrer sur la sémantique du problème plutôt que sur l'aspect de la reconnaissance. Il peut alors être utile de recourir à des générateurs d'analyseurs lexicaux. Parfois plus lents que les analyseurs écrits à la main, ils sont intéressants, car fiables et disponibles très rapidement.

Afin d'avoir une conception plus simple et plus facilement modifiable de notre compilateur, nous avons besoin d'un outil capable de générer un analyseur lexical de spécifications E-LOTOS. Un générateur d'analyseur lexical doit pouvoir :

- lire des caractères du texte en entrée ;

- identifier et produire des entités lexicales pour l'analyseur syntaxique ;
- sauvegarder les lexèmes dans une table (par exemple, la table des identificateurs qui fait partie de la table des symboles) ;
- éliminer les commentaires du texte source ;
- garder la trace des colonnes et numéros de lignes où apparaissent les diverses unités lexicales ;
- générer des actions définies par l'utilisateur au niveau des lexèmes ;
- permettre à l'utilisateur de spécifier des classes de caractères par des expressions régulières ;
- gérer certaines fonctions de macro-traitement, comme les inclusions de fichiers.

3.2.2 Solutions existantes

Lex : (produit AT&T), est le plus célèbre des outils de génération d'analyseurs lexicaux. Il assure les fonctions décrites ci-dessus, à part la gestion de la table des identificateurs et les inclusions de fichiers. Il est disponible sur plusieurs plates-formes (Unix, Ms-dos, Windows). Il produit du code C qu'il faut compiler et lier au code d'un analyseur syntaxique qui sera vu comme un client de Lex.

Flex : (produit GNU) est une version de Lex publique et libre de droits. Il est disponible sous Ms-dos, sous de nombreuses versions d'Unix et sous OS/2.

Flex++ : est une version de Flex, produisant des classes C++.

Aflex : (produit du projet Arcadia, à l'Université de Californie) similaire à Lex, il permet la génération d'analyseurs lexicaux à partir de descriptions données par des expressions régulières. Il est écrit en Ada et génère du code Ada.

Dlg : Dlg (DFA-based Lexical Analyzer Generator) a été réalisé par Terence Parr et Will Cohen, c'est l'analyseur lexical de l'outil Pccts (Purdue Compiler-Construction Tool Set). Il fonctionne comme Lex et Flex et permet de produire du code C et C++. Il est disponible sous Unix, Ms-dos et Windows.

Genlex : est un générateur d'analyseur lexical basé sur le langage Caml : c'est une fonction prenant en entrée une liste de caractères et retournant une liste de tokens.

Lexical_Analyser_G : (produit de Software Engineering Laboratory à Lausanne) est un générateur d'analyseurs basé sur le langage Ada.

Lecl : Lecl fait partie de l'outil Syntax qui est un générateur d'analyseurs lexico-syntaxiques développé par l'Inria. C'est un constructeur d'analyseurs lexicaux qui prend en entrée, d'une part des expressions régulières fournies par l'utilisateur, et d'autre part, des informations extraites d'une table produite par le processeur du niveau syntaxique. Son rôle est de produire un automate d'états finis capable d'effectuer la reconnaissance des terminaux du niveau syntaxique. Il assure l'ensemble des fonctions attendues dans la section 3.2.1 plus quelques fonctionnalités supplémentaires, comme la reconnaissance des mots clés figurant dans la grammaire syntaxique. LECL est disponible sous UNIX et LINUX.

REX : est l'analyseur lexical du système GMD, il produit du code C et MODULA-2. REX est disponible sous MS-DOS et WINDOWS.

ELI : est un outil de génération de compilateurs, intégrant un générateur d'analyseurs lexicaux. Basé sur les expressions régulières, il génère du code C et C++. ELI est disponible sous UNIX et LINUX.

3.3 Analyse syntaxique

Tout langage de programmation a des règles qui prescrivent la structure syntaxique des programmes bien formés [ASU91].

Afin de permettre l'exécution des différentes étapes d'un compilateur, il est nécessaire de s'assurer que la syntaxe du texte source est correcte. A cette fin, il est possible d'écrire un analyseur syntaxique à la main, mais ceci n'est pas recommandé si des changements ultérieurs sont à prévoir dans la syntaxe du langage, ce qui est notre cas.

La structure syntaxique des langages de programmation peut être décrite par des grammaires hors-contexte (présentées en 3.1.2), ou notation BNF (Backus-Naur-Form), à partir desquelles on peut générer automatiquement des analyseurs syntaxiques fiables.

De plus, l'utilisation d'une grammaire BNF, pour décrire la syntaxe d'un langage, permet de déterminer des ambiguïtés au plus tôt dans la construction du compilateur. Dans notre cas, le langage E-LOTOS est en cours de définition et l'utilisation d'une grammaire donnant une spécification précise, peut révéler des erreurs de définition ou des constructions ambiguës.

Les discussions dans le forum COMP.COMPILERS montrent l'intérêt qu'il y a à produire des analyseurs syntaxiques automatiquement, notamment en matière de récupération sur erreur : beaucoup trop d'énergie est dépensée encore aujourd'hui, dans les tâches d'écriture de tels modules.

3.3.1 Besoins

Un générateur d'analyseurs syntaxiques doit être capable de :

- accepter une description de la grammaire du langage sous une forme proche de la syntaxe BNF ;
- coopérer efficacement avec un générateur lexical en limitant au maximum l'écriture de code intermédiaire pour interfacer ces outils ;

- traiter les erreurs de syntaxe en indiquant l'endroit où elles se produisent ;
- gérer des stratégies de récupération sur erreur ;
- construire un arbre abstrait au cours de l'analyse.

3.3.2 Solutions existantes

On trouve de nombreuses solutions concernant la construction d'analyseurs syntaxiques, on ne pourra pas toutes les citer ici.

Yacc : (produit Unix) est le plus célèbre des constructeurs d'analyseurs. Il permet de générer des analyseurs de grammaires Lalr et d'accepter des grammaires ambiguës, sous condition d'une résolution de la part de l'utilisateur. Yacc est disponible sur Ms-dos, Linux et Unix. Mais il est très rudimentaire dans plusieurs de ses aspects :

- Du point de vue traitement des erreurs, l'utilisateur doit écrire ses propres routines, ce qui peut s'avérer délicat à réaliser (localisation et affichage de l'erreur, tentative de correction. . .) ;
- A une réduction de production, on peut associer une fonction écrite en C, avec tout ce que cela comporte comme facilités, mais aussi comme dangers, à savoir la possibilité de réaliser des effets de bord, des allocations de mémoire. . .
- En ce qui concerne la gestion des attributs, seule la manipulation d'attributs synthétisés est réellement agréable. L'utilisation d'attributs hérités est possible, mais l'utilisateur doit déterminer leur place dans la pile d'exécution pour y avoir accès. On préférerait pouvoir déclarer un attribut hérité typé sur un non-terminal et dans des conditions sûres (L'attribut a-t-il été défini ? Est-il correctement typé ? etc).
- L'utilisateur ne peut pas réellement découper son travail en modules séparés. On aimerait pouvoir spécifier les traitements sémantiques ailleurs que dans la grammaire syntaxique.
- Il faut encore écrire du code C et l'interfaçage avec l'analyseur lexical (souvent généré par son compagnon Lex) requiert de l'utilisateur des tâches manuelles, comme l'écriture de « makefiles ».
- Les noms de fichiers produits sont imposés par Yacc.

Bison : (produit GNU) présente une compatibilité ascendante avec Yacc. Tout comme Yacc, Bison présente une interface avec le langage C. Bison est disponible sous Ms-dos, Unix et Linux. Il souffre des mêmes limitations que Yacc, sauf pour les noms de fichiers générés qui ne sont pas imposés.

Bison++ : (produit GNU) est similaire à Bison. C'est un générateur d'analyseurs syntaxiques produisant des classes C++, il est disponible sous Ms-dos, Unix et Linux.

Eli : est un générateur de compilateurs possédant un générateur d'analyseurs syntaxiques. Il est disponible sous Unix et Linux.

Syntax : l'outil Syntax est disponible sous Linux et Unix. Il intègre l'ensemble des fonctions que nous avons listées à la section 3.3.1, avec notamment un système sophistiqué de récupération sur erreur (géré par l'outil Recor). Une spécification destinée au traitement des erreurs doit être décrite dans un fichier, mais en général, on utilise un fichier standard qui a déjà fait ses preuves dans le recouvrement des erreurs sur divers langages.

Syntax permet de reconnaître la structure syntaxique d'un texte source sur la base de la reconnaissance d'une grammaire Bnf. Comme pour Yacc et Bison, on a la possibilité d'effectuer des actions sémantiques pendant l'analyse, mais ce principe contribue au mélange des phases du processus de compilation et ne permet pas d'avoir une structure modulaire correcte des compilateurs.

Pour pallier ce problème, Syntax permet de créer facilement un arbre abstrait qui sera utilisé, dans les phases suivantes par des modules distincts. A la différence de Yacc, Syntax dispose, d'un constructeur d'arbres nommé Atc (Abstract Tree Constructor) qui repose sur une syntaxe abstraite de description d'arbres nommée Asx (Abstract Syntax).

3.4 Analyse sémantique

Certaines propriétés des programmes ne peuvent être décrites à l'aide d'une grammaire hors-contexte. La vérification qu'une variable est bien déclarée ou la cohérence de type sont des exemples de ces propriétés. Ces propriétés sont généralement décrites par des prédicats sur les informations du contexte [RD94].

Le but de l'analyse sémantique est de vérifier que ces propriétés sont vraies et, qu'en cas d'anomalie, l'utilisateur en sera informé par l'émission d'un message clair.

3.4.1 Besoins

Il est maintenant bien établi qu'en matière d'analyse sémantique, une définition dirigée par la syntaxe est une aide appréciable. Pour cette raison, nous nous orientons vers les grammaires attribuées pour spécifier nos traitements sémantiques. Pour l'analyse sémantique, on voudrait que les points suivants puissent être satisfaits :

- On peut vouloir découper les traitements sémantiques en plusieurs phases. A cette fin, les outils utilisés devront nous permettre d'écrire nos grammaires attribuées de manière modulaire.
- Ecrire les grammaires attribuées implique un ordre de calcul implicite dans les règle sémantiques. L'outil que nous utiliserons pour générer automatiquement des évaluateurs devra naturellement prendre cet ordre en compte.
- Certaines règles sémantiques reviennent régulièrement ; ce sont souvent des règles de recopie d'attributs servant à transmettre une valeur d'attribut de proche en proche

dans l'arbre abstrait. L'écriture de ces règles représente un travail fastidieux que l'on souhaite éviter. L'outil utilisé devra être capable de générer automatiquement les règles de recopie manquantes ;

- De plus, il serait préférable que les évaluateurs d'attributs ne nous contraignent pas à écrire les règles sémantiques aux nœuds de l'arbre où il n'y a « rien à dire ». Ce cas est fréquent lorsque l'on modularise les grammaires écrites, un module ne se préoccupant que d'un ensemble restreint de nœuds.

  Il serait préférable que les règles sémantiques manquantes soient produites automatiquement par les outils.

3.4.2 Solutions existantes

Lido et Oil [GHL+92] : Lido est un générateur de grammaires attribuées et Oil est un outil permettant de réaliser du filtrage sur les arbres (pattern–matching). Ces deux outils appartiennent au système de génération de compilateurs Eli mais ils ne disposent pas de syntaxe abstraite permettant de générer les arbres. Le code produit est soit du C, soit du C++. Ces outils sont disponibles sous Unix et Linux.

Gmd [Jan85] : est une boîte à outils pour la génération de compilateurs. Elle permet la construction d'arbres décrits par une syntaxe abstraite. On peut faire du filtrage pour réaliser des transformations d'arbres. Les règles sémantiques peuvent être exprimées avec des grammaires attribuées et l'on dispose d'un générateur d'évaluateurs d'attributs. Cette boîte à outils est disponible uniquement sous Ms-dos et Windows.

Fnc-2 [JP90] : comporte deux parties :

- Un compilateur pour le langage Olga ;
- Un générateur d'évaluateurs d'attributs.

Olga est un langage fonctionnel spécialisé dans le traitement des grammaires attribuées. Il est fortement typé et n'autorise pas les effets de bord. Il dispose de nombreuses fonctions qui permettent une programmation sûre et offre une grande souplesse pour la modularité et la réutilisabilité des modules.

Olga permet la construction et la manipulation des arbres abstraits décrits avec le langage Asx.

Les fonctionnalités cités en 3.1.3 sont toutes supportées par Fnc-2.

3.5 Traduction

3.5.1 Besoins

Après la phase de contrôles sémantiques, nous aurons besoin de traduire la spécification E-Lotos en entrée de notre compilateur, vers une spécification Lotos. Pour réaliser cette phase de traduction, il serait souhaitable de pouvoir satisfaire les points suivants :

- Pour produire une spécification Lotos à partir d'un arbre abstrait, il faut avoir un moyen d'exprimer des sorties pour chacun des nœuds de l'arbre. Nous avons donc besoin d'un langage dans lequel nous puissions donner une représentation ASCII de la valeur de nos attributs, mais aussi dans lequel l'on puisse générer un texte quelconque. Ce genre d'outils est parfois appelé « *décompilateur d'arbre* » (« unparser »).

- Puisque l'on souhaite produire une spécification Lotos en sortie de notre compilateur, il est préférable de produire du code lisible pour faciliter la mise au point lors de la compilation du code Lotos avec le compilateur Cæsar. Un décompilateur d'arbre nous offrant la possibilité de produire des sorties indentées apportera un confort supplémentaire.

- Enfin, il serait souhaitable que ce langage soit aussi proche que celui permettant d'exprimer les grammaires attribuées du niveau sémantique, ceci afin de simplifier la tâche d'apprentissage de l'écrivain du compilateur et d'uniformiser les outils.

3.5.2 Solutions existantes

Ptg [GHL+92] : est le décompilateur d'arbres abstraits du générateur de compilateurs Eli. Il produit du code C ou C++ et il est disponible sous Unix et Linux.

Sorcerer [Par97] : est le décompilateur d'arbres abstraits du générateur de compilateur Pccts. Il permet la décompilation d'arbres décrits par une syntaxe Bnf.

Ppat [Jou91] : fait partie de Fnc-2 ; c'est un langage permettant de donner une représentation concrète des arbres attribués manipulés par Fnc-2. Ppat est très proche d'Olga, aussi bien du point de vue syntaxique que sémantique. La puissance d'expression est aussi grande que celle d'Olga, aussi, la décompilation d'arbres complexes se fait de manière très flexible. Le code produit est du C et le langage est disponible sous Unix.

3.6 Production de paragrapheurs

3.6.1 Définitions

Afin d'uniformiser la présentation des sources, il est très intéressant de disposer au plus tôt d'outils adaptés comme des *paragrapheurs*. Les paragrapheurs sont des programmes prenant en entrée un texte source et fournissant en sortie le même texte source mais présenté différemment. Ceci veut dire qu'ils n'ajoutent aucune information au texte source, comme des *caractères de contrôle* par exemple, mais se limitent à l'ajout de *sauts de lignes* et d'*espaces* pour la présentation du texte. Les paragrapheurs ne se contentent pas d'effectuer une simple indentation des lignes, ils effectuent une analyse plus poussée du texte, de façon à produire la séparation en lignes, ce qui permet de mieux distinguer les notions du langage.

Nous avons besoin d'un outil de paragraphage qui accepte en entrée, soit un arbre syntaxique, soit un arbre abstrait dans lequel toutes les informations du programme source ont été conservées (comme les mots clés et les commentaires).

3.6.2 Besoins

Réaliser des outils de paragraphage, implique certaines exigences :

- Une équipe de développeurs sera plus efficace si elle est confrontée à des programmes dont la forme textuelle est toujours la même, par exemples avec des indentations bien définies et des commentaires placés toujours aux mêmes endroits.
- Cependant, chaque utilisateur peut avoir ses propres préférences au niveau de la mise en page. Un paragrapheur ne doit donc pas être trop contraignant mais au contraire, permettre une certaine souplesse dans la mise en page.
- Pour rester cohérent avec les techniques de génération de compilateurs dirigés par la syntaxe, nous avons souhaité utiliser un outil de paragraphage dirigé, lui aussi, par la syntaxe.
- Un paragrapheur dirigé par la syntaxe, nécessite l'utilisation d'une grammaire BNF pour décrire le langage dans lequel les textes à paragrapher sont écrits. L'utilisation d'une grammaire BNF nous permet d'avoir une exigence supplémentaire : la signalisation des erreurs syntaxiques.
- pour limiter les temps de développement, nous avons voulu réutiliser le maximum de modules déjà écrits, notamment ceux ayant servi à l'analyse lexico-syntaxique des langages LOTOS et E-LOTOS.

3.6.3 Solutions existantes

PPAT [Jou91] : le décompilateur d'arbres attribués que nous avons présenté à la section 3.5.2 est une solution : le langage de boîtes utilisé par PPAT permet de réaliser des sorties paragraphées. Nous pouvons réaliser un paragrapheur par une décompilation de l'arbre issu de l'analyse lexico-syntaxique, à condition de spécifier la conservation des commentaires au niveau de l'analyseur lexical (spécification LECL). Cette solution a l'inconvénient de nécessiter l'écriture d'une spécification avec le langage PPAT, et ne permet pas de satisfaire la deuxième exigence citée en 3.6.2. Effectivement, si un utilisateur souhaite adapter le paragrapheur à ses propres préférences, il devra d'abord apprendre la langage PPAT, puis effectuer une modification de la spécification.

PARADIS [BD85] : c'est un système de paragraphage dirigé par la syntaxe. L'intérêt de ce système est qu'il repose sur une syntaxe BNF identique à celle manipulée par le générateur d'analyseurs syntaxiques SYNTAX. Il permet de générer un paragrapheur basé sur la mise en forme de la grammaire BNF décrivant le langage. C'est donc un avantage considérable, car ce système permet de réutiliser la description de la grammaire concrète BNF, qu'il suffit de mettre en forme manuellement pour produire une sortie adaptée aux exigences de l'utilisateur. De par sa simplicité, il offre une certaine rigidité. En revanche aucune programmation n'est nécessaire et les temps de développement d'un paragrapheur avec ce système sont imbattables. En outre, ce système permet la correction automatique de certaines erreurs dans le texte source.

3.7 Tableau récapitulatif et choix

Le tableau page 23, offre une vue synthétique des principaux outils pour la génération de compilateurs. Les colonnes correspondent aux fonctionnalités attendues et les lignes correspondent aux noms d'outils. Un symbole « $\surd$ » à l'intersection d'une ligne et d'une colonne signifie que la fonctionnalité est assurée pour l'outil correspondant.

Dans certains cas, lorsque la ligne correspond à un ensemble d'outils intégrés (et non à un outil isolé), le nom de l'outil correspondant est indiqué à la place du symbole « $\surd$ ».

Compte tenu des résultats indiqués dans le tableau, nous avons choisi le système SYNTAX/FNC-2, qui dispose de fonctionnalités pour la production de compilateurs. Il nous a semblé que cet ensemble d'outils était le seul à proposer l'ensemble des fonctionnalités attendues par ce projet (voir les besoins exprimés dans les sections 3.2.1 à 3.6.2).

3.8 Présentation du système choisi : SYNTAX/FNC-2

SYNTAX/FNC-2 est un système de génération de compilateurs, basé sur les grammaires attribuées. Le développement du système à l'INRIA Rocquencourt se poursuit actuellement sous la conduite de M. Didier Parigot.

Dans cette section, nous présentons les quatres principales parties du système :

- La première partie repose sur l'outil ASX qui permet de vérifier et compiler les syntaxes abstraites utilisées pour la description d'arbres abstraits.
- La deuxième partie comprend l'outil SYNTAX développé par l'équipe « Langages et Traducteurs » de l'INRIA Rocquencourt [BD88]. C'est un générateur d'analyseurs lexico-syntaxiques capables de corriger automatiquement certaines erreurs du texte source.

  Il peut être utilisé indépendamment de FNC-2, comme il l'a été pour le développement des compilateurs CÆSAR [GS90, Gar90, Gar89a], CÆSAR.ADT [GT93, Gar89b] et XTL [Mat97].

  Mais il peut être aussi utilisé conjointement avec FNC-2 : c'est l'approche que nous avons choisie. Dans ce cas, certaines adaptations doivent être apportées à la version

Comparaison de divers outils de génération de compilateurs

	Grammaires attribuées	Analyseur lexical	Analyseur syntaxique	Evaluateur d'attributs	Gestion avancée des erreurs lexico-syntaxiques	Générateur d'arbres abstraits	Pattern matching sur les arbres	Transformation d'arbres abstraits	Décompilateur d'arbres attribués
ALPHA	√	FLEX	BISON	√					
BISON			√						
BISON++			√						
ELI	√	√	√					√	√
FLEX		√							
FLEX++		√							
GMD		REX	ELL	√		√	√		
JACCL			√						
LEX		√							
LLGEN			√						
PCCTS		√	√						
PRE-CC			√						
SORCERER								√	√
SYNTAX/FNC-2	√	LECL	BNF	FNC-2	RECOR	ATC	√	√	√
T-GEN			√			√			
YACC			√						

standard de Syntax : l'outil Bnf est remplacé par l'outil Atc qui étend ses fonctions pour effectuer la construction du premier arbre abstrait.

Cette partie frontale permet d'analyser un texte source, afin de construire un premier arbre abstrait.

- La troisième partie comprend les évaluateurs nécessaires aux calculs d'attributs. L'outil Fnc-2, spécialisé dans le traitement des grammaires attribuées, possède une grande puissance d'expression : il accepte une large classe de grammaires attribuées que sont les grammaires attribuées *fortement non circulaires* (FNC)[14]. 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 fonctionnel de type ML restreint au premier ordre.
- La quatrième partie permet de produire des résultats par décompilation d'arbres attribués. Il s'agit de l'outil Ppat.

La figure suivante représente schématiquement la structure de l'application prototype Simproc qui est livrée avec le système Syntax/Fnc-2. On pourra s'y reporter pour identifier les différentes parties présentées ci-après.

3.8.1 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.

Notion de syntaxe abstraite

Les entrées et sorties des évaluateurs d'attributs générés par Fnc-2 sont des arbres attribués décrits par des syntaxes abstraites.

Le formalisme utilisé dans la description des arbres abstraits s'appuie sur les notions de *phyla* (pluriel de *phylum*) et d'*opérateurs*, que l'on peut respectivement assimiler aux *non–terminaux* et aux *productions* d'une grammaire hors-contexte. Nous précisons maintenant ces notions :

Phylum : un phylum est un type d'arbre abstrait sur lequel sont définis des opérateurs. Il peut être vu comme un modèle pour la construction d'arbres issus d'un non-terminal d'une grammaire usuelle (voir l'exemple ci-dessous donné après la définition des opérateurs). C'est une notion syntaxique du langage.

Un phylum donné peut servir d'opérande à plusieurs opérateurs ou même servir d'opérande à un ou plusieurs de ses propres opérateurs, ce qui introduit une notion de récursivité dans la description des arbres abstraits.

[14] D'où le nom du système Fnc-2

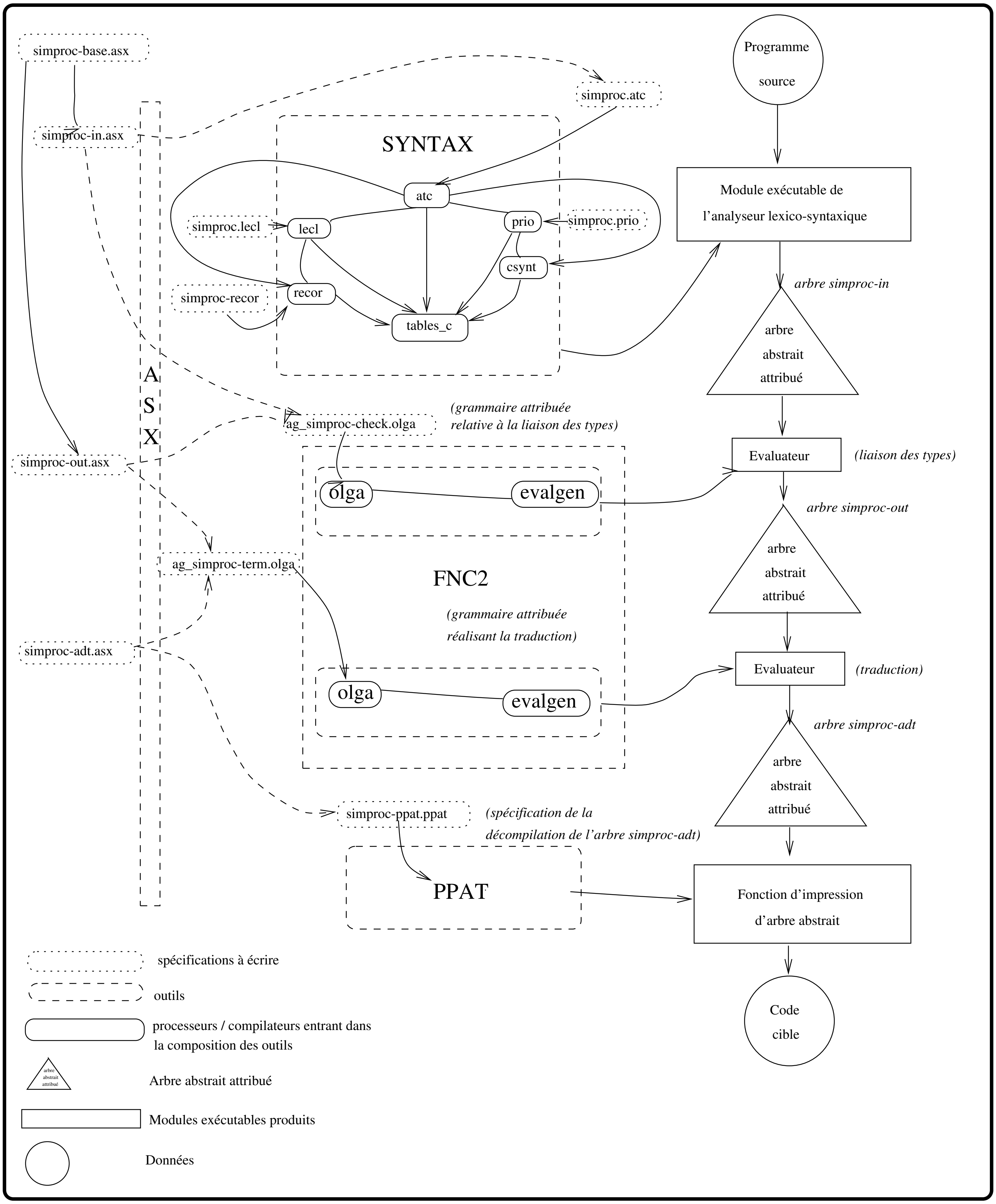

Figure 3.3: Application simproc

Opérateur : un opérateur est défini au sein d'un phylum et ne peut appartenir qu'à un seul phylum donné. C'est une fonction de construction d'arbre abstrait. Un non-terminal pouvant être associé à plusieurs règles de production, et donc donner lieu à la création de plusieurs arbres abstraits, on aura un opérateur par production.

Les opérateurs 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*.

Les deux productions suivantes (« `P1` » et « `P2` ») illustrent les notions de phylum et d'opérateur :

```
P1 : A -> B A
P2 : A -> C
```

On peut définir trois phylum, à partir de ces deux règles : « `ph-a` », « `ph-b` » et « `ph-c` », respectivement associés aux non-terminaux « `A` », « `B` » et « `C` ».

Le phylum « `ph-a` » comporte une notion de polymorphisme sur les arbres, car il représente un arbre pouvant prendre deux formes : la première forme étant un arbre de type « `ph-a` » avec un fils gauche de type « `ph-b` » et un fils droit de type « `ph-a` », la deuxième étant un arbre de type « `ph-a` » avec un fils unique de type « `ph-c` ».

On voit donc que le phylum « `ph-a` », doit posséder deux constructeurs distincts qui sont représentés par deux opérateurs que l'on pourra nommer « `op-p1` » et « `op-p2` ».

Caractéristiques des opérateurs `op-p1` et `op-p2`			
Nom d'opérateur	Arguments	Résultats	

`op-p1`	`b` de type `ph-b`
	`a` de type `ph-a`
		`s1` de type `ph-a`	`label (s1) = A`
			`prod (s1) = P1`
			`fils (s1) = {b, a}`

`op-p2`	`c` de type `ph-c`	`s2` de type `ph-a`	`label (s2) = A`
			`prod (s2) = P2`
			`fils (s2) = {c}`

On voit qu'il est possible de travailler sur des structures abstraites aussi bien que sur des arbres concrets générés à partir de programmes source : c'est la raison pour laquelle dans FNC-2, on préfère manipuler les notions de phyla et d'opérateurs plutôt que celles de non-terminaux et de productions.

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.

Dans le langage Asx de Fnc-22, l'exemple précédent s'écrit :

```
ph-a = op-p1
       op-p2 ;

op-p1 -> ph-b ph-a ;

op-p2 -> ph-c ;
```

Notion de syntaxe abstraite attribuée

Une *syntaxe abstraite attribuée* est une syntaxe abstraite dans laquelle on a défini des attributs sur un ou plusieurs phyla. Les syntaxes abstraites attribuées de Fnc-2 présentent plusieurs avantages :

- elles peuvent figurer dans des unités de compilation séparées, ce qui permet d'augmenter la modularité ;
- on peut définir des attributs typés sur chaque phylum ;
- 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.

Pour définir une syntaxe abstraite attribuée, il faut :

- définir une syntaxe abstraite ;
- déterminer les attributs en listant les phyla auxquels ils sont attachés ;
- déterminer le type des attributs.

Le langage Asx permet également de décrire les syntaxes abstraites attribuées.

Exemple L'exemple suivant est tiré de [Jou91]. Il décrit les syntaxes abstraites des arbres en entrée et en sortie du programme Ppat présenté dans la section 3.6.3.

Cet exemple permet de montrer comment il est possible de factoriser les descriptions d'arbres, grâce aux syntaxes abstraites. Effectivement, les arbres en entrée et en sortie du programme Ppat sont identiques dans leurs formes, seuls les attributs attachés aux phyla de l'arbre changent. Il est donc intéressant de ne décrire qu'une seule fois la structure des arbres et de spécifier à part la définition des attributs sur les arbres d'entrée et de sortie.

L'exemple est donc composé de trois modules :

- Le module ci-dessous (« `boxes-base` ») présente la syntaxe abstraite non attribuée décrivant la forme commune des arbres en entrée et en sortie du programme PPAT. La racine des arbres est représentée par le phylum « `ph-ppat` ». Le phylum « `ph-box` » décrit les diverses formes que peuvent prendre une boîte : soit la boîte est composée d'une liste de boîtes (opérateur « `op-box` »), soit elle est terminale (opérateur « `op-terminal` »), soit elle est vide (opérateur « `op-empty-box` »).

```
grammar boxes-base is    { Le nom de la grammaire est boxe-base }
    root is ph-ppat ; { ph-ppat est le phylum contenant l'opérateur
                        représentant la racine de l'arbre }
    ph-ppat    =  op-ppat ;
        op-ppat   -> ph-box ;  { op-ppat est un opérateur d'arité fixe }

    ph-box     =  op-box
                  op-terminal
                  op-empty-box ;

        op-box   -> ph-boxes ;
        op-terminal  -> ph-string-box ;
        op-empty-box -> ; { op-empty-box est un opérateur d'arité nulle }

    ph-boxes   = op-boxes;
        op-boxes     -> ph-box * ; { op-boxes est un opérateur homogène
                                     d'arité variable }

    ph-string-box = op-string-box ;
        op-string-box -> ;
end grammar ;
```

- Le module ci-dessous (« `boxes-in` ») importe les descriptions du premier module et ajoute la déclaration des attributs de l'arbre en entrée.

```
 grammar boxes-in is
    { inclusion d'un module de déclarations de types }
     from ppat-predef import all ;
    { importation des phyla et des opérateurs définis dans
      la syntaxe de base }
    import grammar boxes-base ;

    root is ph-ppat ;
```

```
        { $separator est un attribut du phylum ph-box, il est de type
          sep-info }
        $SEPARATOR  (ph-box) : sep-info ;
        $STRING-BOX (ph-string-box) : string ;
end grammar ;
```

- Le module ci-dessous (« `boxes-out` ») importe les descriptions du deuxième module et ajoute la déclaration d'un nouvel attribut sur l'arbre de sortie.

```
grammar boxes-out is
    { Importation des phyla et des opérateurs définis dans la syntaxe
      attribuée abstraite de l'arbre d'entrée }
    import grammar boxes-in ;

    root is ph-ppat ;
    { ajout de l'attribut représentant les coordonnées des boîtes
      $COORD est un attribut des phyla ph-ppat et ph-box }
    $COORD (ph-ppat, ph-box) : coord-box ;
end grammar ;
```

Le compilateur Asx

Le compilateur Asx, dont les entrées sont des syntaxes abstraites contenues dans des fichiers « `.asx` », remplit les fonctions suivantes :

- il effectue une analyse lexicale et syntaxique des syntaxes abstraites ;
- il vérifie que les phyla et opérateurs utilisés sont définis et il détecte les doubles déclarations ;
- il fournit en sortie un ensemble de tables contenues dans un fichier « `.ast` » ;
- 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 ;

3.8.2 Description et analyse de la lexicographie et de la syntaxe concrète

Fnc-2 utilise une version particulière de l'outil Syntax [BD88, JP94]. Dans les explications qui suivent, on substitue le processeur Bnf figurant dans la version standard de Syntax, par le processeur Atc livré avec Fnc-2.

L'outil Syntax utilisé avec Fnc-2 comprend sept parties : Atc, Lecl, Csynt, Prio, Recor, Tables_C et Paradis. Chacune de ces parties est détaillée dans les sections suivantes.

Atc

Atc est le constructeur d'arbres abstraits. Ce processeur traite des fichiers de type « `.atc` ». Ces fichiers contiennent à la fois la grammaire Bnf du langage à analyser et les appels de constructeurs d'arbre. Atc permet la construction du premier arbre abstrait qui viendra en entrée de l'outil Fnc-2.

Plus précisément, le processeur Atc a 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 Bnf.
- Il vérifie que chaque terminal peut être atteint 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 diverses erreurs, notamment les productions identiques, les terminaux qui se dérivent eux-mêmes par l'intermédiaire d'une chaîne de productions, l'utilisation de l'axiome en partie droite d'une production...

  Atc 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.
- Il construit des tables de transitions 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.
- Il permet la construction d'arbre décrits par une syntaxe abstraite contenue dans un fichier de type « `.asx` ».

Lecl

Lecl est le constructeur lexical de Syntax. Il offre les caractéristiques suivantes :

- Il prend en entrée un fichier « `.lecl` » contenant la spécification des unités lexicales du langage à analyser. Il lit également les tables internes produites par Atc, notamment celle donnant la liste des terminaux pouvant apparaître après 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 expressions régulières et de les réutiliser dans la construction d'autres expressions. Il offre également la possibilité de créer des synonymes pour des mots clés, 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 être spécifiés par des expressions régulières, par exemple la transformation d'un identificateur en majuscules. LECL permet aussi de définir des prédicats qui permettent à l'utilisateur de vérifier que certains caractères apparaissent dans une colonne donnée, ce qui peut être utile pour des langages tels que FORTRAN ou COBOL.

CSYNT

CSYNT est le constructeur syntaxique. Il travaille à partir des tables internes générées par ATC ou BNF et 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, 4 possibilités sont offertes :

- 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.

PRIO

PRIO est un résolveur d'ambiguïtés. Il prend en entrée un fichier « `.prio` » définissant des priorités sur les opérateurs ou sur les règles de grammaire. La syntaxe et la sémantique des spécifications sont proches de celles de YACC. Autant que possible, nous essayerons d'écrire des grammaires non ambiguës afin d'éviter l'écriture de ces règles de priorité.

RECOR

RECOR est le processeur gérant le mécanisme de rattrapage sur erreur. Le traitement des erreurs lexicales et syntaxiques 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 un fichier « `.recor` » écrit 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.

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 une bibliothèque fournie avec Syntax pour former un analyseur lexico-syntaxique complet.

Paradis

Paradis est un outil permettant de produire des paragrapheurs dirigés par la syntaxe. Comme pour la production d'analyseurs lexico-syntaxiques, on décrit par une grammaire Bnf le langage des programmes à paragrapher. Cette grammaire Bnf est ensuite paragraphée manuellement, selon les préférences de l'utilisateur. Ce paragraphage manuel fournit un modèle pour la production automatique d'un paragrapheur adapté au langage considéré.

3.8.3 Description et analyse de la sémantique statique

Fnc-2 repose sur le concept de fonctions prenant en entrée un arbre attribué et délivrant en sortie un autre[15] arbre attribué, éventuellement décoré différemment (c'est à dire enrichi par de nouveaux attributs). On parle alors de *transformation* d'arbres attribués.

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 (ou phases), chacune prenant en entrée l'arbre attribué fourni par la précédente. 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 cet arbre. Fnc-2 peut ainsi fonctionner avec les outils classiques comme Lex et Yacc. Toutefois nous avons préféré utiliser l'outil Syntax pour ses qualités sus-mentionnées.

Précisions sur les grammaires attribuées utilisées dans Fnc-2

Nous avons déjà présenté les grammaires attribuées en 3.1.2, mais nous devons apporter quelques précisions sur les types de grammaires et les catégories d'attributs supportées par le système Fnc-2.

Pour le système Fnc-2, il existe deux types de grammaires attribuées :

[15] En fait la transformation peut déboucher sur zéro, un ou plusieurs arbres attribués, mais dans cette étude nous ne produirons qu'un seul arbre en sortie.

- 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.

Le langage OLGA pour le calcul des attributs

OLGA est un langage spécialisé dans le traitement des grammaires attribuées. Les points forts de ce langage sont la *facilité* d'utilisation, la *puissance d'expression*, l'*efficacité*, la *modularité* et la *lisibilité*. En fait, OLGA est un langage à vocation générale que l'on pourrait envisager d'utiliser dans un contexte différent de celui de la compilation. Les programmes OLGA sont écrits dans des fichiers « `.olga` ». Les caractéristiques essentielles du langage OLGA sont les suivantes :

- C'est un langage *applicatif*, c'est-à dire un langage pour lequel il n'y a ni affectation ni effet de bord[16], mais seulement des expressions et des fonctions ;

[16]Dans certains cas, il peut être intéressant, voire nécessaire d'appeler des fonctions externes, cela est possible avec OLGA et le programmeur pourra contourner ce principe...

- 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 complexe permettant de décrire 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 ». Il existe également des types plus complexes : *énumération*, *ensemble*, *sous-ensemble*, *enregistrement*, *union* et *liste*.

- OLGA permet la définition de fonctions supportant le *polymorphisme* (les fonctions acceptent des paramètres de différents types) ainsi que la surcharge d'opérateurs. Les définitions de fonctions appartiennent à des blocs pouvant être imbriqués. Comme en ADA et C++, lorsque les fonctions sont déclarées **inline**, le compilateur génère le code de la fonction pour chaque occurence d'utilisation, plutôt que d'effectuer un appel (optimisation du temps d'exécution). En utilisant des directives de compilation (*pragma*), on peut invalider cette indication.

- 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 clauses d'importation de grammaires, ce qui est particulièrement intéressant dans le cas des *grammaires attribuées procédurales* (cf. définition page 33), 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.

- Signaler à l'utilisateur la présence d'erreurs dans le programme source est un rôle important du compilateur [ASU91]. Le langage OLGA comporte à cet effet, toutes les facilités de traitement nécessaires. Il est capable de renseigner l'utilisateur 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 qu'un 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 : $+, -, >=$;
  - 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 ... match** permet de réaliser des tests de concordance de motifs (*pattern matching*). Cette facilité permet de vérifier qu'une valeur correspond à une certaine structure.

- Olga étant spécialisé dans le traitement des grammaires attribuées, il permet de spécifier les règles sémantiques associées aux productions, c'est-à dire de spécifier les algorithmes de calculs d'attributs. Il est *indépendant* des méthodes d'évaluation d'attributs.

  Chaque règle sémantique est associée à un nœud de l'arbre abstrait, donc à un opérateur (voir la notion de syntaxe abstraite en 3.8.1) et, par conséquent, à une production de la grammaire Bnf.

  Une règle sémantique 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 occurrence d'attribut qui peut se faire de deux façons :

  - soit explicitement, en écrivant les instructions adéquates (voir l'exemple ci-dessous).

```
attribute
    {On déclare que les phyla ph-program et ph-block, possèdent un
     attribut dont le nom est $CORRECT}
    synthesized $CORRECT (ph-program, ph-block) ;

    where op-program -> ph-block
    use
        {op-program est un opérateur du phylum ph-program (qu'on ne
         voit pas dans cet exemple). Dans la partie gauche de
         l'affectation, on n'a pas précisé le propriétaire de
         l'attribut $CORRECT, mais le compilateur FNC-2 en déduira
         qu'il appartient au propriétaire de l'opérateur op-program,
         c'est-à-dire au phylum ph-program.

         Cette affectation recopie l'attribut $CORRECT du phylum
         ph-block dans l'attribut $CORRECT du phylum ph-program.}

         $CORRECT := $CORRECT (ph-block) ;
    end where
```

  - soit implicitement, par l'utilisation de *règles de copies par défaut.* Il s'agit d'une facilité introduites par Bernard Lorho [Lor77] et permettant d'omettre l'écriture de certaines règles sémantiques.

Ces règles entrent en action lorsque le compilateur FNC-2 s'aperçoit, après avoir évalué toutes les règles sémantiques attachées à une production, qu'il reste des attributs de sortie à évaluer.

Sur la figure 3.4, nous 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).

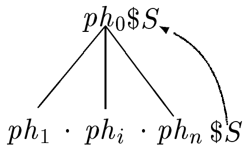

Figure 3.4: Exemple de règle de copie par défaut

Dans l'exemple précédent, nous aurions pu omettre l'écriture de l'affectation « `$CORRECT := $CORRECT (ph-block) ;` » s'agissant d'un attribut synthétisé, FNC-2 aurait pris la décision de recopier l'attribut de même nom du phylum le plus à droite dans la règle de production.

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 de ce style.

- OLGA fournit des attributs *prédéfinis*, 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 permet de repérer le texte source de la notion syntaxique associée au non-terminal représenté par un phylum donné. Cette dernière possibilité est vraiment très intéressante pour l'écrivain d'un compilateur, car elle le décharge totalement de la gestion des numéros de lignes, colonnes et noms de modules qu'il faudrait, sinon, prendre en compte pour la génération de messages d'erreurs significatifs.

- 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.

- OLGA fournit un mécanisme de construction d'arbres basé sur les syntaxes abstraites présentées en 3.8.1. Un opérateur d'arité fixe appartenant au phylum `ph` et déclaré ainsi :

$$\mathtt{op} \;\rightarrow\; \mathtt{ph}_1\;\mathtt{ph}_2 \cdots \mathtt{ph}_n$$

introduit la fonction de construction **function** `op (ph`$_1$ `; ph`$_2$ `;`$\cdots$ `; ph`$_n$`) : ph`.
Un opérateur liste appartenant au phylum `ph` et déclaré ainsi :

`op` $\rightarrow$ `ph`$_1$`*` ou `op` $\rightarrow$ `ph`$_1$`+`

introduit les fonctions de construction suivantes :

- **function** `op () : ph`
  constuit une liste vide de type `ph`
- **function** `op (ph`$_1$`) : ph`
  construit une liste d'un seul élément de type `ph`$_1$
- **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 ph`$_1$`) : 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.

Lorsque l'on veut décorer l'arbre abstrait, c'est-à dire attacher des attributs aux nœuds, on le fait en utilisant l'expression **with... end with**.

Exemple Dans l'exemple ci-dessous, on cherche à traduire le plus haut niveau syntaxique d'un programme source, dans le plus haut niveau syntaxique d'un programme cible et à donner un nom au programme cible.

```
attribute
{ Ici, on définit un attribut de type arbre ph-program-cible. Ce phylum
  ph-program-cible est définit dans une syntaxe abstraite que l'on ne
  voit pas ici mais qui servira à construire un nouvel arbre abstrait.
  Nous sommes donc dans le cas d'une grammaire attribuée procédurale }
  synthesized $PROGRAM (ph-program) : ph-program-cible ;

{ BLOCK-MODEL servira à la construction du sous-arbre représentant
  un bloc du programme cible }
  synthesized $BLOCK-MODEL (ph-block) : ph-block-cible ;

where op-program -> ph-block
use
        $PROGRAM := op-program-cible (op-program-id ()
```

```
                            with
                                $ID := token ("NomDeProgramme")
                            end with,
                            $BLOCK-MODEL (ph-block)) ;
        end where
```

Fonctionnement de Fnc-2

L'architecture du compilateur Fnc-2 est donnée sur la figure 3.5. Son rôle 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.

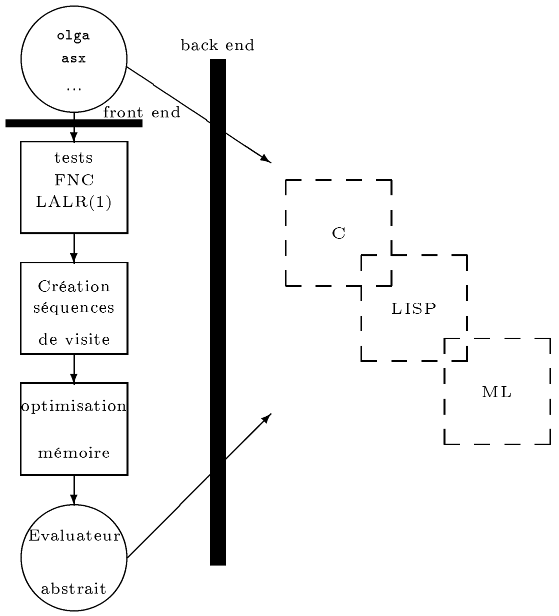

Figure 3.5: Architecture du compilateur Fnc-2

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

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 évaluateurs abstraits qui sont ensuite traduits dans l'un des trois langages–cible actuellement supportés par Fnc-2 : C, Lisp et ML.

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 (compilation) 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*, [CH94, Fan72, KW76, 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 de 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.

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].

Chapitre 4

Réalisation

Le but de ce chapitre est de donner une vision du travail réalisé. Plus technique que les précédents, il donne des détails sur la méthode employée pour le développement du compilateur TRAIAN pour E-LOTOS. La section 4.6 présente aussi les paragrapheurs pour LOTOS et E-LOTOS que nous avons réalisés.

Pour qu'un lecteur intéressé par le système SYNTAX/FNC-2 puisse tirer profit de la lecture de ce chapitre, nous avons eu le souci d'utiliser un exemple unique comme « fil conducteur » : le renommage de types en E-LOTOS.

4.1 Architecture du compilateur E-Lotos

Sur la figure 4.1, on présente la structure du compilateur E-LOTOS. Cette structure suit celle du compilateur SIMPROC (voir la figure 3.3). Le schéma de la page 42, représente les 3 trois arbres abstraits utilisés dans notre prototype de compilateur :

- L'arbre abstrait « `elotos-in` » est le premier figurant en entrée de FNC-2. Cet arbre est produit par l'analyseur lexico-syntaxique généré par l'outil SYNTAX et représente une modélisation du langage E-LOTOS.

- L'arbre abstrait « `elotos_typage` » est construit grâce à la grammaire attribuée procédurale « `ag_elotos_typage.olga` ». Cet arbre, qui constitue le résultat de l'analyse sémantique, est une « redécoration » de l'arbre « `elotos-in` ».

- L'arbre abstrait « `elotos-lotos` » est construit grâce à la grammaire attribuée fonctionnelle « `ag_elotos-lotos.olga` ». Cet arbre, qui constitue l'entrée de la phase de traduction, sert à la production du code cible LOTOS. C'est une modélisation du langage LOTOS.

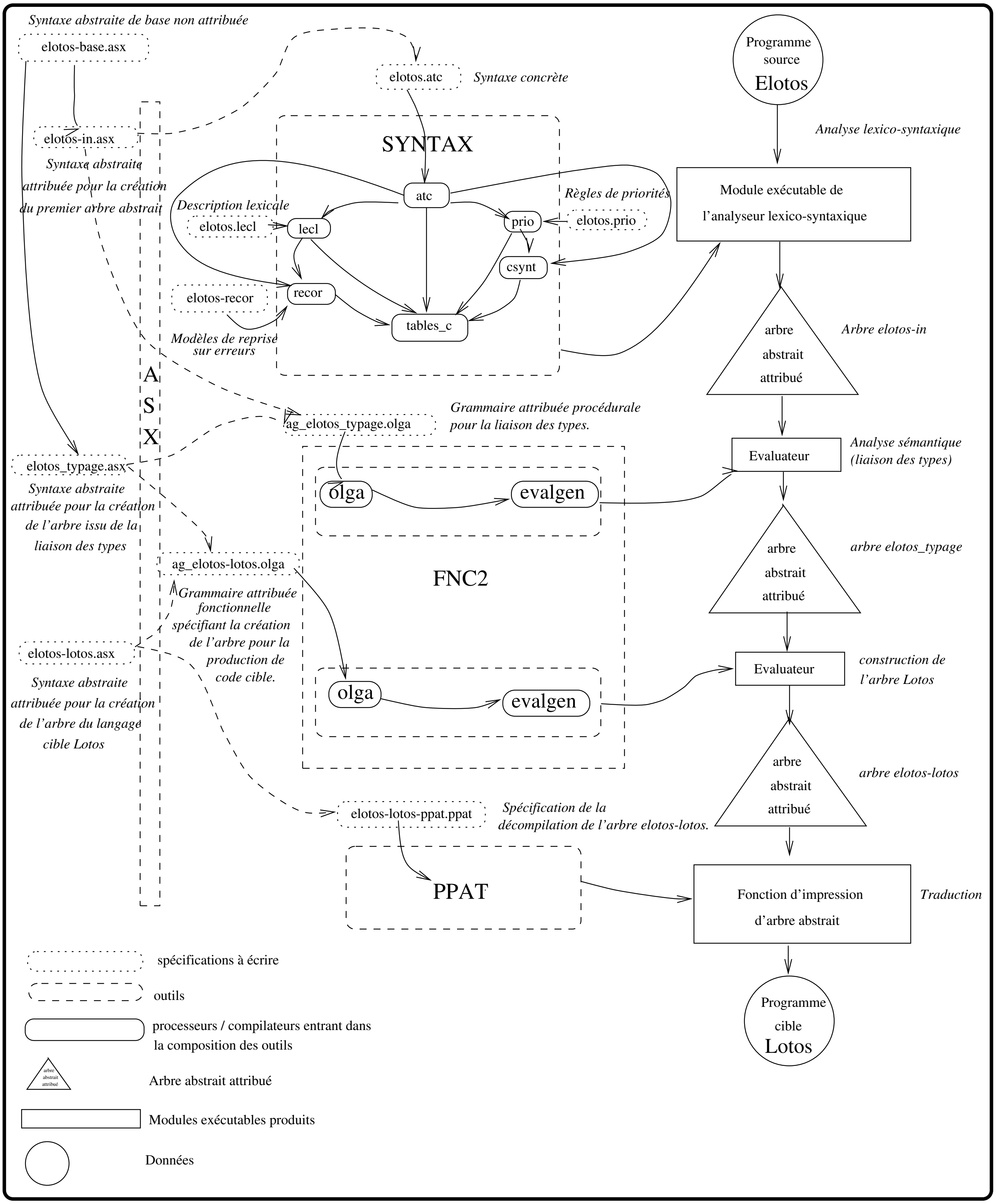

Figure 4.1: Architecture du compilateur E-Lotos

4.2 Syntaxes abstraites

Pour décrire notre compilateur, nous avons besoin d'une syntaxe abstraite non attribuée (« `elotos-base.asx` ») et de trois syntaxes abstraites attribuées (« `elotos-in.asx` », « `elotos_typage.asx` » et « `elotos-lotos.asx` »). Dans les sections suivantes, nous présentons successivement ces quatre syntaxes abstraites.

4.2.1 Syntaxe abstraite pour le langage source E-Lotos

Ci-dessous, nous présentons un extrait de la syntaxe abstraite « `elotos-base.asx` » modélisant le langage E-Lotos. Cette syntaxe abstraite est également représentée sur le schéma 4.2 page 44.

```
grammar elotos-base is

root is ph-elotos-axiom ;

ph-elotos-axiom = op-elotos-axiom ;
    op-elotos-axiom -> ph-specification ;

ph-specification  = op-specification ;
    op-specification -> ph-identifier
                        ph-definitions
                        ph-behaviour ;

ph-definitions   = op-definitions ;
    op-definitions  -> ph-definition * ;

ph-definition    = op-type-synonym-definition
                   op-type-definition
                   op-type-external-definition
                   op-process-definition
                   op-function-definition ;

    op-type-synonym-definition -> ph-identifier
                                  ph-identifier ;
...
```

La racine de l'arbre abstrait est représentée par le phylum « `ph-elotos-axiom` » auquel appartient l'opérateur « `op-elotos-axiom` ». Ce dernier modélise une spécification[17]. L'opérateur « `op-specification` » décrit les différentes parties d'une spécification E-Lotos :

[17] E-Lotos étant un langage pour la description formelle des protocoles de communication et des systèmes distribués, les sources ne sont pas des programmes, mais des spécifications.

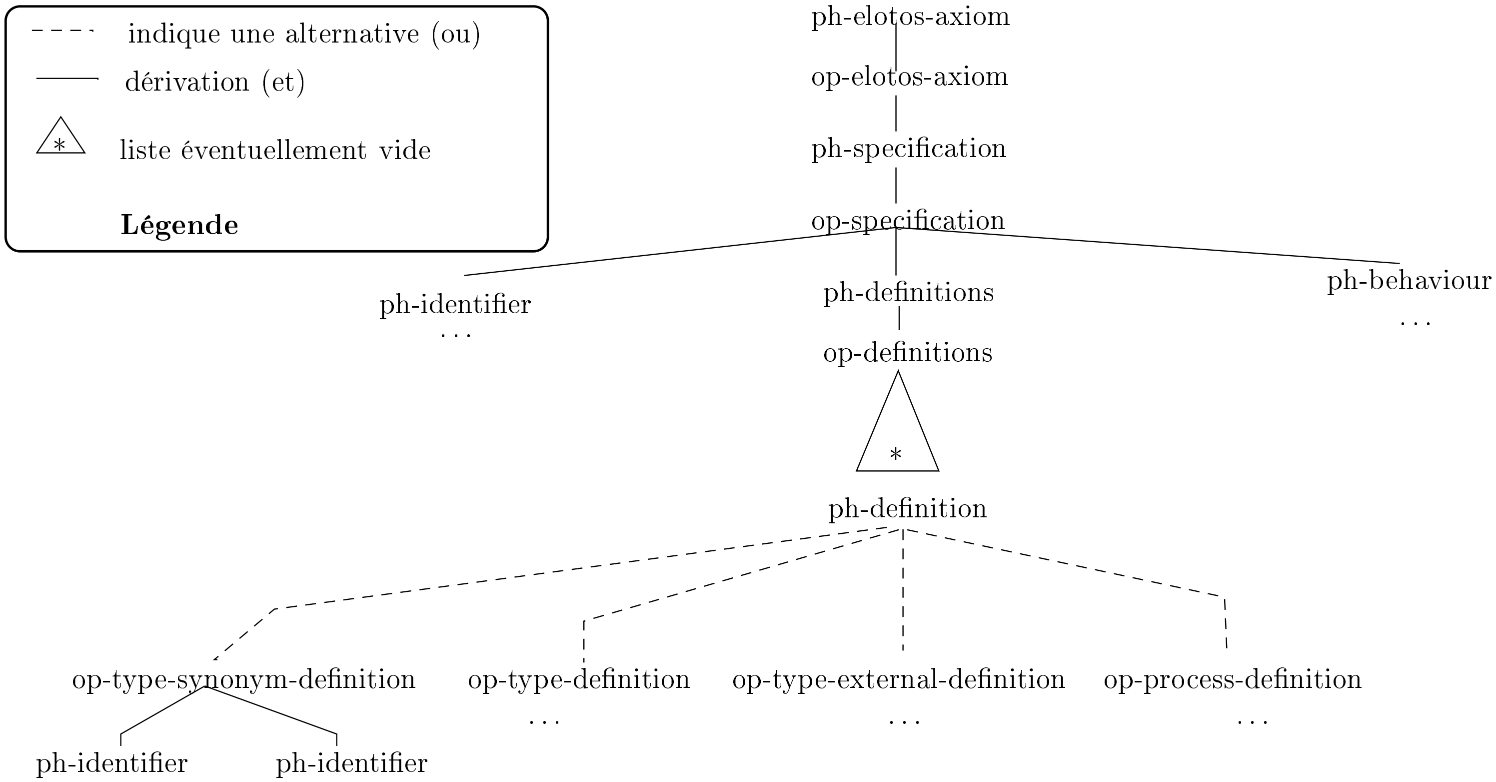

Figure 4.2: Une partie de l'arbre abstrait E-LOTOS

- une partie « nom de spécification » représentée par le phylum « `ph-identifier` » ;
- une partie « definitions » représentée par le phylum « `ph-definitions` » ;
- une partie « comportement » représentée par le phylum « `ph-behaviour` » ;

L'opérateur « `op-definitions` » appartenant au phylum « `ph-definitions` », modélise une liste de définitions.

Les opérateurs appartenant au phylum « `ph-definition` » modélisent les définitions que l'on peut trouver en E-LOTOS. Seul l'opérateur « `op-type-synonym-definition` » correspondant à l'exemple du renommage de types est détaillé. Comme on peut le voir sur le schéma de traduction donné en 4.5.1 page 60, le renommage de type E-LOTOS nécessite deux identificateurs (S et S' dans le schéma de traduction).

4.2.2 Syntaxe abstraite attribuée pour l'analyse syntaxique

La syntaxe abstraite attribuée « `elotos-in.asx` » permet de déclarer les attributs nécessaires à l'analyse syntaxique. Comme on peut le voir ci-dessous, cette syntaxe importe la syntaxe non attribuée « `elotos-base.asx` » décrite dans la section précédente.

```
grammar elotos-in is

    import grammar elotos-base ;
    root is ph-elotos-axiom ;

    $identifier (ph-identifier) : token ;
    $integer (ph-integer) : token ;
    $real (ph-real) : token ;
    $special-comment (ph-special-comment) : token ;
end grammar ; {elotos}
```

Dans cet exemple, on déclare 4 attributs de type « `token` » (« `$identifier` », « `$integer` », « `$real` » et « `$special-comment` »), ils seront valorisés au moment de l'analyse lexico-syntaxique.

4.2.3 Syntaxe abstraite attribuée pour l'analyse sémantique

La syntaxe abstraite attribuée « `elotos_typage.asx` » permet de déclarer les attributs nécessaires à l'analyse sémantique. Comme la syntaxe présentée ci-avant, elle importe la syntaxe non attribuée « `elotos-base.asx` » et ne contient que des déclarations d'attributs, aussi nous ne la représenterons pas.

4.2.4 Syntaxe abstraite attribuée pour la traduction vers LOTOS

Pour effectuer la traduction d'un langage source vers un langage cible, on commence par spécifier la forme attendue pour le langage cible. Grâce aux schémas de traduction présentés en section B.1 page 129, nous sommes en mesure de décrire l'arbre abstrait à produire. La syntaxe abstraite de cet arbre est détaillée ci-dessous et la représentation graphique correspondante est donnée page 47.

Pour le moment, nous ne nous préoccupons pas des attributs qu'il faudra inévitablement attacher aux phyla, nous devons juste identifier les informations dont nous aurons besoin dans l'arbre construit, au moment où PPAT devra produire le code cible.

Dans le schéma de traduction donné en B.1 page 129, nous pouvons faire abstraction des mots-clés qui seront générés au moment de la décompilation de l'arbre ; les informations à prendre en compte dans l'arbre abstrait à produire sont donc :

- un identificateur donnant le nom de la spécification en sortie ;
- une partie contenant les informations de déclarations de types LOTOS ;
- une partie décrivant le comportement de la spécification LOTOS ;
- une partie de déclaration de processus que nous n'utiliserons pas dans le cas de l'exemple de renommage de type.

On retrouve très clairement ces informations dans le début de la syntaxe abstraite ci-dessous :

```
grammar elotos-lotos is

from elotos-lotos-types import  all ;
root is ph-lot-axiom ;

ph-lot-axiom                = op-lot-axiom ;
    op-lot-axiom           -> ph-lot-specification-identifier
                              ph-lot-declarations
                              ph-lot-behaviour-expression
                              ph-lot-where-declarations ;
```

Une déclaration LOTOS comporte trois parties. En fonction du schéma de traduction donné en 4.5.1 page 59, nous décrivons le sous-arbre « `ph-lot-declarations` » :

```
ph-lot-declarations         = op-lot-declarations ;
    op-lot-declarations    -> ph-lot-sorts-declarations
                              ph-lot-opns-declarations
                              ph-lot-forall-eqns-declarations ;
```

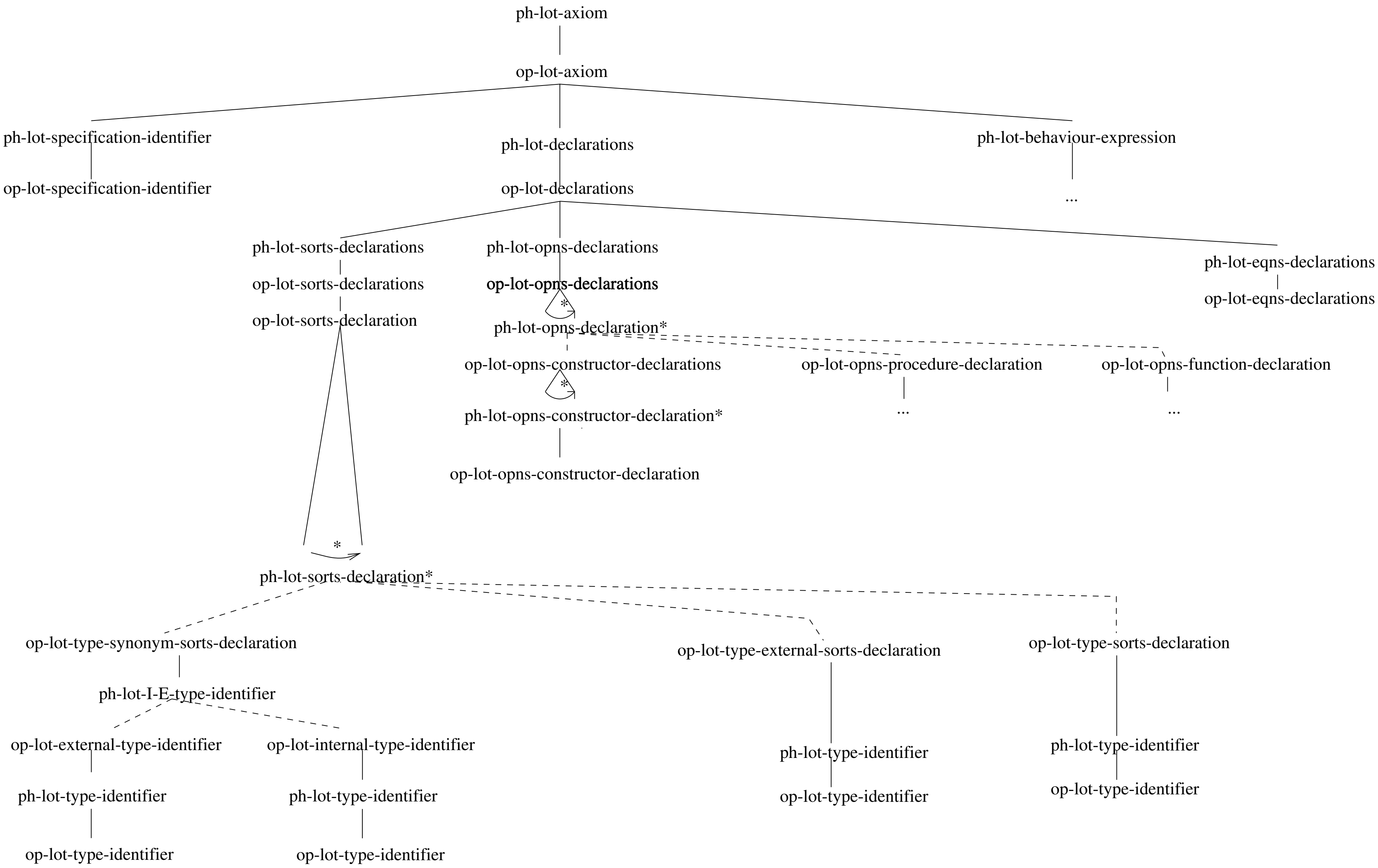

Figure 4.3: Arbre à construire

Pour la suite de la syntaxe abstraite, nous nous reporterons aux schémas de traduction donnés en 4.5.1 page 60. Cette syntaxe abstraite a été débarrassée des opérateurs et phyla qui n'étaient pas nécessaires à la compréhension de notre exemple.

```
ph-lot-sorts-declarations   = op-lot-sorts-declarations ;
    op-lot-sorts-declarations
                           -> ph-lot-sorts-declaration* ;

ph-lot-sorts-declaration    = op-lot-type-synonym-sorts-declaration ;
    op-lot-type-synonym-sorts-declaration
                           -> ph-lot-internal-external-type-identifier ;

ph-lot-internal-external-type-identifier
                            = op-lot-external-type-identifier
                              op-lot-internal-type-identifier ;

    op-lot-internal-type-identifier
                           -> ph-lot-type-identifier ;
    op-lot-external-type-identifier
                           -> ph-lot-type-identifier
                              ph-lot-type-identifier ;
```

On peut avoir plusieurs déclarations de sortes à produire et nous avons une traduction différente selon que la sorte renommée est de type « external » ou non : c'est pourquoi l'on trouve les deux opérateurs « `op-lot-external-type-identifier` » et « `op-lot-internal-type-identifier` ».

La fin de la syntaxe abstraite « `elotos-lotos` » regroupe des déclarations d'attributs. Nous donnons ci-dessous, la déclaration de l'attribut « `$LOT-IDENTIFIER` » de type « `token` ». On l'utilise dans la grammaire attribuée fonctionnelle présentée en section 4.5.2, à chaque fois que l'on a besoin d'affecter un nom significatif à un élément de la traduction. On voit que l'attribut « `$LOT-IDENTIFIER` » est déclaré sur plusieurs phyla. Par exemple, l'attribut attaché au phylum « `ph-lot-specification-identifier` » nous servira à générer le nom de la spécification LOTOS.

```
$LOT-IDENTIFIER (ph-lot-specification-identifier,
                 ph-lot-type-identifier,
                 ph-lot-declarations,
                 ph-lot-identifier) : token ;
```

4.2.5 Fichiers produits

Les spécifications écrites		
Nom du fichier	*Nb de lignes*	*Description*
`elotos-base.asx`	993	Description de la syntaxe abstraite de base, non attribuée.
`elotos-in.asx`	30	Description de la syntaxe abstraite attribuée (cette syntaxe importe elotos-base.asx).
`elotos_typage.asx`	132	Syntaxe abstraite attribuée pour l'arbre issu de la phase de typage.
`elotos-lotos.asx`	993	Syntaxe abstraite de l'arbre pour la production de code cible

4.3 Description lexicographique du langage E-LOTOS

4.3.1 Description lexicale

Pour obtenir un analyseur lexical adapté au langage E-LOTOS, nous avons écrit des spécifications dans le méta-langage lexical LECL. Le fichier « `elotos.lecl` » comprend 160 lignes. Il s'inspire du fichier de description lexicale du compilateur CÆSAR.

4.3.2 Description syntaxique et construction de l'arbre abstrait

Afin d'obtenir un analyseur syntaxique pour le langage E-LOTOS nous avons décrit la grammaire concrète avec une notation proche de la BNF (fichier « `elotos.atc` ») et nous avons ajouté les appels aux constructeurs d'arbres après chaque production.

Dans l'exemple présenté ci-dessous, les non-terminaux définis en partie gauche des règles de production sont séparés de la partie droite par le caractère « `=` ». Les non–terminaux sont encadrés par des chevrons (« `< >` »). Les terminaux génériques – c'est à dire ceux qui ne sont ni des mots–clés du langage, ni des suites de caractères spéciaux – sont précédés du caractère « `%` ». Les autres terminaux sont encadrés par des guillemets (« `" "` »). Les productions, écrites dans une forme proche de la syntaxe BNF, sont terminées par un point-virgule. La ligne suivant immédiatement une production, peut être vierge ou contenir un appel à un constructeur d'arbre.

Le texte ci-dessous, extrait du fichier `elotos.atc`, définit la syntaxe concrète du langage E-LOTOS avec l'appel des constructeurs d'arbres.

```
import grammar elotos-in ;

<elotos-axiom> = <specification> ;
    op-elotos-axiom (<specification>)
```

```
<specification> = "SPECIFICATION"  <specification-identifier>  "IS"
                     <definitions>
                  "BEHAVIOUR"
                     <behaviour>
                  "ENDSPEC" ;
                  op-specification (<specification-identifier>,
                                    <definitions>,
                                    <behaviour>)

<definitions> = <definitions>
                <definition> ;
                op-definitions-post

<definitions> = ;
                op-definitions

<definition> = <type-definition> ;

<definition> = <function-definition> ;

<definition> = <process-definition> ;

...
<type-definition> = "TYPE" <type-identifier> "RENAMES"
                      <type-identifier>
                    "ENDTYPE" ;
                     op-type-synonym-definition

<specification-identifier> =  <identifier> ;

...

<identifier> = %IDENTIFIER ;
               op-identifier () with
                                 $identifier := %IDENTIFIER
                                end with
```

Nous commentons tour à tour les différentes règles de cette grammaire :

- La première production de notre exemple est l'axiome de la grammaire. Quand l'analyseur syntaxique réduit cette production, le sommet de notre arbre abstrait est construit en appelant l'opérateur « `op-elotos-axiom` » associé au phylum

« `ph-elotos-axiom` » (voir les syntaxes abstraites présentées dans les sections 4.2.1 et 4.2.2).

On voit que le non-terminal « `<specification>` » est passé comme opérande au constructeur « `op-elotos-axiom` ». Ceci signifie que les productions associées à ce non-terminal devront construire des arbres du type « `ph-specification` » (fils unique de « `op-elotos-axiom` »).

Ceci nous permet de souligner un intérêt majeur de SYNTAX/FNC-2 : le fichier « `elotos.atc` » qui contient la syntaxe BNF est analysé conjointement aux tables systèmes issues de la compilation du module « `elotos-in.asx` ». Ceci signifie que des inconsistances peuvent être détectées par le système. Par exemple la construction d'un arbre qui n'est pas du bon type (c'est à dire qui ne correspond pas aux phyla dans lesquels dérive l'opérateur), peut être détectée au moment de l'analyse du fichier « `elotos.atc` ».

- La deuxième production nous indique qu'une spécification E-LOTOS commence par le mot clé « `SPECIFICATION` », suivi d'un nom de spécification, suivi par le mot clé « `IS` » etc.

  L'appel au constructeur « `op-specification` » nous permet de constater qu'aucun des mots clés ne figure dans les paramètres du constructeur : nous ne les retrouverons donc pas dans notre arbre abstrait. Les trois paramètres passés correspondent à la définition de l'opérateur « `op-specification` » de notre syntaxe abstraite.

  Les concepteurs de SYNTAX/FNC-2 ont remarqué que les opérandes des constructeurs étaient souvent les non-terminaux de la partie droite des règles syntaxiques, et dans le même ordre. Ils se sont appuyés sur cette observation pour nous donner la possibilité d'omettre l'écriture des opérandes. Ainsi, nous pouvons écrire :

```
<specification> = "SPECIFICATION"  <specification-identifier>  "IS"
                      <definitions>
                  "BEHAVIOUR"
                      <behaviour>
                  "ENDSPEC" ;
                  op-specification
```

- La troisième production concerne les listes de definitions du langage E-LOTOS. Dans la syntaxe abstraite correspondante (voir les sections 4.2.1 et 4.2.2), le lecteur pourra vérifier que le phylum « `ph-definitions` » possède l'opérateur homogène d'arité variable « `op-definitions` ». A cet opérateur correspondent plusieurs constructeurs et, en particulier, le constructeur « `op-definitions-post` » qui ajoute un élément de type « `ph-definition` » à la fin d'une liste de type « `ph-definitions` »[18]. Ces opérateurs sont définis automatiquement à partir de la syntaxe abstraite.

[18] Le pluriel indique une liste

4.3.3 Résolution des ambiguïtés

Nous avons utilisé l'outil ATC, qui nous a signalé quelques ambiguïtés dans la grammaire concrète du langage E-LOTOS. Celle-ci a été corrigée par une réécriture des constructions ambiguës. Nous n'avons donc pas eu à écrire de fichier « .prio ».

4.3.4 Modèles pour la reprise sur erreur

La description des modèles pour la reprise sur erreur est celle fournie par le système SYNTAX/FNC-2, à laquelle nous avons ajouté la description des mots clés du langage E-LOTOS.

4.3.5 Fichiers produits

Les spécifications écrites		
Nom du fichier	*Nb lignes*	*Description*
`elotos.atc`	2777	Description de la syntaxe concrète E-LOTOS avec une grammaire BNF + appel des constructeurs de l'arbre abstrait.
`elotos.recor`	82	Fichier contenant les modèles pour la reprise sur erreur.

4.4 Analyse sémantique

4.4.1 La gestion des messages d'erreur

Une des fonctions importantes d'un compilateur est la génération de messages d'erreur clairs, identifiant et localisant précisément le problème à l'intérieur du texte source.

Il est souhaitable de disposer d'une fonction capable de générer un libellé en fonction d'un type d'erreur et capable de donner des informations complémentaires, comme la phase de compilation ou des informations de mise au point masquables pour la version définitive du compilateur... Ceci est assez facile à réaliser, mais la localisation et l'affichage d'une portion de texte source à l'origine d'une erreur, est un problème beaucoup plus difficile à résoudre. En utilisant le langage OLGA, nous avons optimisé au mieux toutes ces tâches de développement.

OLGA gère un attribut « `$scx` » de type « `source-index` » pour chaque phylum de la syntaxe abstraite. Les valeurs de type « `source-index` » comprennent un nom de fichier, un numéro de ligne et un numéro de colonne. L'attribut « `$scx` » de chaque phylum est affecté pendant l'analyse lexico-syntaxique, au moment de la construction de l'arbre abstrait. Puisqu'un phylum est en réalité un représentant d'un non-terminal de la grammaire BNF, le langage OLGA a été doté d'une fonction « `message` » permettant d'afficher le nom du fichier, le numéro de ligne et le numéro de colonne correspondant au texte associé au phylum.

Toutefois, afin de simplifier davantage la gestion des messages d'erreur, nous avons écrit deux modules permettant une utilisation aisée de la fonction « `message` ». Le principe de gestion des messages est le suivant : lorsqu'une erreur est détectée au cours de l'exécution d'une règle sémantique, on appelle une fonction « `message-elotos` » à laquelle on fournit les opérandes suivants :

- un identificateur de message, qui est une valeur faisant partie d'un type énuméré. Cet identificateur est associé à :
  - un libellé ;
  - un nom de phase de compilation ;
  - un niveau de sévérité ;
  - un code retour de type booléen ;
- un booléen indiquant si l'on se trouve ou pas, en mode « mise au point » ;
- un texte affiché uniquement en cas de mode « mise au point » (« debug ») ;
- un attribut de type « `source-index` ».

Dans l'exemple suivant, l'attribut « `$CORRECT` », de type booléen, se voit attribuer une valeur par le biais de la fonction « `message-elotos` ».

```
$CORRECT := message-elotos (ELOTOS_UNDECLARED_ID,
                            $MODE_DEBUG [ph-specification],
                            "{Du1}",
                            $scx(ph-identifier2))
```

La fonction « `message-elotos` » renvoie « `false` » dans le cas d'une erreur fatale et « `true` » dans le cas d'un simple message d'avertissement.

La signification de chacun des paramètres de l'exemple précédent est donnée ci-dessous :

« `ELOTOS_UNDECLARED_ID` » : est l'identificateur de messages correspondant aux informations suivantes :
  - `"Identifier undeclared"` (libellé du message) ;
  - `"Error occured during checking phase"` (nom de phase de compilation) ;
  - `"2"` (code d'erreur sévère) ;
  - `"false"` (code de retour booléen) ;

« `$MODE_DEBUG [ph-specification]` » : est l'attribut de type booléen, indiquant si l'on se trouve en mode « mise au point » ou non (« `true` » ou « `false` »). Cet attribut est affecté dans les grammaires attribuées Olga utilisant la fonction

« `messages-elotos` ». Ce choix de fonctionnement nous donne la possibilité de définir un mode « mise au point » par grammaire attribuée, ce qui est très pratique lors de la mise au point du compilateur.

« `Du1` » **:** est un libellé identifiant une règle de sémantique statique dans le document [SG96]. Ce libellé sera affiché si le paramètre précédent est valorisé à « `true` » (mode «debug »).

« `$scx(ph-identifier2)` » **:** est un attribut de type « `source-index` ». Il contient les informations relatives à la position du texte source dérivant du non-terminal associé à « `ph-identifier2` ». Il permet à la fonction « `message` » d'afficher le nom du fichier source, le numéro de la ligne et le numéro de la colonne du texte source correspondant à l'erreur signalée.

4.4.2 Vérification des règles de sémantique statique

La phase de contrôle de la sémantique statique est une implémentation, sous forme de grammaires attribuées, des règles décrites dans le document « E-LOTOS User Langage » [SG96]. Ce document définit la sémantique statique sous forme de règles de Plotkin.

Dans la suite de ce paragraphe, nous allons donner un exemple de règle de Plotkin et la règle sémantique équivalente exprimée sous forme de grammaire attribuée. L'exemple que nous avons choisi concerne le renommage de types. La construction syntaxique pour le renommage de type en E-LOTOS, est :

$$\textbf{type } S \textbf{ renames } S' \textbf{ endtype}$$

La règle de Plotkin suivante spécifie la sémantique statique liée à cette construction.

$$\frac{C \ \vdash \ S' \rightarrow S''}{C \ \vdash \ (\textbf{type } S \textbf{ renames } S'\textbf{endtype}) \ \Rightarrow (S \ \mapsto \ S'')}$$

Elle signifie que dans l'hypothèse où l'on se trouve dans le contexte C avec une déclaration de type (sorte) S' bien formée et liée au type S'' (c'est à dire S renomme S''), alors, dans le même contexte C, la déclaration « **type** S **renames** S' **endtype** » donne un nouveau contexte dans lequel S est lié à S''.

Voici l'implémentation en OLGA de la règle de Plotkin précédente. Le lecteur pourra se reporter au schéma de l'arbre abstrait page 43 pour visualiser les opérateurs sur lesquels portent les règles sémantiques.

```
where op-type-synonym-definition -> ph-identifier1
                                    ph-identifier2
declare value
    { Le type 1 ne doit pas etre deja defini }
  CORRECT_ID1 : bool :=
    if ($identifier (ph-identifier1) = error-token) then
      false
```

```
      elsif (lookup_S ($identifier (ph-identifier1),
                       $MODULE_DC [ph-specification],
                       $h.SORT_ENV) != ERROR_SORT_BINDING) then
        message-elotos (ELOTOS_ALREADY_DECLARED_ID,
                        $MODE_DEBUG [ph-specification],
                        "{Du1}",
                        $scx(ph-identifier1))
      else
        true
      end if;

      { Le type 2 doit etre deja defini }
    CORRECT_ID2 : bool :=
      if ($identifier (ph-identifier2) = error-token) then
        false
      elsif (lookup_S ($identifier (ph-identifier2),
                       $MODULE_DC [ph-specification],
                       $h.SORT_ENV) = ERROR_SORT_BINDING) then
        message-elotos (ELOTOS_UNDECLARED_ID,
                        $MODE_DEBUG [ph-specification],
                        "{Du1}",
                        $scx(ph-identifier2))
      else
        true
      end if;

      RENAMED_S : TYPE_SORT :=
        if (CORRECT_ID2) then
          entry_S ($identifier (ph-identifier2),
                   $MODULE_DC [ph-specification],
                   $h.SORT_ENV)
        else
          ERROR_SORT
        end if;
  use
     $CORRECT := CORRECT_ID1 & CORRECT_ID2;

     { Liaison des sortes }
     $s.SORT_ENV :=
       if (CORRECT_ID1 & CORRECT_ID2) then
         insert_S ($identifier (ph-identifier1),
                   TYPE_SORT_BINDING (
                     NewNumber (),
                     RENAMED_S.BINDING.S_KIND,
```

```
                        $MODULE_DC [ph-specification],
                        TYPE_SORT_LIST (RENAMED_S),
                        TYPE_SORT_MAPPING ("", "", "", ""),
                        null),
                      $h.SORT_ENV)
          else
            $h.SORT_ENV
          end if;
     end where ;
```

La clause « `where` » permet d'indiquer quel nœud de l'arbre abstrait nous traitons. Puisque notre exemple porte sur le renommage de type, nous spécifions une règle sémantique sur l'opérateur « `op-type-synonym-definition` » appartenant au phylum « `ph-definition` ».

La règle sémantique est composée de deux parties :

- Dans la première, introduite par le mot clé « `declare` », on déclare des variables auxiliaires (ou attributs locaux) : « `CORRECT1` », « `CORRECT2` » et « `RENAMED_S` ». Le rôle joué par chacun de ces attributs est détaillé ci-dessous :
  - L'attribut local « `CORRECT_ID1` » est un booléen qui a la valeur « `true` » si l'attribut « `$identifier` » du phylum « `ph-identifier1` » possède une valeur de type « `token` » et n'existe pas déjà dans la table de liaison des types « `$h.SORT_ENV` » associée au module (S'il existe déjà, on émet le message d'erreur « `ELOTOS_ALREADY_DECLARED_ID` »).
  - L'attribut local « `CORRECT_ID2` » sert à vérifier que l'attribut « `$identifier` » du phylum « `ph-identifier2` » est bien une valeur de type « `token` » et que le type « renommé » est déjà déclaré. Dans le cas contraire, on affiche le message associé à l'identificateur « `ELOTOS_UNDECLARED_ID` ».
  - L'attribut « `RENAMED_S` » permet de récupérer les informations relatives au type renommé à l'intérieur de la table des symboles héritée « `$h.SORT_ENV` ».

  Le test « `if ($identifier (ph-identifier1) = error-token) then` », permet de s'assurer que les terminaux génériques n'ont pas été altérés par le processeur de recouvrement d'erreur. Effectivement, Recor peut corriger un source erroné en procédant à des insertions, des suppressions ou des substitutions de textes, selon des modèles définis par l'utilisateur. Or dans le cas où il manque un terminal générique dans le source du programme analysé, Recor ne peut pas choisir de texte significatif du point de vue sémantique. Dans ce cas, le terminal générique prend la valeur « `error-token` ». Les règles sémantiques écrites en Olga doivent donc contenir le code nécessaire pour traiter ce cas.

- Dans la deuxième, intoduite par le mot clé « `use` », on spécifie des calculs sur les attributs en fonction des valeurs déclarées précedemment et/ou d'autres attributs. Ces calculs sont décrits ci-dessous :

– On synthétise l'attribut « `$CORRECT` » qui servira ailleurs dans la grammaire attribuée et qui indique si la sémantique du renommage de type est correcte.
– Enfin, on spécifie le calcul de l'attribut « `$s.SORT_ENV` » : il reçoit une table de symboles, résultat de l'ajout d'un nouvel élément à la table de symboles héritée « `$h.SORT_ENV` ».

Sur cet exemple on peut constater que nous avons utilisé des fonctions, comme « `entry_S` » et « `insert_S` ». Ces fonctions sont définies ailleurs, dans le module « `def_elotos_types.olga` ». La possibilité d'écrire de telles fonctions améliore considérablement la lisibilité des grammaires attribuées et permet une bonne réutilisation du code.

Nous avons employé à dessein le terme de « spécification » car, à aucun moment, nous ne nous soucions de l'ordre dans lequel sont calculés les attributs. Cette tâche est prise en charge par l'évaluateur d'attributs EVALGEN.

4.4.3 Les divers modules produits

Les spécifications écrites		
Nom du fichier	*Nb lignes*	*Description*
`dec_elotos_messages.olga`	85	Module de déclaration des messages + fonctions de gestion des messages
`def_elotos_messages.olga`	149	Module de définition des messages + fonctions de gestion des messages
`ag_elotos_typage.olga`	6985	Grammaire attribuée procédurale pour la phase de vérifications sémantiques
`dec_ag_elotos_typage.olga`	12	Module de déclaration de la grammaire attribuée pour le typage
`dec_elotos_types.olga`	669	Module de déclaration des types utilisés
`def_elotos_types.olga`	1471	Module de définition des types utilisés

4.5 Traduction vers LOTOS

Le but de cette section est de décrire la méthode utilisée pour réaliser la traduction du langage E-LOTOS vers le langage LOTOS.

Au cours de la phase d'analyse, nous avons manipulé des arbres abstraits ayant la forme et les attributs nécessaires aux divers contrôles réalisés. Nous pourrions réaliser directement la traduction vers le langage cible (LOTOS) à partir du dernier arbre E-LOTOS produit dans la phase d'analyse, mais cela n'est pas souhaitable, pour au moins deux raisons :

- cet arbre n'est pas adapté à la forme du langage souhaité en sortie et contient des attributs qui n'ont aucune utilité pour la phase de traduction ;

- si la grammaire du langage est profondément remaniée, l'arbre abstrait en sortie de la phase d'analyse sera également bouleversé, et nous devrons modifier le code du décompilateur d'arbres PPAT pour qu'il s'adapte au nouvel arbre abstrait.

Nous avons choisi d'effectuer cette traduction en deux étapes successives (voir le schéma 4.4 page 58 :

1. traduire le dernier arbre abstrait E-LOTOS vers un arbre abstrait ayant la forme et les attributs nécessaires à la production du langage cible. Cette traduction est écrite en OLGA.

2. traduire l'arbre abstrait LOTOS vers un programme LOTOS à l'aide de l'outil PPAT.

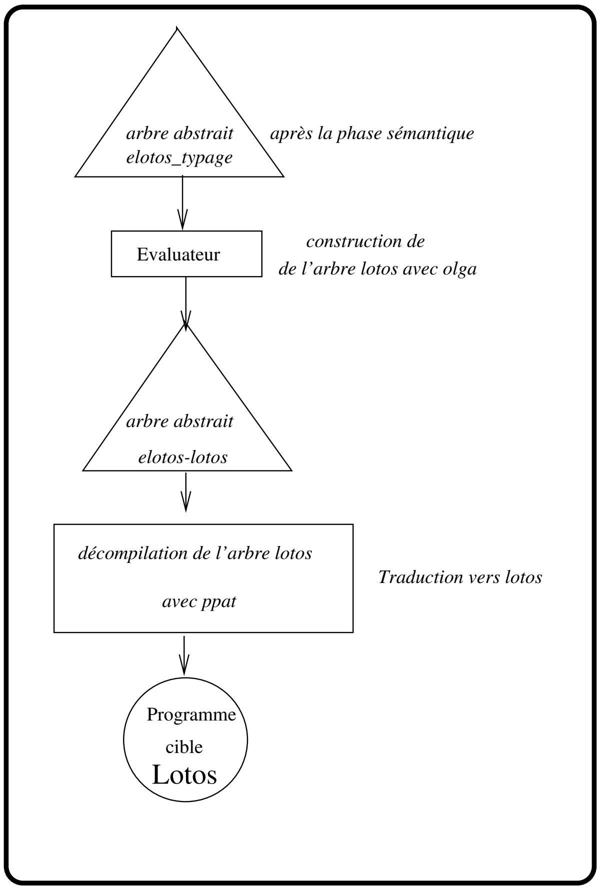

Figure 4.4: Traduction vers LOTOS

Avec le système SYNTAX/FNC-2 nous pouvons découper la production de code intermédiaire en quatre étapes :

1. identifier les constructions à traduire et donner un schéma de traduction ;

2. décrire la syntaxe attribuée abstraite qui permettra la création d'un arbre attribué abstrait ayant la structure du langage cible et les attributs nécessaires ;

3. décrire la grammaire attribuée OLGA (fonctionnelle) qui créera l'arbre de l'étape 2 ;

4. décrire la spécification du décompilateur d'arbre PPAT.

L'étape d'écriture de la syntaxe abstraite pour le langage cible a déjà été décrite dans la section 4.2.4, la suite de cette section décrit les étapes 1, 3 et 4.

4.5.1 Création du schéma de traduction

Pour produire un programme acceptable par le compilateur CÆSAR pour LOTOS, nous devons d'abord traduire la forme générale d'un programme E-LOTOS vers la forme générale d'un programme LOTOS. Le schéma suivant décrit cette traduction :

$Trad_{lotos}$ désigne une fonction traduisant des éléments de spécification E-LOTOS en éléments de spécification LOTOS. Ceci nous permet de simplifier les schémas de traduction, en invitant le lecteur à se reporter aux schémas de traduction des éléménts concernés.

Spécification E-LOTOS	Spécification LOTOS

specification Spec **is**
  `<declarations de`
  `types et de processus>`*
 	
behaviour
  `<B>`
endspec
	specification Spec **: noexit**
	library Boolean, Natural endlib
	  $Trad_{lotos}$(`<declarations de types>`)
	behaviour
	  $Trad_{lotos}$(`<B>`)
	where
	  $Trad_{lotos}$(`<declarations de processus>`)
	endspec

Afin de simplifier le code à produire, nous avons décidé de traduire l'ensemble des déclarations de types E-LOTOS en un seul type LOTOS ayant la structure syntaxique suivante :

type NomDeSpécificationE-LOTOS **is Boolean Natural**
  sorts
    <déclarations de sortes>
  opns
    <déclarations de constructeurs>
    <déclarations de fonctions>
  eqns
    <équations>
endtype

Le langage LOTOS produit utilise les conventions et les extensions particulières aux compilateurs CÆSAR et CÆSAR.ADT.

Le tableau ci-dessous présente la traduction des types E-LOTOS vers la clause « `sort` » du langage LOTOS.

Déclaration E-LOTOS	Traduction en clause **sorts** LOTOS
type S **renames** S' **endtype** où **S** est un type externe	S (*! **implementedby** S **external** *)
type S **renames** S' **endtype** où **S** n'est pas un type externe	S
type S **is external endtype**	S (*! **implementedby** S **external** *)
type S **is** $C_1(V_1{:}S_1, \cdots, V_n{:}S_n)_1$ | $\vdots$ | $C_m(V_1{:}S_1, \cdots, V_n{:}S_n)_m$ **endtype**	S

Le tableau ci-dessous présente la traduction des types E-LOTOS vers la clause « `opns` » du langage LOTOS.

Déclaration E-LOTOS	Traduction dans la partie **opns** LOTOS
type S **renames** S' **endtype** si S' est un type external	*Rien à traduire*
type S **renames** S' **endtype** si S' n'est pas un type external	C_1`(*! constructor *) :` $S_{1_1}, \cdots, S_{n_1}$ `->` S $\vdots$ C_m`(*! constructor *) :` $S_{1_m}, \cdots, S_{n_m}$ `->` S On recopie les constructeurs de S' dans lesquels on substitue S à S'.
type S **is external endtype**	*Rien à traduire*
type S **is** $C_1(V_1{:}S_1, \cdots, V_n{:}S_n)_1$ | $\vdots$ | $C_m(V_1{:}S_1, \cdots, V_n{:}S_n)_m$ **endtype**	C_1`(*! constructor *) :` $S_{1_1}, \cdots, S_{n_1}$ `->` S $\vdots$ C_m`(*! constructor *) :` $S_{1_m}, \cdots, S_{n_m}$ `->` S

4.5.2 Ecriture de la grammaire attribuée fonctionnelle

Le but de cette grammaire est de construire un arbre abstrait pour la production de code cible. Elle crée un arbre LOTOS à partir d'un arbre E-LOTOS.

Ci-dessous, nous présentons un extrait du fichier « `ag_elotos-lotos.olga` » contenant cette grammaire attribuée. L'en-tête de cette grammaire est le suivant :

```
attribute grammar elotos-lotos (elotos_typage in ph-elotos-axiom) :
                              ($s_LOT_PROGRAM : ph-lot-axiom ;
                               $s_CORRECT_LOTOS :  bool ;
                               $s_LOT_PROGRAM_IDENTIFIER : string) is
```

Il spécifie que la grammaire attribuée « `elotos-lotos` » :

- admet en entrée un arbre abstrait décrit par la syntaxe « `elotos_typage` » (voir section 4.2.3) et dont l'axiome est « `ph-elotos-axiom` » ;

- délivre en sortie une structure donnant accès à trois attributs, dont le premier (« `$s_LOT_PROGRAM` ») est la représentation de l'arbre de sortie, le deuxième (« `$s_CORRECT_LOTOS` ») un booléen nous indiquant si les contrôles sémantiques sont corrects, et le troisième (« `$s_LOT_PROGRAM_IDENTIFIER` ») un identificateur utilisé pour nommer le fichier LOTOS contenant le code cible généré par PPAT.

Ci-dessous nous présentons la déclaration des attributs :

```
const
        { Extension des fichiers LOTOS generes }
        LOT_EXTENSION := ".lotos" ;
attribute
        { Pour activer/desactiver le mode DEBUG }
  inherited $h_MODE_DEBUG (ph-specification) : bool ;

        { Pour donner un nom au type aplati }
  inherited $h_DEFINITION_IDENTIFIER (ph-definitions) : token ;

        { Nom de fichier en sortie, cet attribut est visible du
          programme traian_smp.c }
  synthesized $s_LOT_PROGRAM_IDENTIFIER (ph-elotos-axiom) : string ;

        { $s_CORRECT_LOTOS = false en cas de problème }
  synthesized $s_CORRECT_LOTOS (ph-elotos-axiom) : bool ;

        { Modeles pour la creation des noeuds de l'arbre en sortie }
  synthesized $s_LOT_PROGRAM (ph-elotos-axiom) : ph-lot-axiom ;
  synthesized $s_LOT_DECLARATIONS_MODEL (ph-definitions)
                                        : ph-lot-declarations ;
  synthesized $s_LOT_BEHAVIOUR_MODEL (ph-behaviour)
                                        : ph-lot-behaviour-expression ;
  synthesized $s_LOT_SORTS_DECLARATION_MODEL (ph-definition)
                                        : ph-lot-sorts-declaration ;
```

Ces attributs sont en des modèles d'arbres pour la construction de l'arbre abstrait en sortie. Effectivement, un attribut qui est déclaré de type phylum peut contenir des valeurs correspondant à différentes formes d'arbres, selon le nombre d'opérateurs détenus par le phylum (voir à ce sujet la notion de syntaxe abstraite présentée dans la section 3.8.1 page 24).

La règle sémantique présentée ci-dessous s'applique au phylum « `ph-specification` ». Elle permet de créer l'arbre LOTOS.

```
{ Règle sémantique pour le phylum ph-specification }
```

```
where op-specification  -> ph-identifier
                           ph-definitions
                           ph-behaviour
use
        { On donne un nom au type LOTOS aplati }
    $h_DEFINITION_IDENTIFIER (ph-definitions) :=
                                         $identifier (ph-identifier) ;

        { On vérifie la correction }
    $s_CORRECT_LOTOS := $s_CORRECT_LOTOS (ph-identifier) &
                        $s_CORRECT_LOTOS (ph-definitions) &
                        $s_CORRECT_LOTOS (ph-behaviour) ;

        { On donne un nom au fichier LOTOS produit }
    $s_LOT_PROGRAM_IDENTIFIER :=
                string ($identifier (ph-identifier)) + LOT_EXTENSION ;

    $s_LOT_PROGRAM := op-lot-axiom (op-lot-specification-identifier ()
                      with
                         $LOT-IDENTIFIER := $identifier(ph-identifier)
                      end with,
                      $s_LOT_DECLARATIONS_MODEL (ph-definitions),
                      $s_LOT_BEHAVIOUR_MODEL (ph-behaviour),
                      $s_LOT_WHERE_DECLARATIONS_MODEL (ph-definitions)) ;
end where ;
```

L'essentiel de la règle sémantique pour le phylum « `ph-specification` » consiste à :

- initialiser l'attribut hérité « `$h_DEFINITION_IDENTIFIER` » avec le nom de la spécification E-Lotos (cet attribut servira à nommer le type aplati) ;
- calculer l'attribut « `$s_CORRECT_LOTOS` » en effectuant un « et » logique entre les attributs « `$s_CORRECT_LOTOS` » des fils ;
- donner un nom pour la spécification Lotos cible ;
- créer l'arbre en sortie.

La règle sémantique présentée ci-dessous s'applique au phylum « `ph-definitions` ». Elle illustre le traitement des listes.

```
    { Règle sémantique pour le phylum ph-definitions }
where op-definitions -> ph-definition *
declare value
```

```
  Sorts : ph-lot-sorts-declarations :=
            map left CreateLotSortsDeclarations
                value null-ph-lot-sorts-declarations ()
                other $s_LOT_SORTS_DECLARATION_MODEL (ph-definition)
            end map ;
  Opns : ph-lot-opns-declarations :=       ...
  Forall : ph-lot-variable-declarations := ...
  Eqns : ph-lot-eqns-declarations :=       ...
use
  $s_CORRECT_LOTOS := map left &
                          value true
                          other $s_CORRECT_LOTOS (ph-definition)
                      end map ;

  $s_LOT_DECLARATIONS_MODEL :=
                 op-lot-declarations (Sorts, Opns, ForallEqns)
                 with
                     $LOT-IDENTIFIER := $h_DEFINITION_IDENTIFIER
                 end with ;

  $s_LOT_WHERE_DECLARATIONS_MODEL :=
              map left CreateLotWhereDeclarations
                  value null-ph-lot-where-declarations ()
                  other $s_LOT_WHERE_DECLARATION_MODEL (ph-definition)
              end map ;
end where ;
```

Le code ci-dessus est un exemple d'utilisation de l'opérateur « `map` » (présenté à la section 3.8.3). Il a été simplifié de manière à ne traiter que la partie relative à notre exemple de déclaration de type. Nous décrivons ci-dessous, les fonctions qu'il réalise.

- Il regroupe toutes les déclarations de sortes en vue de la création du type unique Lotos. L'arbre représenté par « `ph-definitions` » est une liste d'arbres de type « `ph-definition` ». Cet arbre est parcouru de gauche à droite grâce à l'opérateur « `map` » et à une fonction spéciale « `CreateLotSortsDeclarations` ». Nous avons écrit cette fonction pour ignorer les attributs « `$s_LOT_SORTS_DECLARATION_MODEL` » contenant une valeur de type « `null-ph-lot-sorts-declaration` ».

- Il répercute les éventuelles anomalies détectées dans la construction des fils en calculant l'attribut « `$s_CORRECT_LOTOS` ». On voit que c'est l'opérateur logique « `&` » qui est utilisé comme fonction de cumul.

- Il construit le type unique Lotos.

- Il construit la partie « `where` » conformément au schéma de traduction donné en B.1 page 129.

4.5.3 Décompilation d'arbre avec Ppat

L'arbre attribué crée par la grammaire « `ag_elotos-lotos.olga` » doit être décompilé pour produire le code cible. C'est le langage Ppat qui nous permet de spécifier le décompilateur. Pour cela, seule nous importe la structure de l'arbre abstrait sur lequel Ppat opère. Comme on l'a vu en section 4.2.4, celui-ci a été volontairement adapté à la forme du langage cible pour que l'écriture de la spécification Ppat devienne très simple.

L'écriture d'une spécification Ppat est très semblable à l'écriture d'une spécification Olga, car on écrit des règles de décompilation similaires aux règles sémantiques pour chaque nœud de l'arbre abstrait à décompiler. Les règles de décompilation sont le pendant des règles sémantiques Olga, elles comportent un format d'édition utilisant un langage à base de boîtes.

Les boîtes peuvent être vides, simples ou composées. Lorsque les boîtes sont composées, elles sont assemblées les unes par rapport aux autres, au moyen de séparateurs. On compose plusieurs boîtes en les mettant entre crochets (« `[ ]` ») et en les faisant préceder d'un séparateur.

Des exemples de chacun des formats, sont présentés dans la figure 4.5.3 page 66. Sur chacune des figures de l'exemple, les boîtes sont représentées par des cadres contenant une même lettre. Par exemple, le séparateur « `<h 1>` » (qui signifie «Alignement horizontal suivi d'un caractère blanc»), agencera deux boîtes « `B1` » et « `B2` » de telle manière que la première ligne de « `B2` » soit alignée avec la dernière ligne de « `B1` » et qu'il existe un espace horizontal équivalent à un caractère entre les deux lignes. Sur la figure 4.5.3, on peut visualiser cet agencement, ainsi que les résultats de huit autres formats utilisés dans la spécification Ppat.

Nous allons maintenant, commenter un extrait de la spécification Ppat pour la traduction vers Lotos.

```
from elotos_messages import  all ;

separator
  <h1> = <h 1> ;
  <h0> = <h 0> ;
  <v00> = <v 0 , 0> ;
  <v01> = <v 0 , 1> ;
  <v04> = <v 0 , 4> ;
  <v40> = <v 4 , 0> ;
  <v20> = <v 2 , 0> ;
  <hov100> = <hov 1 , 0 , 0> ;
```

Dans la section « `separator` », on peut nommer les formats d'affichage afin de rendre plus

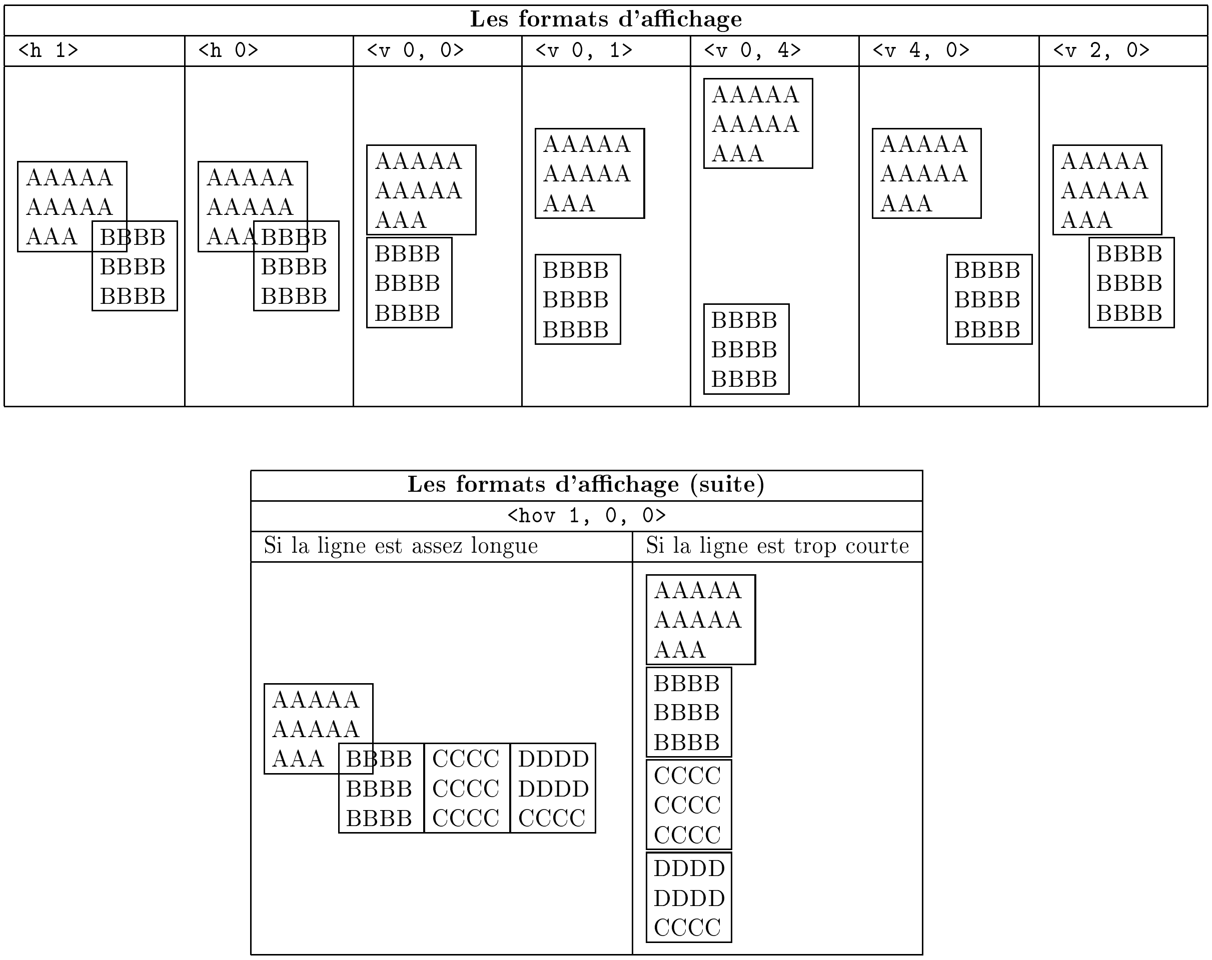

Figure 4.5: Les formats de boîtes pour l'affichage

aisée l'écriture de la spécification. Un séparateur se déclare ainsi : « `<NomDeSeparateur> = <FormatSeparateur>` », ce que l'on peut constater sur le texte ci-dessus.

L'exemple suivant montre comment l'on traite les listes.

```
where type SORT_TABLE pprint
        pplist SORT_TABLE left
                global h1
                insert LOT_COMMA_SYMBOL
                subbox SORT_TABLE
        end pplist
end where ;
```

D'une manière générale, lorsqu'un opérateur d'arité variable ou un attribut de type liste doit être affiché, on utilise l'instruction « `pplist` ».

Dans la règle présentée ci-dessus, on spécifie comment décompiler un attribut de type liste. En l'occurrence il s'agit de la décompilation de l'attribut « `SORT_TABLE` » qui est une liste de tokens (chaînes de caractères). Lorsque cet attribut apparaîtra dans une boîte, il sera décompilé de gauche à droite. Comme le format « `h1` » est utilisé, un caractère « espace » sera affiché avant la décompilation de la liste. Le mot clé « `insert` », permet d'insérer automatiquement un séparateur entre chaque token lorsque l'arité de la liste est supérieure à un. Dans l'exemple ci-dessus, lorsque l'attribut-liste « `SORT_TABLE` » contient plus d'un token, ceux-ci sont écrits un à un et sont séparés par des virgules (« `LOT_COMMA_SYMBOL` »).

```
    { Décompilation du phylum ph-lot-axiom }
where null-ph-lot-axiom ->
pprint
end where ;

where op-lot-axiom ->  ph-lot-specification-identifier
                       ph-lot-declarations
                       ph-lot-behaviour-expression
                       ph-lot-where-declarations
pprint
  case ph-lot-where-declarations is
    null-ph-lot-where-declarations () :
        [v00    [h1 "specification" ph-lot-specification-identifier ": noexit"]
                [h1 "library BOOLEAN, NATURAL endlib"]
                [v40 ph-lot-declarations]
                "behaviour"
                [v40 ph-lot-behaviour-expression]
                "endspec"] ;
    other :
        [v00    [h1 "specification" ph-lot-specification-identifier ": noexit"]
```

```
                    [h1 "library BOOLEAN, NATURAL endlib"]
                    [v40 ph-lot-declarations]
                    "behaviour"
                    [v40 ph-lot-behaviour-expression]
                    "where"
                    ph-lot-where-declarations
                    "endspec"] ;
  end case
end where ;
```

Nous sommes obligés d'écrire une règle pour le cas où le phylum « `ph-lot-axiom` » possède un fils construit par l'opérateur « `null-ph-lot-axiom` »[19].

La première règle de décompilation associée à l'opérateur « `op-lot-axiom` » écrit la structure générale d'une spécification E-LOTOS.

On peut constater que le filtrage (*pattern-matching*) sur les arbres est mis à profit : ici l'on décide de ne pas générer le mot clé « `where` » lorsque l'arbre abstrait relatif à cette partie est vide (cas « `null-ph-lot-where-declarations ()` »).

Ensuite on écrit les règles de décompilation dans l'ordre qui nous convient le mieux. La traduction du renommage de type E-LOTOS, s'effectue ainsi :

```
    { Décompilation du phylum ph-lot-sorts-declaration }
...
...
    { Cas du renommage de types }
where op-lot-type-synonym-sorts-declaration
                        -> ph-lot-internal-external-type-identifier
pprint
        ph-lot-internal-external-type-identifier
end where ;


    { Ce cas ne doit pas se produire }
where null-ph-lot-internal-external-type-identifier ->
pprint
end where ;

    { Cas d'un renommage de type précédemment déclaré }
where op-lot-internal-type-identifier -> ph-lot-type-identifier
```

[19] C'est PPAT qui nous oblige à tester ce cas. En réalité ce cas ne se produira pas, car la grammaire qui a crée l'arbre que décompilons, valorise un attribut « `$s_CORRECT_LOTOS` » qui nous permet de ne pas appeler le décompilateur.

```
pprint
        ph-lot-type-identifier
end where ;

    { Cas d'un renommage de type externe }
where op-lot-external-type-identifier-> ph-lot-type-identifier1
                                        ph-lot-type-identifier2
pprint
        [h1     ph-lot-type-identifier1
                LOT_BEGIN_EXTERNAL_COMMENT
                ph-lot-type-identifier1
                LOT_END_EXTERNAL_COMMENT]
end where ;


    { Décompilation d'un token (identificateur) }
where null-ph-lot-internal-type-identifier
                        ->
pprint
end where ;

where op-lot-internal-type-identifier
                        ->
pprint
        $LOT-IDENTIFIER
end where ;
```

Le reste de la spécification PPAT ne pose pas plus de difficultés et respecte souvent les règles simples que nous venons de mettre en œuvre dans l'exemple ci-dessus. Toutefois, dans certains cas, nous avons besoin de réaliser des décompilations spécifiques. Nous pouvons avoir besoin de transmettre certaines informations dans l'arbre, par le moyen de *contextes*.

La suite de cette section montre comment nous avons résolu le problème particulier de la traduction de l'expression de comportement E-LOTOS « `i` » (*internal action*). Le schéma de traduction de cette expression est donné en B.7 page 136. En E-LOTOS, la composition séquentielle binaire s'exprime par l'opérateur « `;` » utilisé pour combiner deux expressions de comportement.

Lorsque « `i` » est en partie droite d'une composition séquentielle, il faut générer le mot clé « `stop` ». En revanche, si « `i` » figure en partie gauche, il faut le laisser tel quel. Ainsi, l'expression « `i;i;i` » doit être traduite en « `i;i;i;stop` », tandis-que l'expression « `i` » doit être traduite en « `i;stop` ».

Nous allons compléter un peu la description de la syntaxe abstraite `elotos-lotos` donnée en 4.2.4 page 46, en donnant la description des expressions de comportement.

```
        { ph-lot-behaviour-expression }
    ph-lot-behaviour-expression     =  op-lot-internal-action
                                       op-lot-sequential-composition-behaviour
                                       ...
            { Opérateur internal action }
        op-lot-internal-action      -> ;

            { Opérateur de composition séquentielle }
        op-lot-sequential-composition-behaviour
                                    -> ph-lot-behaviour-expression
                                       ph-lot-behaviour-expression ;
        ...
```

En décompilant l'opérateur « `op-lot-internal-action` », nous devons savoir si ce nœud dérive d'un « fils droit » de l'opérateur « `op-lot-sequential-composition-behaviour` ». Pour cela, nous allons définir un contexte sur le phylum « `ph-lot-behaviour-expression` ». Un contexte se comporte comme un attribut hérité. Il se déclare immédiatement après les noms de formats de séparateurs dans l'en-tête de la spécification PPAT. Nous donnons ci-dessous la définition du contexte « `$RIGHT_STOP` ».

```
    context
            { false => "i" est la dernière partie d'une
                       composition séquentielle }
            { true  => "i" n'est pas la derniere partie
                       d'une composition séquentielle }
        $RIGHT_STOP (ph-lot-behaviour-expression) : bool ;
```

A chaque fois que le phylum « `ph-lot-behaviour-expression` » apparaît parmi les fils d'un opérateur, nous devons affecter le contexte « `$RIGHT_STOP` » à « `true` » ou « `false` ». C'est sur le phylum « `ph-lot-behaviour-expression1` » appartenant au fils gauche de l'opérateur « `op-lot-sequential-composition-behaviour` » que l'on est certain de ne pas se trouver en partie droite d'une composition séquentielle. Dans la spécification PPAT, nous allons donc coder la règle suivante :

```
    where op-lot-sequential-composition-behaviour
                            -> ph-lot-behaviour-expression1
                               ph-lot-behaviour-expression2
    pprint
            [h1 ph-lot-behaviour-expression1 ($RIGHT_STOP := false)
                ";"
                ph-lot-behaviour-expression2 ($RIGHT_STOP := $RIGHT_STOP)]
    end where ;
```

La valeur « `false` » est donnée au premier fils (« `ph-lot-behaviour-expression1` ») de l'opérateur « `op-lot-sequential-composition-behaviour` ».

Pour le deuxième fils (« `ph-lot-behaviour-expression2` »), nous ne pouvons pas savoir si nous sommes réellement dans la dernière partie d'une composition séquentielle... Effectivement, l'opérateur que nous sommes en train de décompiler dérive du phylum « `ph-lot-behaviour-expression` » qui peut se trouver lui-même en partie gauche d'une composition séquentielle. La solution consiste donc à transmettre au deuxième fils la valeur du contexte de l'opérateur (« `ph-lot-behaviour-expression2 ($RIGHT_STOP := $RIGHT_STOP)` »).

Pour tout autre opérateur dérivant dans un phylum « `ph-lot-behaviour-expression` », nous donnons la valeur « `true` » au contexte « `$RIGHT_STOP` ».

L'utilisation du contexte « `$RIGHT_STOP` » se fait au moment de la décompilation de l'opérateur « `op-lot-internal-action` ». Nous donnons ci-dessous la règle de décompilation de cette opérateur.

```
where op-lot-internal-action
                        ->
pprint
        if $RIGHT_STOP = true then
            "i ; stop"
        else
            "i"
        end if
end where ;
```

4.5.4 Les divers modules produits

Le tableau suivant donne une vue quantitative de l'ensemble des modules écrits pour effectuer la phase de traduction.

Les spécifications écrites		
Nom du fichier	*Nb lignes*	*Description*
`ag_elotos-lotos.olga`	2544	Grammaire attribuée fonctionnelle (transformation d'arbre)
`dec_ag_elotos-lotos.olga`	14	Module de déclaration pour la transformation d'arbre
`def_elotos-lotos-types.olga`	342	Module de définition des types et fonctions utilisées lors de la transformation d'arbre
`dec_elotos-lotos-types.olga`	165	Module de déclaration des types et fonctions utilisées lors de la transformation d'arbre
`elotos-lotos-ppat.ppat`	1243	Spécification de décompilation (génération de code intermédiaire)

4.6 Paragrapheurs pour LOTOS et E-LOTOS

Nos réalisations les plus rapides, mais aussi les plus spectaculaires, concernent les deux paragrapheurs pour les langages LOTOS et E-LOTOS.

PARADIS est un système de paragraphage dirigé par la syntaxe. Il a été développé en 1985 à l'INRIA Rocquencourt par Mrs Philippe Deschamp et Pierre Boullier. PARADIS s'est imposé à nous car il comportait l'essentiel des fonctionnalités dont nous souhaitions disposer.

Au début, nous avons été confrontés au manque de documentation pour cet outil.

Après récupération d'un rapport de recherche [BD85], un important effort a été consacré pour assembler les divers fichiers C et modules disponibles. Une vingtaine de messages échangés avec les concepteurs de PARADIS et une analyse approfondie des fichiers C dont nous disposions, nous ont permis d'en comprendre le fonctionnement et l'utilisation.

Cet effort a été récompensé par le fait que nous sommes aujourd'hui capables de produire rapidement un paragrapheur, pour tout langage dont l'analyseur lexico-syntaxique a été construit avec SYNTAX.

Ci-dessous, nous donnons l'ensemble des opérations à réaliser pour obtenir un paragrapheur.

- La première chose à faire consiste à indenter et/ou exdenter la grammaire décrivant la syntaxe concrète (fichier « `.atc` » ou « `.bnf` ») afin de fabriquer un fichier « `.paradis` » utilisé en entrée de l'outil PARADIS. On peut même décider d'indenter la grammaire BNF dès la phase d'écriture de la syntaxe concrète. De cette manière, la grammaire BNF sera plus lisible, tout en définissant les consignes pour le paragraphage des programmes.

- Ensuite, il faut reprendre la partie d'analyse lexicographique du compilateur. Eventuellement, si l'utilisateur a des actions lexicales précises à réaliser (ce qui a été notre cas), il doit écrire un fichier « `actions.c` » selon le modèle que nous avons mis au point. Dans notre cas, nous avons dû créer une fonction qui change la casse des mots clés. Seulement quarante lignes de code C ont été nécessaires pour l'écriture de cette fonction.

- Il faut recopier le fichier « `.recor` » contenant les modèles pour la reprise sur erreur.

- Il faut recopier le fichier « `pp_main.c` » contenant la fonction principale du paragrapheur. Nous avons modifié ce fichier pour lui ajouter de nouvelles options et de nouvelles fonctions comme par exemple, la fonction de sauvegarde du texte source.

Finalement, nous avons préparé un « makefile » qui automatise ces différentes opérations.

4.6.1 Exemple 1 : le renommage de types en E-LOTOS

Une fois de plus, le renommage de types va nous servir d'exemple. Commençons par donner un extrait de la grammaire BNF indentée :

```
<type-definition> = <type-symbol>  <type-identifier>
                         <renames-symbol>  <type-identifier>
                    <end-type-symbol> ;
                    op-type-synonym-definition

<end-type-symbol> = "endtype" ;

<renames-symbol> = "renames" ;

<type-symbol> = "type" ;

<type-identifier> = <identifier> ;

<identifier> = %IDENTIFIER ;
               op-identifier ()
               with
                   $identifier := %IDENTIFIER
               end with
```

La grammaire indentée donnée ci–dessus permet à partir du texte :

```
type NomDeType1 renames NomDeType2 endtype
```

d'obtenir la sortie suivante :

```
type NomDeType1
   renames NomDeType2
endtype
```

Lors de la décompilation d'un arbre syntaxique, notre paragrapheur associe, à chaque nœud de l'arbre son indentation dans le texte produit. Dans l'exemple précédent, si le nœud correspondant au non–terminal « `<type-definition>` » est imprimé en première colonne de la première ligne, il en sera de même du nœud « `<type-symbol>` » et donc du terminal « `"type"` », tandis que le nœud « `<renames-symbol>` » sera imprimé en quatrième colonne de la deuxième ligne.

On voit dans l'exemple ci-dessus, que l'outil PARADIS va laisser cohabiter les directives de construction d'arbres, avec les règles de production BNF. Ceci est très important car cela nous permet de réutiliser la grammaire BNF (fichier « `.atc` ») du compilateur.

Il est à remarquer que la grammaire BNF va nous permettre non seulement des *indentations*, comme pour le cas de « `<renames-symbol>` », mais également ce qu'il est convenu d'appeler des *exdentations*, c'est à dire des indentations négatives. L'exemple suivant donne un exemple d'exdentation.

4.6.2 Exemple 2 : les enregistrements en PASCAL

La spécification donnée ci-dessous (tirée de [BD85]), a pour objet de faire ressortir visuellement la partie variante des enregistrements de programmes PASCAL, en alignant le mot-clé « `case` » avec les mots-clés « `record` » et « `end` » .

```
<RECORD-TYPE> = "record"
                    <FIELD-LIST>
                "end" ;

<FIELD-LIST> = <FIXED-PART> ";"
            <VARIANT-PART> ";"

<VARIANT-PART> = "case" <TAG> "of"
                     <VARIANT-LIST> ;
```

Soit le texte suivant :

```
record ... ; case tag of ... end
```

le paragarapheur donnera, en sortie, le texte suivant :

```
record
   ... ;
case tag of
   ... ;
end
```

Cet alignement est possible bien que la partie variante n'apparaisse pas en partie droite d'une règle définissant le non–terminal « <RECORD-TYPE> » : il suffit d'exdenter le non–terminal « <VARIANT-PART> » de la même quantité que son ancêtre « <FIELD-LIST> » a été indenté.

La figure 4.6 page 79 donne une vue synthétique des divers processeurs et modules à mettre en œuvre pour réaliser un paragrapheur avec l'outil Paradis.

Les paragrapheurs que nous avons produits réalisent les fonctions suivantes :

- analyse lexicale et syntaxique du texte source (en utilisant la grammaire qui a servi à la réalisation du compilateur) ;
- reprise sur erreur (insertion, correction automatique du texte source...) ;
- signalisation des erreurs ;
- paragraphage du texte selon les choix de l'utilsateur (il suffit d'indenter la grammaire Bnf selon ses préférences) ;
- production automatique d'un fichier « .bak » contenant la spécification non indentée ;
- soulignement ou mise en gras des mots-clés (optionnel) ;
- réglage de la marge droite avec possibilité d'écrêtage du texte sur la marge droite ou retour à la ligne en conservant la structure du texte source.

4.6.3 Exemple 3 : le produit de convolution systolique

Dans le tableau suivant, nous donnons un exemple réel de ce qu'est capable de produire notre paragrapheur pour Lotos. Cet exemple est tiré de [Gar89a].

<table>
<tr><th>Avant paragraphage</th><th>Après paragraphage</th></tr>
<tr><td>

```
specification
Systolic_Convolution [Y,
T] : noexit

hide X in
(Generator[X](AjoutBV)
|[X]|
Array [X, Y]
(W1, W2, W3))
whEre process
Generator [X] : noexit
:= X ! X1; X ! X2;
     X ! X3; X   ! XM;
    stop endproc endproc
 process Zero [Y] :
 noexit :=
      Y !
(0 of EXP);
 Zero [Y]
           endproc
ENDSPEC
```

</td><td>

```
specification Systolic_Convolution [Y, T] : noexit

behaviour

   hide X in
      (
         Generator [X] (AjoutBV)
         |[X]|
         Array [X, Y] (W1, W2, W3)
      )
where
   process Generator [X] : noexit :=
      X ! X1;
         X ! X2;
            X ! X3;
               X ! XM;
                  stop
   endproc

   process Zero [Y] : noexit :=
      Y ! (0 of EXP);
         Zero [Y]
   endproc
endspec
```

</td></tr>
</table>

Le programme non paragraphé est non seulement très mal indenté, mais il comporte des erreurs qui on été automatiquement corrigées par l'outil :

- le mot-clé « `behaviour` » a été ajouté automatiquement ;
- le mot-clé « `whEre` » a été normalisé en « `where` » ;
- le mot-clé « `endproc` » supplémentaire a été effacé ;
- le mot-clé « `ENDSPEC` » a été normalisé en « `endspec` » ;
- la casse des terminaux génériques est restée inchangée grâce à l'action lexicale que nous avons programmée.

Les tableaux suivants donnent une vue quantitative de l'ensemble des modules écrits pour réaliser les paragrapheur.

Spécifications écrites pour le paragrapheur LOTOS		
Nom du fichier	*Nb lignes*	*Description*
`Makefile`	184	Fichier de directives permettant de produire automatiquement un paragrapheur
`actions.c`	49	Fonction passant en minuscules les mots-clés
`lotos.cpp`	1074	Grammaire BNF contenant les directives de paragraphage
`lotos_kw.h`	67	Définition des mot-clés sous forme de « `#define` »
`lotos.lecl`	191	Fichier description lexicale (modifié pour la conservation des commentaires)
`lotos.recor`	70	Fichier standard pour la récupération des erreurs
`pp_main.c`	418	Sauvegarde du fichier à paragrapher et appel du paragrapheur

Spécifications écrites pour le paragrapheur E-LOTOS		
Nom du fichier	*Nb lignes*	*Description*
`Makefile`	184	Fichier de directives permettant de produire automatiquement un paragrapheur
`actions.c`	49	Fonction passant en minuscules les mots-clés
`traian.cpp`	2122	Grammaire BNF contenant les directives de paragraphage
`traian_kw.h`	112	Grammaire BNF contenant les directives de paragraphage
`traian.lecl`	191	Fichier description lexicale (modifié pour la conservation des commentaires)
`traian.recor`	82	Fichier standard pour la récupération des erreurs
`pp_main.c`	418	Sauvegarde du fichier à paragrapher et appel du paragrapheur

4.7 Tests

Pour effectuer les tests sur le compilateur TRAIAN, nous avons écrit plusieurs spécifications E-LOTOS pour lesquels nous attendions un comportement précis du compilateur.

Nous avons aussi écrit un script qui permet d'exécuter un test particulier ou l'ensemble des tests, en vérifiant qu'il n'y a pas de régression par rapport au(x) test(s) précédent(s). Ce script stocke les résultats du premier test dans un répertoire de référence. Lors de tests ultérieurs ces résultats sont comparés avec ceux du test courant. Si une différence existe, un message nous signale une non conformité avec la référence. Le script comporte également une option « `-copy` » qui nous permet de mettre à jour les fichiers de référence.

En ce qui concerne les tests pour les paragrapheurs, c'est un fichier de directives pour l'utilitaire `make`, qui nous permet de les effectuer. Lorsqu'un test de paragraphage est demandé, le fichier de référence est copié sous un nom différent puis le nouveau fichier est soumis au paragrapheur. Ceci nous permet de « rejouer » les mêmes tests sans que les paragrapheurs ne les modifient.

Les fichiers de test sont constitués de spécifications LOTOS ou E-LOTOS.

Les divers modules produits pour les tests du compilateur TRAIAN

Fichiers ayant servi aux tests		
Nom du fichier	*Nb lignes*	*Description*
`try1`	146	Script permettant d'effectuer les tests et de vérifier la conformité avec une référence
`user:(001-045).elot`	3358	Ensemble des spécifications E-LOTOS ayant servi aus tests

Les divers modules produits pour les tests des paragrapheurs

Fichiers ayant servi aux tests		
Nom du fichier	*Nb lignes*	*Description*
`Makefile (2 fichiers)`	180	Fichiers de directives pour effectuer les tests des paragrapheurs
`testxxx (20 fichiers)`	3664	Spécifications LOTOS et E-LOTOS

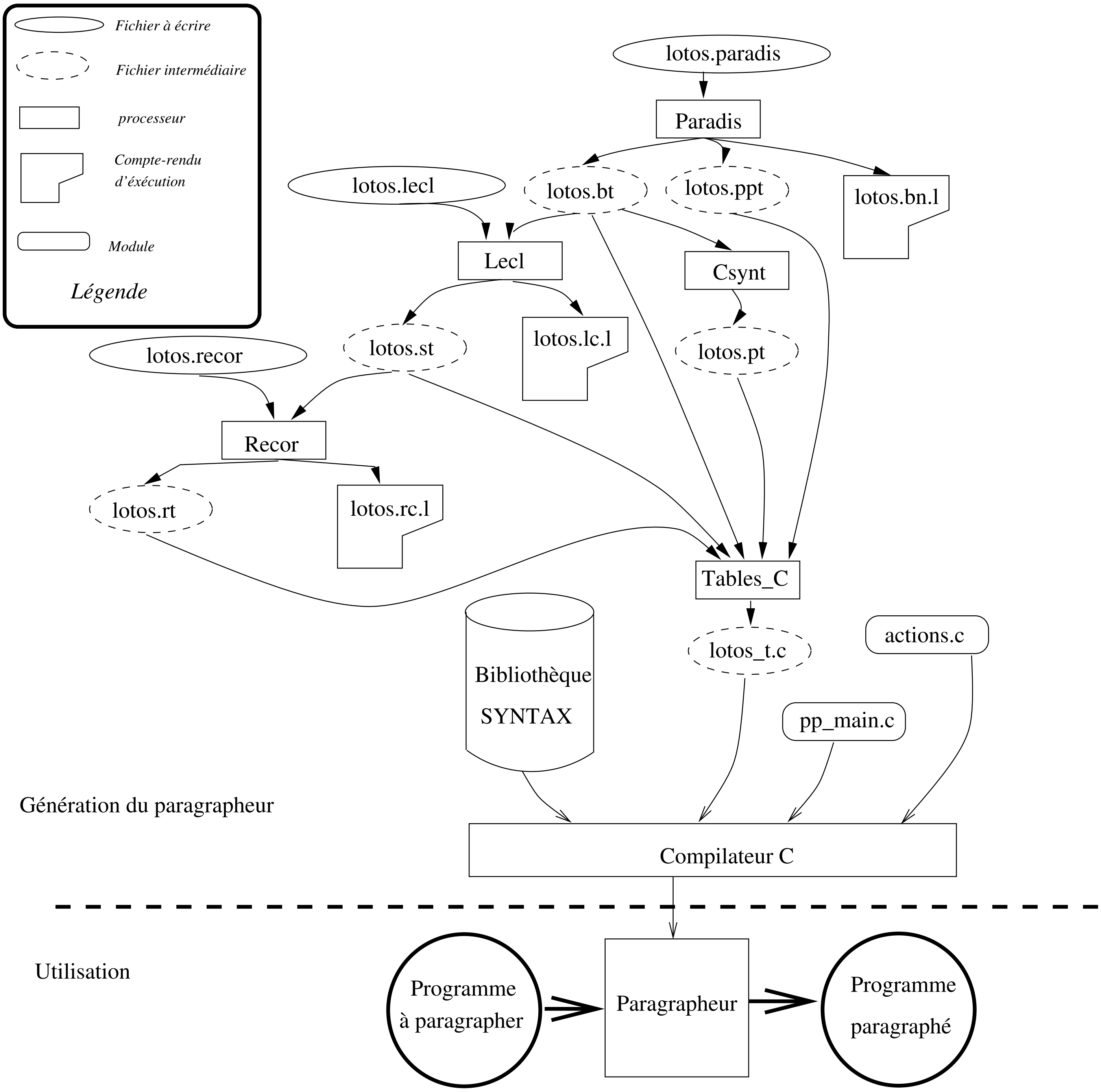

Figure 4.6: Fonctionnement du paragrapheur

Chapitre 5

Conclusion

Ce chapitre présente le bilan et les perspectives du projet réalisé. Pour la partie bilan, nous présenterons successivement :

- les réalisations logicielles ;
- une application de ces réalisations ;
- les apprentissages et expériences liés au projet ;
- une évaluation des outils utilisés ;
- et enfin, une liste de contributions diverses.

5.1 Bilan du travail effectué

5.1.1 Réalisations logicielles

Compilateur pour le langage E-LOTOS : Le premier prototype du compilateur E-LOTOS a été réalisé en deux mois et il a servi comme BNF pour le Committee Draft. Il était constitué de la partie avant comprenant l'analyse lexico-syntaxique et l'analyse de sémantique statique. L'analyse de sémantique statique regroupait la liaison des types et des variables et correspondait à deux grammaires attribuées dans deux modules distincts. Ce prototype était disponible dès janvier 1997 et a continué à évoluer jusque fin mars 1997, date à laquelle il comportait la partie arrière, effectuant la traduction vers le langage LOTOS.

A partir d'avril 1997, de profonds bouleversements ont eu lieu dans la syntaxe du langage, ce qui a entraîné également des changements dans les syntaxes abstraites et donc la nécessité de réécrire les grammaires attribuées. Seule la partie arrière du compilateur n'a pas subit de changement et le nouveau prototype a pu être produit en un mois environ.

Par la suite, nous avons beaucoup amélioré la traduction vers Lotos, pendant un mois environ.

Le tableau ci-dessous présente l'aspect quantitatif de ce travail.

Compilateur Traian			
Description	*Nb lignes*	*Nb modules*	*Langage*
Description lexicale E-Lotos	160	1	Lecl
Syntaxe concrète E-Lotos avec appel des constructeurs de l'arbre abstrait.	2777	1	Atc
Syntaxes abstraites	2148	4	Asx
Description des modèles de reprise sur erreur	82	1	Recor
Grammaires attribuées	12436	10	Olga
Spécification de décompilation de l'arbre final	1243	1	Ppat
TOTAL	**18846**	**18**	

Paragrapheur pour les langages Lotos et E-Lotos : en un mois et demi, nous avons réalisé deux paragrapheurs pour les langages Lotos et E-Lotos. Ils permettent de produire des sorties bien paragraphées, et d'effectuer certaines corrections du texte source. L'essentiel du temps a été dépensé en recherche documentaire et analyse de code pour mettre en œuvre le système Paradis. Deux jours environ ont été nécessaires pour indenter correctement les syntaxes Bnf des deux langages et pour l'écriture de la fonction de gestion de la casse des mots clés.

Le tableau ci-dessous donne l'aspect quantitatif de ce travail.

Paragrapheurs pour Lotos et E-Lotos			
Description	*Nb lignes*	*Nb modules*	*Langage*
Description lexicale (modifiée pour la conservation des commentaires)	333	2	Lecl
Actions lexicales	98	2	C
Grammaires Bnf et directives de paragraphage	3375	4	Bnf
Fichier standard pour la récupération des erreurs	152	2	Recor
Appel des constructeurs (procédure principale)	836	2	C
TOTAL	**4794**	**12**	

Jeux de tests : pour la mise au point du compilateur TRAIAN, nous avons produit des jeux de tests *rejouables*. Les tests sont mis en œuvre par un script capable de définir tel ou tel résultat de test, comme une *référence*, puis, lors d'une utilisation ultérieure, de rejouer ce test et comparer les résultats par rapport à la référence. Les tests peuvent s'effectuer individuellement ou par bloc.

Pour les deux paragrapheurs produits, nous avons également réalisé des jeux de tests rejouables.

Le tableau ci-dessous donne une vue quantitative des fichiers produits.

Fichiers ayant servi aux tests			
Description	*Nb lignes*	*Nb modules*	*Langage*
lanceur/vérificateur de tests pour le compilateur TRAIAN	146	1	« SHELL »
Jeux de tests pour le compilateur TRAIAN	3358	45	E-LOTOS
Lanceurs de tests pour les paragrapheurs	180	2	« SHELL »
Jeux de tests pour les paragrapheurs	3664	20	LOTOS et E-LOTOS
TOTAL	**7348**	**68**	

5.1.2 Application

Notre compilateur, TRAIAN, a été utilisé dans le cadre du standard IEEE-1394, aussi connu sous le nom de FIRE WIRE. Ce standard décrit un bus série haute performance pour les ordinateurs personnels (PC) nécessitant des débits très élevés, comme le traitement de la vidéo. A cette norme se sont ralliés des constructeurs comme AT&T, CANON, COMPAQ, FUJI, HEWLETT-PACKARD, IBM, KODAK, MICROSOFT, TEXAS INSTRUMENT, YAMAHA et quelques autres. Certaines de ces firmes commercialisent déjà des cartes FIRE WIRE.

Le protocole IEEE-1394 consiste en trois couches : la couche *transition*, la couche *liaison*, et la couche *physique*. En partant d'une description en μCRL [GP90] du protocole de transmission asynchrone des paquets pour la couche liaison [Lut97], nous avons spécifié ce protocole en E-LOTOS (en ajoutant la spécification de la couche transition).

Cette description E-LOTOS a été traduite en LOTOS en utilisant le compilateur TRAIAN et nous l'avons vérifiée à l'aide de la boîte à outils CADP.

Cette application a permis de détecter et de corriger un blocage potentiel causé par la sémantique ambiguë des machines d'états qui décrivent le protocole [SM97]. Cette inconsistance peut amener à une implémentation erronée du standard.

5.1.3 Apprentissage et expérience

Les réalisations logicielles précitées ont été l'occasion d'acquérir un certain nombre de connaissances, parmi lesquelles on peut citer :

Les méthodes formelles : nous avons appris des langages de spécification formelle, comme LOTOS et E-LOTOS. Le domaine de la compilation a été en lui-même une excellente illustration de ce que sont les méthodes formelles. Pour construire un compilateur, on commence par définir la grammaire du langage source au moyen de règles formelles appropriées. Après transformation de celles-ci, on peut obtenir un analyseur syntaxique efficace grâce à des mécanismes généraux découverts par Knuth dans les années soixante... On a ici tous les ingrédients d'une méthode formelle, à commencer par un langage formel pour décrire les règles de grammaire de manière précise – une BNF (forme normale de Backus-Naur) ; mais surtout on dispose d'une solide base mathématique sur la théorie des langages et des automates, qui donne la signification des règles de grammaire (*sémantique*) et justifie les algorithmes généraux utilisés ;

Les langages fonctionnels : nous avons étudié la programmation fonctionnelle, avec plus particulièrement, l'apprentissage du langage OLGA (langage de type ML).

Production de compilateurs et paragrapheurs : dans le domaine de la production de compilateurs, nous avons étudié des outils modernes basés sur les grammaires BNF, les syntaxes abstraites de description d'arbres, les grammaires attribuées [Knu68] ainsi que les techniques de compilation [ASU91, RD94]. Nous avons également étudié les techniques de production de paragrapheurs, avec des outils de haut niveau basés sur les grammaires BNF.

5.1.4 Evaluation de l'utilisation des outils

Problèmes rencontrés

Parmi les inconvénients rencontrés, nous pouvons citer :

Le temps d'apprentissage : avant d'utiliser le système SYNTAX/FNC-2, il faut acquérir de nombreuses connaissances sur les outils et langages. Pour la partie SYNTAX, il faut apprendre le méta-langage lexical LECL, le méta-langage syntaxique BNF, le langage PRIO et les techniques de description des modèles de recouvrement liés au traitement des erreurs. Pour la partie FNC-2, il faut apprendre les syntaxes abstraites, le langage OLGA et le langage PPAT.

Une documentation à améliorer : la documentation comporte des parties non encore implémentées dans le système, ce qui nous a occasionné une perte de temps non négligeable. Il faut souligner toutefois, que, mise à part pour les *exceptions* qui ne sont pas implémentées, rares sont les cas où l'utilisateur ne peut pas implémenter lui-même les fonctions manquantes.

L'indisponibilité du système : nous avons été privé du système pendant deux mois car Syntax/Fnc-2 n'existait qu'en version Sun-os et la seule machine qui fonctionnait sous ce système d'exploitation est tombée en panne.

Un manque de documentation pour Paradis : la mise en œuvre de l'outil Paradis s'est montrée difficile et longue. Nous avons dû contacter les auteurs car la documentation était insuffisante.

Avantages

Partie Syntax : nous avons décrit des analyseurs lexico-syntaxiques sans devoir produire de code C, mais uniquement en fournissant des fichiers de spécification (lexicale, syntaxique et de traitement des erreurs).

Partie Fnc-2 : le langage Asx nous a permis de décrire de manière homogène, tous les arbres abstraits utilisés dans notre compilateur, sans avoir à créer de fonction de construction, mais en utilisant simplement les constructeurs éprouvés fournis par le système. Nous avons particulièrement apprécié le fait que les syntaxes abstraites puissent partager, par le moyen de clauses d'importation, des phyla, opérateurs et attributs. Ceci a été très utile pour décrire les phases de compilation qui ne construisent pas de nouvel arbre, mais qui ajoutent des informations à leur arbre d'entrée.

Grâce à l'utilisation du langage fonctionnel *déclaratif* Olga, le code de notre compilateur n'effectue aucun effet de bord et nous n'avons jamais eu à nous soucier de l'ordre d'évaluation des attributs. La génération des règles de copies par défaut du genre « `$Attribut(Phylum_A) := $Attribut(Phylum_B)` », nous a permis de concentrer notre attention sur les calculs importants plutôt que sur de simples transferts d'informations [JP90]. Le paradigme de structuration fondé sur les productions, a permis un développement incrémental de notre compilateur et à partir d'un développement embryonnaire, nous avons pu tester progressivement l'outil Traian. Enfin, le générateur d'évaluateur Evalgen crée des évaluateurs très efficaces en temps. De ce fait, nous n'avons jamais eu à prendre en compte un quelconque souci d'efficacité.

L'utilisation de Ppat pour produire le code cible, a montré que c'était un outil pratique permettant de donner une représentation concrète des arbres manipulés par Fnc-2. La syntaxe du langage Ppat est très proche de la syntaxe du langage Olga, et nous avons appris très rapidement à l'utiliser. La possibilité de définir des formats d'édition sur les nœuds de l'arbre à décompiler (les opérateurs de la syntaxe abstraite), avec un langage de boîtes, nous a permis de produire du code cible bien indenté, ce qui facilite la mise au point du compilateur lors de la compilation du code cible. La possibilité de transmettre des informations de contexte sur les nœuds de l'arbre à décompiler, nous a procuré un bon moyen d'effectuer des traductions complexes, dans lesquelles un nœud est décompilé différemment, selon l'endroit où il se situe dans l'arbre.

Partie Paradis : nous avons pu produire des paragrapheurs/correcteurs, en écrivant qu'une seule fonction C (destinée à changer la *casse* des mots clés). La technique de description de paragrapheurs dirigés par la syntaxe s'est montrée très efficace. Sous certaines conditions[20], nous sommes capables de produire un paragrapheur en moins de 15 minutes, pour tout langage dont la partie lexico-syntaxique a été développée avec l'outil Syntax.

5.1.5 Autres contributions

Nous donnons ci-dessous, une liste d'activités diverses, réalisées pendant le séjour à l'Inria Rhône–Alpes.

- Nous avons contribué à l'amélioration du document E-Lotos User Language [SG96], par apport de corrections.
- Avec les membres du projet Oscar[21], notamment avec M. Didier Parigot, nous avons contribué à l'amélioration du système Syntax/Fnc-2, par des rapports d'anomalies (dans le code et la documentation) avec fourniture de pistes pour résoudre certains problèmes d'implémentation. Nous avons également donné quelques conseils de portage sur Solaris. Le résultat de cette collaboration s'est montré très positif, puisque nous utilisons désormais la version Solaris du système Syntax/Fnc-2 pour nos développements. Nous sommes à l'origine d'un projet de portage du système Syntax/Fnc-2, sur notre machine Linux.
- Nous avons réalisé la rédaction d'un petit manuel d'utilisation du système Syntax/Fnc-2 ;
- Nous avons également effectué les premiers essais d'une distribution de Syntax et Paradis pour Linux.
- Nous avons réalisé des tests et propositions d'améliorations sur les outils développés dans l'action Vasy, comme l'interface graphique Eucalyptus et l'outil Installator.
- Nous avons également effectué diverses actions d'assistance système, comme la réparation de PC, l'achat de matériels et l'installation des systèmes (Linux et Windows NT) sur machines PC.

5.2 Perspectives

La définition du langage E-Lotos, continue à évoluer et à s'affiner. Il y a eu un *Committee Draft* en juillet 97 et un *final Committee Draft* est en cours de préparation.

Le prototype Traian développé pendant ce projet, servira de base pour les futurs développements. Il devra être adapté pour tenir compte des nouvelles évolutions d'E-Lotos, ce

[20] Il faut que la syntaxe concrète Bnf ait été correctement indentée et qu'aucune action lexicale ne soit à écrire.

[21] Outils syntaxiques pour la construction et l'analyse de programme

qui ne devrait pas poser de problème, étant donné la flexibilité des outils employés pour son développement.

Derrière le front-end, on est en train de développer une partie finale qui comportera notamment, une traduction vers des réseaux de Pétri étendus (avec variables et horloges). A terme, le compilateur TRAIAN devrait être intégré à la boîte à outils CADP.

Appendix A

Exemple d'utilisation du système SYNTAX/FNC-2

Nous allons montrer sur l'exemple SIMPROC[22] comment générer un compilateur avec le système SYNTAX/FNC-2. 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 scalaire, 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 définit 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.

[22] SIMPROC est compris dans la distribution du système SYNTAX/FNC-2

- 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.

A.1 Initialisation de l'environnement de travail

Dans la suite du texte, « `$HOME` » indique le répertoire d'accueil de l'utilisateur.

SYNTAX/FNC-2 crée, pour ses propres besoins, un certain nombre de répertoires. Il est préférable de regrouper ceux-ci dans un répertoire de travail particulier. Notre but est de créer le compilateur SIMPROC, pour le langage SIMPROC, aussi, nous décidons d'appeler le répertoire de travail « `simproc` ». C'est à partir de lui que nous lancerons le script d'utilisation du système SYNTAX/FNC-2.

```
> mkdir $HOME/simproc
> cd $HOME/simproc
```

Afin d'effectuer un certain nombre d'initialisations, nous devons lancer le script « `FNC2` », ce qui nous permettra de :

- donner un nom à notre compilateur (SIMPROC) ;

- donner un nom à notre langage (SIMPROC) ;

- créer automatiquement les sous-répertoires utilisés par le système ;

- créer automatiquement les fichiers « `makesimproc` » et « `simproc.mkfnc2` » utilisés pour l'appel des divers constructeurs ;

- créer automatiquement les fichiers :

  « `simproc_main.c` » qui contient la procédure principale (« `main` ») de notre compilateur (nous ne modifierons pas ce fichier) ;

  « `simproc_sact.c` » qui permet d'appeler les éventuelles actions lexicales (dans notre cas, ce fichier n'est pas utilisé) ;

  « `simproc_smp.c` » qui permet d'appeler séquentiellement les grammaires attribuées ainsi que le décompilateur écrit avec PPAT ;

« `simproc.recor` » qui est le fichier par défaut contenant les directives de recouvrement des erreurs lexicales.

Après avoir lancé la commande « `FNC2` », nous voyons apparaître le menu ci-dessous.

```
> FNC2
-----------------------------------------
 To use this shell-script gnu-make and gcc compiler are require.
-----------------------------------------
 Main menu      (I|V|A|P|D|K|C|F|E|R)
 I nit          : Initialisation of environment variables f2, sx, etc
 V ariables     : Your environment variables
 A pplication   : Name of Application and of Language
 P refnc2       : mkdir of sub-directories (spec, src, ... etc)
 D ebmkfnc2     : creation of an X.MKFNC2 based on spec/*
 K akemkfnc2    : creation of makefiles based on f2aux/*
 C onstructor   : call constructors upon specifications
 F inal         : object (C) generation and compilation
 E xit          : Exit
 R m            : Clean
-----------------------------------------
 Sub-Menu replies
 y es           : yes
 n o            : no
 h elp          : help
 e xit          : exit
 m enu          : back to Main Menu
-----------------------------------------
```

Nous n'utiliserons pas les deux premières options « `I nit` » et « `V ariables` ». La première (« `I nit` ») permet d'initialiser diverses variables d'environnement. Mais nous avons supposé que l'installation avait été faite selon les prescriptions des auteurs de ce fait les initialisations de variables sont effectuées une fois pour toutes, lors de la phase de connexion. La deuxième option du menu (« `V ariables` ») permet de visualiser l'état des variables d'environnement. La troisième option (« `A pplication` ») permet de donner un nom au compilateur et au langage. C'est ce que nous faisons ci-dessous.

```
A
-----------------------------------
 Name of application : (simproc) and Name of language : (simproc)
 reply: (n|y|e|m|help)
-----------------------------------
y
```

Nous avons choisi l'option « `A pplication` ». Le système nous informe que le nom de l'application (le compilateur) est « `simproc` » et que le nom du langage est également « `simproc` » (le nom du répertoire courant sert à générer les noms par défaut). Ceci nous convient et nous répondons « `y` » à la question.

A la question suivante (« `prefnc2 simproc simproc ?` »), nous répondrons par « `y` », ce qui aura pour effet de créer les fichiers et répertoires dont nous avons parlé plus haut. Nous voyons alors apparaître un écran ressemblant à ceci :

```
-----------------------------------
 prefnc2 simproc simproc ? reply (n|y|l|e|m|help)
-----------------------------------
y
---------------
 Your environment variables
 sx = chemin complet du répertoire SYNTAX
 f2 = chemin complet du répertoire FNC-2
 MACHINE = solaris
 F2PATH = .:/chemin pour accés aux bibliothèques FNC-2
 LDFLAGS = chemin pour accès à la librairie syntax libsx.a
 CFLAGS = les options du compilateur
 PATH = vos chemins
---------------

mkdir ...
mkdir ... done
```

les variables « système » ont été listées pour information (masquées ici pour des raisons de confidentialité). Le système a créé les répertoires nécessaires.

Sur l'écran suivant, deux versions du compilateur sont proposées :

- une dans laquelle on ne dispose ni de compte-rendu de compilation, ni d'option de compilation ;
- une dans laquelle on dispose d'un compte-rendu et d'options de compilation.

Pour le moment, nous ne nous soucions pas des ces options, mais nous nous réservons le droit d'en bénéficier, c'est pourquoi on choisira l'option « `2` » ci-dessous.

```
-----------------------------
 Creation of X_main.c, X_smp.c and X_sact.c files: (n|1|2)
 1: without listing-output and main-options
 2: with listing-output and main-options
-----------------------------
2
```

Le système liste l'ensemble des répertoires et certains fichiers créés automatiquement (comme le fichier « `simproc.recor` »), après quoi il nous demande de décrire notre application.

```
 List of your application : /local_home/users/vivien/simproc
```

```
total 32
drwxr-xr-x  11 vivien   vasy        512 Sep 17 09:09 .
drwxr-x---  32 vivien   vasy       3072 Sep 17 09:09 ..
drwxr-xr-x   3 vivien   vasy        512 Sep 17 09:09 bin
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 f2aux
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 incl
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 incl_fnc2
drwxr-xr-x   3 vivien   vasy        512 Sep 17 09:09 lib
-rw-r--r--   1 vivien   vasy       1410 Sep 17 09:09 makesimproc
-rw-r--r--   1 vivien   vasy         41 Sep 17 09:09 simproc.mkfnc2
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 spec
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 src
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 src_fnc2
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 tmp

 List of your source file : /local_home/users/vivien/simproc/src

total 30
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 .
drwxr-xr-x  11 vivien   vasy        512 Sep 17 09:09 ..
-rw-r--r--   1 vivien   vasy       8322 Sep 17 09:09 simproc_main.c
-rw-r--r--   1 vivien   vasy        608 Sep 17 09:09 simproc_sact.c
-rw-r--r--   1 vivien   vasy       2979 Sep 17 09:09 simproc_smp.c

 List of your specification file : /local_home/users/vivien/simproc/spec

total 10
drwxr-xr-x   2 vivien   vasy        512 Sep 17 09:09 .
drwxr-xr-x  11 vivien   vasy        512 Sep 17 09:09 ..
-r-xr-x--x   1 vivien   vasy       2216 Sep 17 09:09 simproc.recor

---------------
 Describe your specifications
---------------

-------------------------
 debmkfnc2 ? reply (n|y|e|m|help)
-------------------------
```

L'écriture des spécifications pouvant prendre beaucoup de temps, il est préférable de quitter le script avant de se rendre dans le répertoire « `$HOME/simproc/spec` », pour y décrire le compilateur. C'est ce que nous faisons ci-dessous en choisissant l'option « `e` ».

```
-------------------------
 debmkfnc2 ? reply (n|y|e|m|help)
```

```
------------------------
e
```

A.2 Ecriture des spécifications lexico-syntaxiques

A.2.1 Analyse lexicale

Les descriptions lexicales se trouvent dans le fichier « `simproc.lecl` », listé ci-dessous :

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

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

Dans la partie « `Classes` », on définit une classe de caractères « `SPACE` ». Elle regroupe les caractères « `SP` » (espace), « `HT` » (tabulation horizontale), « `NL` » (ligne suivante) et « `FF` » (saut de page). Cette classe permet de simplifier l'écriture de « `Comments` », dans la partie « `Tokens` ».

Dans la partie « `Tokens` », on définit les commentaires et les unités lexicales génériques. Le mot clé « `Comments` » est associé à une expression régulière permettant de reconnaître un commentaire. Le « `-` » devant l'expression régulière indique que les commentaires ne seront pas gardés dans l'arbre syntaxique. Les deux terminaux génériques (« `%ID` » et « `%INT` »), permettent de reconnaître, respectivement, les identificateurs et les entiers.

A.2.2 Analyse syntaxique et recouvrement d'erreurs

L'analyse syntaxique doit déboucher sur la création d'un arbre abstrait. Ce dernier viendra en entrée d'un évaluateur d'attributs pour la phase suivante d'analyse de sémantique statique. La description de cet arbre se trouve dans le fichier de syntaxe abstraite « `simproc-base.asx` ». Cette syntaxe abstraite ne contient aucune déclaration d'attribut, ceux-ci sont déclarés dans la syntaxe abstraite « `simproc-in.asx` » qui importe la syntaxe de base « `simproc-base.asx` ». Ces deux fichiers sont listés ci-dessous.

```
grammar simproc-base is

{Une syntaxe abstraite non attribuee, pour le langage simproc}
{=============================================================}
root is PROGRAM ;
```

```
{Programme et blocs}
{==================}
PROGRAM      = program ;
program     -> BLOCK ;

BLOCK        = block ;
block       -> DECLS STMTS ;

{Declarations}
{============}
DECLS        = decls ;
decls       -> DECL* ;

DECL         = int-decl array-decl proc-decl ;
int-decl    -> ID {identificateur de variable} ;
array-decl  -> ID {identificateur de variable}
              NUMBER {dimension du tableau} ;
proc-decl   -> ID {identificateur de procedure}
               ID {identificateur de parametre par valeur}
               ID {identificateur de parametre par reference}
               BLOCK ;

{Instructions}
{============}
STMTS        = stmt-list ;
stmt-list   -> STMT+ ;

STMT         = assign call ifthenelse ;
assign      -> VAR EXPR ;
call        -> ID   {identificateur de procedure}
               EXPR {paramètre valeur}
               VAR  {parametre par reference} ;
ifthenelse  -> EXPR EXPR
               STMTS {partie si vrai}
               STMTS {partie si faux} ;

{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 {index};

{Terminaux}
{=========}
NUMBER      = number ;
number     -> ;

ID          = id;
id         -> ;
end grammar {simproc-base} ;
```

Le contenu de la syntaxe attribuée décrivant le premier arbre abstrait est listé ci-dessous.

```
{Une syntaxe abstraite attribuee, pour le langage simproc}
{========================================================}
grammar simproc-in is
import grammar simproc-base ;

root is PROGRAM ;

{Attributs}
{=========}
$text (NUMBER) : token ;
$id (ID) : token ;
end grammar ;
```

L'arbre abstrait et la syntaxe absraite sont très dépendants de la syntaxe concrète du langage. L'expérience prouve qu'il est très commode de décrire le fichier de syntaxe abstraite en parallèle avec le fichier de syntaxe concrète (« `simproc.atc` »).

Nous donnons ci-dessous, la syntaxe concrète contenue dans le fichier « `simproc.atc` ».

```
import grammar simproc-in ;

{Programme et blocs}
```

```
{==================}
<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>)

{Instructions}
{============}
<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 opérateurs}
{=========================}
<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>)

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

La syntaxe concrète décrite dans le fichier « `simproc.atc` » est non ambiguë, c'est pourquoi le fichier permettant de résoudre les ambiguïtés, « `simproc.prio` », est vide.

A ce stade, nous sommes en présence de six fichiers de spécification :

- « `simproc-base.asx` » ;
- « `simproc-in.asx` » ;
- « `simproc.atc` » ;
- « `simproc.lecl` » ;
- « `simproc.prio` » (vide) ;
- « `simproc.recor` » (fichier standard) ;

Nous pouvons soumettre ces spécifications aux constructeurs concernés. Pour cela, on appellera de nouveau « `FNC2` ».

En choisissant l'option « `D ebmkfnc2` », on permet au système de générer un ensemble de fichiers « `.aux` ». Ces fichiers contiennent les règles de dépendance des fichiers de spécification qui se trouvent dans le répertoire « `spec` ». Ils permettent d'appeler les processeurs dans le bon ordre.

```
-------------------------
 debmkfnc2 ? reply (n|y|e|m|help)
-------------------------
y
ASX
Release 1.15 of (97/08/18 10:35:24)
/local_home/users/vivien/simproc/spec/simproc-base.asx:
   Scanner & Parser & Abstract Tree

        Listing Output
Release 1.15 of (97/08/18 10:35:24)
/local_home/users/vivien/simproc/spec/simproc-in.asx:
   Scanner & Parser & Abstract Tree

        Listing Output
OLGA
PPAT
ATC
Release 1.15 of (97/08/18 10:35:24)
/local_home/users/vivien/simproc/spec/simproc.atc:
   Scanner & Parser & Abstract Tree
```

En choisissant l'option « `K akemkfnc2` », on permet au système de créer le fichier « `simproc.MKFNC2` » en fonction des fichiers « `.aux` ». Ce fichier permet l'appel des divers constructeurs.

```
----------------------------
 makemkfnc2 ? reply (n|y|e|m|help)
----------------------------
y
simproc.MKFNC2:
Pretty Printer char count:   237
---------------
 Your simproc.MKFNC2
---------------
application simproc is

ASX simproc-base simproc-base import
```

```
end import ;

ASX simproc-in simproc-in import

     ASX simproc-base ?
end import ;

ATC simproc simproc import

     ASX_FUNC simproc-in ?
end import ;

end application ;
```

En choisissant l'option « `C onstruction` », on déclenche l'appel des constructeurs sur l'ensemble des spécifications dans le répertoire « `spec` ». Le menu boucle sur cette option, ce qui permet à l'utilisateur de corriger les spécifications et relancer les constructions, jusqu'à ce qu'elles soient correctes.

```
-----------------------------------
 Construction ? reply (n|m|e|y|s|a|help)
-----------------------------------
y
```

A.3 Ecriture des grammaires attribuées pour l'analyse sémantique

Lorsque les erreurs signalées par les constructeurs de la partie lexico-syntaxique ont été corrigées, on peut commencer à écrire les diverses grammaires attribuées pour l'analyse sémantique.

La grammaire attribuée concernant les vérifications sémantiques sur le langage SIMPROC est contenue dans le fichier « `ag_simproc-check.olga` ». Nous trouvons sa description ci-dessous.

```
attribute grammar simproc-check (simproc-in in PROGRAM;
                                 simproc-out out PROGRAM) : ($correct:bool) is

from simproc-types import all ;

attribute

{Attributs de travail}
{====================}
```

```
    compound $temp-st (BLOCK, DECLS, DECL) : symbol-table ;
    synthesized $correct1 (BLOCK, DECLS, DECL) : bool ;
    synthesized $correct2 (BLOCK, DECLS, DECL, STMT,
                           STMTS, EXPR, VAR, NUMBER) : bool ;
{Attributs exportés}
    synthesized $value () ;
    synthesized $op () ;
    global $symtab () { symbol-table} ;
    synthesized $correct () ;

{Les règles sémantiques}
{=====================}

{Programme et bloc}
{=================}
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 STMTS use
    $correct1 := $correct1 (DECLS) ;
    $correct2 := $correct2 (DECLS) &
                 $correct2 (STMTS) ;
    $s.temp-st := $s.temp-st (DECLS) ;
end where ;

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

{Declarations}
{============}
where decls -> DECL* use
    $correct1 := map left &
                     value true
                     other $correct1 (DECL)
                 end map ;
```

```
    $correct2 := map left &
                     value true
                     other $correct2 (DECL)
                 end map ;
    $h.temp-st (DECL) :=
                         {cette definition sert a donner un exemple pour
                           la construction «case position»; Elle aurait
                           pu etre generee automatiquement par les regles
                           par defaut}
                         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 : bool := if  $id (ID) = error-token then
                          false
                      elsif lookup-in-block ($id (ID),
                                             $h.temp-st) then
                          message "Identifier already declared."
                          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 : bool := (if  $id (ID) = error-token then
                           false
                       elsif lookup-in-block ($id (ID),
                                             $h.temp-st) then
                           message "Identifier already declared."
                           position ID
                           severity 2
                           value false
                       else
                           true
                       end if)
                       &
                      ($correct2 (NUMBER) and
                       if  int (string ($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 : bool := (if  $id (ID1) = error-token then
                           false
                       elsif lookup-in-block ($id (ID1),
                                             $h.temp-st) then
                           message "Identifier already declared."
```

```
                           position ID.1
                           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 "Formal parameters may not have the same name."
                           position ID.3
                           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 STMTS1 STMTS2 use
    $correct2 := $correct2 (EXPR1) &
                 $correct2 (EXPR2) &
                 $correct2 (STMTS1) &
                 $correct2 (STMTS2) ;
end where ;

where call -> ID EXPR VAR declare
value
    mode : mode := lookup ($id (ID),
                           $symtab [BLOCK]) ;
use
    $correct2 := (if  $id (ID) = error-token then
                      false
                  else
                      if  mode = error-mode then
                          message "Undeclared identifier."
                          position ID
                          severity 2
                          value false
                      elsif mode != procid then
                          message "Not a procedure."
                          position ID
                          severity 2
                          value false
                      else
                          true
                      end if
                  end if)
```

```
                  &
                 $correct2 (EXPR) &
                 $correct2 (VAR) ;
end where ;

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

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

{Expressions et operateurs}
{=======================}
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 used-var -> 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 ;

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

where div -> use
```

```
    $op := division ;
end where ;

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


{Variables}
{=========}
where null-VAR -> use
    $correct2 := false ;
end where ;

where simple-var -> ID use
    $correct2 := if  $id (ID) = error-token then
                     false
                 else
                     let
                         mode : mode := lookup ($id (ID),
                                                $symtab [BLOCK])
                     in
                     if  mode = error-mode then
                         message "Undeclared identifier."
                         position ID
                         severity 2
                         value false
                     elsif (mode = varid)
                            or
                           (mode = refid)
                            then
                         true
                     else
                         message "Not a variable."
                         position ID
                         severity 2
                         value false
                     end if
                 end if ;
end where ;

where indexed-var -> ID EXPR use
    $correct2 := (if  $id (ID) = error-token then
                      false
```

```
                  else
                      let
                          mode : mode := lookup ($id (ID),
                                                 $symtab [BLOCK])
                      in
                      if  mode = error-mode then
                          message "Undeclared identifier."
                          position ID
                          severity 2
                          value false
                      elsif mode != arrayid then
                          message "Not an array."
                          position ID
                          severity 2
                          value false
                      else
                          true
                      end if
                  end if)
                  &
                 $correct2 (EXPR) ;
end where ;


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

where null-NUMBER -> use
    $correct2 := false ;
    $value := token (string (0)) ;
end where ;

where id -> use
end where ;

where null-ID -> use
```

```
end where ;

end grammar ;
```

Le module de déclaration de la grammaire précédente, se trouvre dans le fichier « `dec_simproc-check.olga` ». Il est donné ci-dessous :

```
declaration module simproc-check is

import grammar simproc-in;

function simproc-check (r:PROGRAM; r1:PROGRAM) : bool;

end module;
```

La grammaire « `simproc-check` » importe un module qui implémente une table de symboles, avec les fonctions de manipulation de cette table. Nous donnons ci-dessous le contenu de ce module (fichier « `def_simproc-types.olga` ») ainsi que le contenu de son module de déclaration (fichier « `dec_simproc-types.olga` »).

```
definition module simproc-types is

{Implementation d'une table de symboles utilisant une liste lineaire}
{===================================================================}

const
    empty-table := symbol-table (block-element ()) ;
    offset-undefined := 0 ;
    error-element := element (error-token,
                              error-mode,
                              offset-undefined) ;

function lookup-in-block-mode (key : token ;
                               st : block-element) : mode is
    if  empty (st) then
        error-mode
    else
        let
            element : element := head (st)
        in
        if  (element.key = key)
             then
            element.info
        else
            lookup-in-block-mode (key,
                                  tail (st))
```

```
        end if
    end if
end function { lookup }  ;

function lookup (key : token ;
                 st : symbol-table) : mode is
    if  empty (st) then
        error-mode
    else
        let
            mode : mode := lookup-in-block-mode (key,
                                                 head (st))
        in
        if  (mode = error-mode)
             then
            lookup (key,
                    tail (st))
        else
            mode
        end if
    end if
end function { lookup }  ;

function lookup-in-block (key : token ;
                          st : symbol-table) : bool is
    if  (lookup-in-block-mode (key,
                               head (st)) != error-mode)
         then
        true
    else
        false
    end if
end function { lookup-in-block }  ;

function insert (key : token ;
                 info : mode ;
                 st : symbol-table) : symbol-table is
    symbol-table (block-element (element (key,
                                          info,
                                          offset-undefined),
                                 head (st)),
                  st)
end function { insert }  ;
```

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

end module ;
```

Le module de déclaration associé est donné ci-dessous.

```
declaration module simproc-types is

type
    mode = (varid,
            refid,
            arrayid,
            procid,
            error-mode) ;
    operator = (wrong-op,
                addition,
                subtraction,
                multiplication,
                division) ;
    element = record
                  key : token ;
                  info : mode ;
                  offset : int ;
              end record ;
    block-element = list of element ;
    symbol-table = list of block-element ;

const
    empty-table : symbol-table ;
    error-element : element ;
    offset-undefined : int ;

function lookup-in-block (id : token ;
                          st : symbol-table) : bool ;

function lookup (id : token ;
                 st : symbol-table) : mode ;

function insert (id : token ;
                 info : mode ;
                 st : symbol-table) : symbol-table ;
```

```
function enter-block (st : symbol-table) : symbol-table ;

end module ;
```

La grammaire attribuée « `simproc-check` » est une grammaire attribuée procédurale (elle redécore l'arbre d'entrée sans créer de nouvel arbre). On peut voir dans l'en-tête de la grammaire que l'arbre redécoré est décrit par la syntaxe attribuée abstraite « `simproc-out` », contenue dans le fichier « `simproc-out.asx` ». Cette syntaxe est donnée ci-dessous :

```
grammar simproc-out is

import grammar simproc-base ;
from simproc-types import  all {Table des symboles et type operator} ;

root is PROGRAM ;

{Attributs}
{=========}
$symtab (BLOCK) : symbol-table ;
$op (OP) : operator ;
$value (NUMBER) : token { remplace $text }  ;
$id (ID) : token { le seul apparaissant aussi dans "simproc-in" }  ;
$correct (PROGRAM) : bool ;

end grammar ;
```

Nous sommes arrivés à la fin de la description de la phase de sémantique statique. Nous pouvons vérifier nos spécifications en relançant le script « `FNC2` » et en exécutant les options « `D` », « `K` » et « `C` ».

A.4 Ecriture des grammaires attribuées pour la production de code cible

Pour la production de code intermédiaire, nous avons besoin de décrire :

- un arbre abstrait adapté à la structure du code cible à produire ;
- une grammaire attribuée fonctionnelle construisant cet arbre à partir de l'arbre issu de la phase de sémantique statique ;
- une spécification de décompilation d'arbre (avec le langage PPAT), pour générer le code cible.

Les trois descriptions précédentes se trouvent respectivement dans les fichiers « `simproc-adt.asx` », « `ag_simproc-term.olga` » (avec son fichier de déclaration « `dec_simproc-term.olga` ») et « `simproc-ppat.ppat` ».

Nous donnons la description de ces fichiers ci-dessous.

```
grammar simproc-adt is

from simproc-types import  all {Table de symboles} ;

root is Program ;

Program = Execute ;
Execute -> Procid Block ;

Block              = Concat ;
Concat            -> Decl-source-modifs Stmt-source-modifs ;

Decl-source-modifs = Decl-concat ;
Decl-concat       -> Decl-source-modif* ;

Decl-source-modif  = Procdecl Intdecl Arraydecl ;

Stmt-source-modifs = Stmt-concat ;
Stmt-concat       -> Stmt-source-modif* ;

Stmt-source-modif  = Assignint Assignvar Condit Proccall ;
Assignint         -> Var Int ;
Assignvar         -> Refvar Int ;
Condit            -> Int Int Stmt-source-modifs Stmt-source-modifs ;
Proccall          -> Procid Int Var Procid ;

Procdecl          -> Procid Varid Refid Procid Block ;
Intdecl           -> Varid Procid ;
Arraydecl         -> Arrayid Int Procid ;

Call               = Currentcall ;
Currentcall       -> ;

{Identificateurs}
{===============}
Varid              = VarId ;
VarId             -> ;
Refid              = RefId ;
RefId             -> ;
Arrayid            = ArrayId ;
ArrayId           -> ;
Procid             = ProcId ;
```

```
ProcId           -> ;

{Variables}
{=========}
Var               = Designates Refersto Element ;
Designates       -> Varid Call ;
Refersto         -> Refvar ;
Element          -> Arrayid Call Int ;
Refvar            = Refdesignates ;
Refdesignates    -> Refid Call ;

{Entiers}
{=======}
Int               = Plus Difference Times Quotient Valueof Const ;
Plus             -> Int Int ;
Difference       -> Int Int ;
Times            -> Int Int ;
Quotient         -> Int Int ;
Valueof          -> Var ;
Const            -> ;

{Attributs attachés à l'arbre de sortie}
{======================================}
$Value-att (Int) : token ;
$Id (Varid, Refid, Arrayid, Procid, Var) : token ;
$block-table(Block) : symbol-table ;
end grammar ;
attribute grammar simproc-term (simproc-out in PROGRAM)
                                            : ($Program: Program) is

from simproc-types import all ;

const
    Prog := "Prog" ;

attribute


{Pour la construction de l'arbre en sorti}
{========================================}
    synthesized $Block-model (BLOCK) : Block ;
    synthesized $Decl-models (DECLS) : Decl-source-modifs ;
    synthesized $Stmt-models (STMTS) : Stmt-source-modifs ;
```

```
    synthesized $Decl-model (DECL) : Decl-source-modif ;
    synthesized $Stmt-model (STMT) : Stmt-source-modif ;
    synthesized $Address (VAR) : Var ;
    synthesized $Value (EXPR) : Int ;
    inherited $procid (BLOCK) : token { name of enclosing proc }  ;
    global $symbtab-term (BLOCK) : symbol-table ;


{Les regles sémantiques}
{=====================}

{Programme et blocs}
{=================}

where program -> BLOCK use
    $Program := Execute (ProcId ()
                         with $Id := token (Prog)
                         end with,
                         $Block-model (BLOCK)) ;
    $procid (BLOCK) := token (Prog) ;
    $symbtab-term (BLOCK) := $symtab (BLOCK) ;
end where ;

where null-PROGRAM -> use
    $Program := null-Program () ;
end where ;

where block -> DECLS STMTS use
    $Block-model := Concat ($Decl-models (DECLS),
                            $Stmt-models (STMTS))
                    with $block-table := $symbtab-term (block)
                    end with ;
end where ;

where null-BLOCK -> use
    $Block-model := null-Block ()
                    with $block-table := empty-table
                    end with ;
end where ;


{Déclarations}
{===========}
where decls -> DECL* use
```

```
    $Decl-models := map left Decl-concat-post
                        value Decl-concat ()
                        other $Decl-model (DECL)
                    end map ;
end where ;

where null-DECLS -> use
    $Decl-models := null-Decl-source-modifs () ;
end where ;

where null-STMTS -> use
    $Stmt-models := null-Stmt-source-modifs () ;
end where ;

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

where array-decl -> ID NUMBER use
    $Decl-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
    $symbtab-term (BLOCK) := $symtab (BLOCK) ;
    $Decl-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,
                             $Block-model (BLOCK)) ;
    $procid (BLOCK) := $id (ID1) ;
end where ;

where null-DECL -> use
    $Decl-model := null-Decl-source-modif () ;
end where ;


{Instructions}
{============}
where stmt-list -> STMT+ use
    $Stmt-models := map left Stmt-concat-post
                        value Stmt-concat ()
                        other $Stmt-model (STMT)
                    end map ;
end where ;

where assign -> VAR EXPR use
    $Stmt-model := case $Address (VAR) is
                       Refersto (Refdesignates (X,Y)) : Assignint ($Address (VAR),
                                                        $Value (EXPR)) ;
                       other :
                           Assignint ($Address (VAR),
                                      $Value (EXPR)) ;
                   end case ;
end where ;

where ifthenelse -> EXPR1 EXPR2 STMTS1 STMTS2 use
    $Stmt-model := Condit ($Value (EXPR1),
                           $Value (EXPR2),
                           $Stmt-models (STMTS1),
                           $Stmt-models (STMTS2)) ;
end where ;

where call -> ID EXPR VAR use
    $Stmt-model := Proccall (ProcId ()
                             with $Id := $id (ID)
                             end with,
                             $Value (EXPR),
                             $Address (VAR),
```

```
                             ProcId ()
                             with $Id := $procid [BLOCK]
                             end with) ;
end where ;

where null-STMT -> use
    $Stmt-model := null-Stmt-source-modif () ;
end where ;


{Expressions et operateurs}
{========================}
where bin-expr -> OP EXPR1 EXPR2 use
    $Value := case $op (OP) is
                  addition  :
                      Plus ($Value (EXPR1),
                            $Value (EXPR2))
                      with $Value-att := token (string (0))
                      end with ;
                  subtraction :
                      Difference ($Value (EXPR1),
                                  $Value (EXPR2))
                      with $Value-att := token (string (0))
                      end with ;
                  multiplication :
                      Times ($Value (EXPR1),
                             $Value (EXPR2))
                      with $Value-att := token (string (0))
                      end with ;
                  division :
                      Quotient ($Value (EXPR1),
                                $Value (EXPR2))
                      with $Value-att := token (string (0))
                      end with ;
                  other :
                      null-Int ()
                      with $Value-att := token (string (0))
                      end with ;
              end case ;
end where ;

where used-var -> VAR use
    $Value := Valueof ($Address (VAR))
              with $Value-att := token (string (0))
```

```
              end with ;
end where ;

where null-EXPR -> use
    $Value := Const ()
              with $Value-att := token (string (0))
              end with ;
end where ;



{Variables}
{=========}
where null-VAR -> use
    $Address := null-Var ()
                with $Id := error-token
                end with ;
end where ;

where simple-var -> ID use
    $Address := let
                    id : token := $id (ID)
                in
                case lookup (id,
                             $symbtab-term [BLOCK]) is
                    refid :
                        Refersto (Refdesignates (RefId ()
                                                 with $Id := id
                                                 end with,
                                                 Currentcall ()))
                        with $Id := id
                        end with ;
                    other :
                        Designates (VarId ()
                                    with $Id := id
                                    end with,
                                    Currentcall ())
                        with $Id := id
                        end with ;
                end case ;
end where ;

where indexed-var -> ID EXPR use
    $Address := Element (ArrayId ()
```

```
                        with $Id := $id (ID)
                        end with,
                        Currentcall (),
                        $Value (EXPR))
               with $Id := $id (ID)
               end with ;
end where ;


{Nombres}
{=======}
where number -> use
    $Value := Const ()
              with $Value-att := $value
              end with ;
end where ;

where null-NUMBER -> use
    $Value := null-Int ()
              with $Value-att := token (string (0))
              end with ;
end where ;

end grammar ;
```

Le module de déclaration correspondant à la grammaire précédente est donné ci-dessous.

```
declaration module simproc-term is

from simproc-types import all ;
import grammar simproc-in;
import grammar simproc-adt;

function simproc-term (r:PROGRAM) : Program;

end module;
```

Enfin, la dernière spécification concerne la décompilation de l'arbre attribué créé par la grammaire précédente. Elle est listée ci-après.

```
ppat simproc-ppat ( simproc-adt ) ;

from simproc-types import all ;

separator
  <h1> = <h 1> ;
```

```
  <h0> = <h 0> ;
  <v00> = <v 0 , 0> ;
  <v01> = <v 0 , 1> ;
  <v04> = <v 0 , 4> ;
  <v40> = <v 4 , 0> ;
  <v20> = <v 2 , 0> ;
  <hov100> = <hov 1 , 0 , 0> ;

{-  ---------------------------------- -}
{- Programme -}
{-  ----------------------------------- -}

where null-Program -> pprint

end where ;

where Execute -> Procid Block  pprint
        [v00 "Execute" Procid Block]
end where ;
{-  ----------------------------------- -}
{- Bloc -}
{-  ----------------------------------- -}

where null-Block -> pprint

end where ;
where Concat -> Decl-source-modifs Stmt-source-modifs  pprint
        [v40 "Decl" Decl-source-modifs "Stmt" Stmt-source-modifs]
end where ;
{-  ----------------------------------- -}
{- Decl-source-modifs -}
{-  ----------------------------------- -}

where null-Decl-source-modifs -> pprint

end where ;
where Decl-concat -> Decl-source-modif * pprint
        pplist left
                global v00
                subbox Decl-source-modif
        end pplist
end where ;
{-  ----------------------------------- -}
{- Decl-source-modif -}
```

```
{-  ---------------------------------- -}

where null-Decl-source-modif -> pprint

end where ;
{-  ---------------------------------- -}
{- Stmt-source-modifs -}
{-  ---------------------------------- -}

where null-Stmt-source-modifs -> pprint

end where ;
where Stmt-concat -> Stmt-source-modif  * pprint
        pplist left
                global v00
                subbox Stmt-source-modif
        end pplist
end where ;

{-  ---------------------------------- -}
{- Stmt-source-modif -}
{-  ---------------------------------- -}

where null-Stmt-source-modif -> pprint

end where ;
where Assignint -> Var Int  pprint
        [h1 "Assignint" Var Int]
end where ;
where Assignvar -> Refvar Int  pprint
        [h1 "Assignvar" Refvar Int]
end where ;
where Condit -> Int1 Int2 Stmt-source-modifs1 Stmt-source-modifs2  pprint
        [v00    [h1 "Condit" Int1 Int2]
                [h1 "Then" Stmt-source-modifs1]
                [h1 "Else" Stmt-source-modifs2]
        ]
end where ;
where Proccall -> Procid1 Int Var Procid2  pprint
        [h1 "Call" Procid1 Int Var Procid2]
end where ;
where Procdecl -> Procid1 Varid Refid Procid2 Block  pprint
        [v00    [h1 "Procdecl" Procid1 Varid Refid Procid2]
                Block
```

```
                "EndProddecl"
        ]
end where ;
where Intdecl -> Varid Procid  pprint
        [h1 "Intdecl" Varid Procid]
end where ;
where Arraydecl -> Arrayid Int Procid  pprint
        [h1 "Arraydecl" Arrayid Int Procid]
end where ;
{-  ---------------------------------- -}
{- Call -}
{-  ---------------------------------- -}

where null-Call -> pprint

end where ;
where Currentcall ->   pprint

end where ;
{-  ---------------------------------- -}
{- Varid -}
{-  ---------------------------------- -}

where null-Varid -> pprint

end where ;
where VarId ->   pprint
        $Id
end where ;
{-  ---------------------------------- -}
{- Refid -}
{-  ---------------------------------- -}

where null-Refid -> pprint

end where ;
where RefId ->   pprint
        $Id
end where ;
{-  ---------------------------------- -}
{- Arrayid -}
{-  ---------------------------------- -}

where null-Arrayid -> pprint
```

```
end where ;
where ArrayId ->   pprint
        $Id
end where ;
{-  ---------------------------------- -}
{- Procid -}
{-  ---------------------------------- -}

where null-Procid -> pprint

end where ;
where ProcId ->   pprint
        $Id
end where ;
{-  ---------------------------------- -}
{- Var -}
{-  ---------------------------------- -}

where null-Var -> pprint

end where ;
where Designates -> Varid Call  pprint
        [h1 "Designates" Varid Call]
end where ;
where Refersto -> Refvar  pprint
        [h1 "Refersto" Refvar]
end where ;
where Element -> Arrayid Call Int  pprint
        [h1 "Element" Arrayid Call Int]
end where ;
{-  ---------------------------------- -}
{- Refvar -}
{-  ---------------------------------- -}

where null-Refvar -> pprint

end where ;
where Refdesignates -> Refid Call  pprint
        [h1 "Refdesignates" Refid Call]
end where ;
{-  ---------------------------------- -}
{- Int -}
{-  ---------------------------------- -}
```

```
where null-Int -> pprint

end where ;
where Plus -> Int1 Int2  pprint
        [h1 "Plus" Int1 Int2]
end where ;
where Difference -> Int1 Int2  pprint
        [h1 "Difference" Int1 Int2]
end where ;
where Times -> Int1 Int2  pprint
        [h1 "Times" Int1 Int2]
end where ;
where Quotient -> Int1 Int2  pprint
        [h1 "Quotient" Int1 Int2]
end where ;
where Valueof -> Var  pprint
        Var
end where ;
where Const ->   pprint
        $Value-att
end where ;


end ppat ;
```

Comme nous l'avons fait à la fin des descriptions de sémantique statique, nous pouvons soumettre nos spécifications au système SYNTAX/FNC-2.

A.5 Appel des grammaires attribuées

La description du compilateur SIMPROC est terminée ; il nous reste à écrire l'appel des grammaires attribuées et de la spécification PPAT dans le fichier « `simproc_smp.c` » produit par « `simproc.mkfnc` ».

```
#include "simproc.h"    /* Pour inclure éventuellement nos applications */

/* La passe sémantique et l'appel du décompilateur d'arbre */
/* ====================================================== */
static VOID simprocsmp ()

{
    /* Pointeurs sur les résultats de nos grammaires attribuées */
    /* ========================================================= */
```

```
    struct simproc__check_result *r;
    struct simproc__term_result *r1;

    FILE *sortie, *arbre;
    char lst_name [132];

    /* Cette fonction est definie dans simproc.c
       elle est produite avec  mkfnc2 avec le fichier simproc.mkfnc2 */
    /* ============================================================ */
    init___simproc ();


    /* C'est ici que prend place l'appel de nos grammaires attribuées dans */
    /* =================================================================*/

    /* 1) La grammaire attribuée procédurale de sémantique statique  */
    /*    - simproc__check est le nom de notre premiere grammaire attribue */
    /*      (simproc-check.olga) */
    /*    - f2atcvar.atc_lv.abstract_tree_root est la racine de l'arbre */
    /*      abstrait issu de l'analyse lexico-syntaxique (SYNTAX) */
    r = simproc__check (f2atcvar.atc_lv.abstract_tree_root);

    /* 2) Si la passe sémantique est correcte, on construit l'arbre abstrait */
    /*    pour la génération du code cible. */
    /*    Dans cette phase, la grammaire attribuée fonctionnelle  */
    /*    simproc-check.olga, est utilisée pour construire l'arbre abstrait */
    /*    décrit par la syntaxe attribuée abstraite simproc-adt.asx. */
    /*    Finalement, on appelle le décompilateur ppat crée a partir de la */
    /*    spécification simproc-ppat.ppat. r1->Program est la racine de */
    /*    l'arbre. La chaîne vide "" indique que la décompilation se fera  */
    /*    sur la sortie standard. Une chaîne non-vide indiquerait un nom de  */
    /*    fichier pour contenir le résultat de le décompilation. 80 indique  */
    /*    la longueur maximale des lignes avant application des règles de  */
    /*    césure (voir la documentation de ppat) */
    if (r -> correct) {
        r1 = simproc__term (f2atcvar.atc_lv.abstract_tree_root);
        ppat___simproc__ppat(r1->Program,80,"");
    }
}

/* Dans le cas de l'exemple simproc, le code ci-dessous n'a pas */
/* a etre modifie par l'utilisateur */
/* ============================================================ */
```

```
static VOID simprocfinal ()
{
    FILE        *sortie;
    char        lst_name [132];

}

static simprocopen (tables)
    register struct sxtables *tables;
{
    f2atcvar.atc_lv.node_size = sizeof (NODE) /* size of the olga_node */;
}

VOID simproc_smp (entry,arg)
int  entry;
struct sxtables *arg;
{

    switch (entry) {
    case OPEN:
        simprocopen (arg);
        break;

    case CLOSE:
        break;

    case ACTION:
    case SEMPASS:
        simprocsmp ();
        break;

    case INIT:
        break;

    case FINAL:
        simprocfinal ();
        break;

    default:
        fprintf (sxstderr, "The function \"simprocsmp\"
                 is out of date with respect to its specification.\n");
        break;
    }
    return;
```

```
    /* end simprocsmp */
}
```

Enfin, nous pouvons réaliser l'étape finale avec le script « `FNC2` ». Nous choisirons l'option « `F inal` », ce qui aura pour effet de compiler l'ensemble des fichiers C produits par Syntax/Fnc-2.

Le résultat de la compilation (le compilateur Simproc) se trouve dans le répertoire « `$HOME/simproc/bin/solaris/simproc` » (dans le cas d'une utilisation sous machine Solaris).

Il ne reste plus qu'à utiliser le compilateur en lui soumettant un programme de test, comme celui-ci par exemple :

```
integer J;

proc factorial (val I , var F )
    begin
    integer TEMP;
    if I eq 0 then F := 1
              else  call factorial(I-1, TEMP);
                    F := TEMP * I
    fi
end;

call factorial (5, J)
```

En appelant notre compilateur Simproc avec comme paramètre le nom d'un fichier contenant le texte ci-dessus, on verra la génération de code cible sur la sortie standard :

```
Execute
Prog
    Decl
    Intdecl J Prog
    Procdecl factorial I F Prog
        Decl
        Intdecl TEMP factorial
        Stmt
        Condit Designates I 0
        Then Assignint Refersto Refdesignates F 1
        Else Call factorial Difference Designates I 1 Designates TEMP factorial
             Assignint Refersto Refdesignates F Times Designates TEMP Designates I
    EndProddecl
    Stmt
    Call factorial 5 Designates J Prog
```

Appendix B

Schemas de traduction E-LOTOS vers LOTOS

Dans la suite de ce document, $Trad_{lotos}$ désigne une fonction traduisant des éléments de spécification E-Lotos en éléments de spécification Lotos.

B.1 Traduction d'une spécification E-LOTOS

Spécification E-Lotos	Spécification Lotos

specification Spec **is**
`<déclarations de`
`types et de processus>*`
 	
behaviour
`<B>`
endspec
	specification Spec **: noexit**
	library Boolean, Natural endlib
	$Trad_{lotos}$(`<déclarations de types>`)
	behaviour
	$Trad_{lotos}$(`<B>`)
	where
	$Trad_{lotos}$(`<déclarations de processus>`)
	endspec

L'ensemble des déclarations de type E-Lotos fera l'objet d'un seul type Lotos.

B.2 Traduction d'une déclaration de type

type NomDeSpécificationE-LOTOS **is Boolean Natural**
  sorts
    <déclarations de sortes>
  opns
    <déclarations de constructeurs>
    <déclarations de fonctions>
  eqns
    <équations>
endtype

La spécification LOTOS produite utilise les conventions et les extensions particulières aux compilateurs CÆSAR et CÆSAR.ADT.

Le tableau ci-dessous présente la traduction des types E-LOTOS vers la clause « `sort` » du langage LOTOS.

Déclaration E-LOTOS	Traduction en clause **sorts** LOTOS
type S **renames** S' **endtype** où **S** est un type externe	S (*! **implementedby** S **external** *)
type S **renames** S' **endtype** où **S** n'est pas un type externe	S
type S **is external endtype**	S (*! **implementedby** S **external** *)
type S **is** $\mathrm{C}_1(\mathrm{V}_1{:}\mathrm{S}_1, \cdots, \mathrm{V}_n{:}\mathrm{S}_n)_1$ | $\vdots$ | $\mathrm{C}_m(\mathrm{V}_1{:}\mathrm{S}_1, \cdots, \mathrm{V}_n{:}\mathrm{S}_n)_m$ **endtype**	S

Le tableau ci-dessous présente la traduction des types E-LOTOS vers la clause « `opns` » du langage LOTOS.

Déclaration E-LOTOS	Traduction en clause **opns** LOTOS
type S **renames** S' **endtype** si S' est un type external	*Rien à traduire*
type S **renames** S' **endtype** si S' n'est pas un type external	C_1`(*! constructor *) :` $\mathrm{S}_{1_1}, \cdots, \mathrm{S}_{n_1}$ `->` S $\vdots$ C_m`(*! constructor *) :` $\mathrm{S}_{1_m}, \cdots, \mathrm{S}_{n_m}$ `->` S On recopie les constructeurs de S' dans lesquels on substitue S à S'.
type S **is external endtype**	*Rien à traduire*
type S **is** $\mathrm{C}_1(\mathrm{V}_1{:}\mathrm{S}_1, \cdots, \mathrm{V}_n{:}\mathrm{S}_n)_1$ | $\vdots$ | $\mathrm{C}_m(\mathrm{V}_1{:}\mathrm{S}_1, \cdots, \mathrm{V}_n{:}\mathrm{S}_n)_m$ **endtype**	C_1`(*! constructor *) :` $\mathrm{S}_{1_1}, \cdots, \mathrm{S}_{n_1}$ `->` S $\vdots$ C_m`(*! constructor *) :` $\mathrm{S}_{1_m}, \cdots, \mathrm{S}_{n_m}$ `->` S

B.3 Traduction d'une déclaration de processus

Déclaration E-LOTOS	Traduction en clause **where** LOTOS
process Π [**[**[G:S (, G:S)*]**]**] [**(**[**in**]V^{in}:S^{in} (, [**in**]V^{in}:S^{in})***)**] **:noexit** [**raises** **[**[X:S (, X:S)*]**]**] **is** <B> **endproc**	**process** Π [**[**G (, G)***]**] [**(**V:S (, V:S)***)**] **:noexit** *Impossible* := $Trad_{lotos}$(<B>) **endproc**
process Π [**[**[G:S (, G:S)*]**]**] **(**[**in**]V^{in}:S^{in}| **out** V^{out}:S^{out} (, [**in**]V^{in}:S^{in}| **out** V^{out}:S^{out})***)** Au moins un paramètre « out » doit-être présent dans cette liste, dans le cas contraire nous sommes en présence d'un «noexit-process» [**raises** **[**[X:S (, X:S)*]**]**] **is** <B> **endproc**	**process** Π [**[**G (, G)***]**] [**(**V^{in}:S^{in} (, V^{in}:S^{in})***)**] **:exit** (S^{out} (, S^{out})*) *Impossible* := $Trad_{lotos}$(<B>) **endproc**

B.4 Traduction d'une déclaration de fonction

Déclaration E-LOTOS	Traduction en clause **opns** LOTOS
function F [(**[in**]V^{in}:S^{in} (, [**in**]V^{in}:S^{in})*)]: S [**raises [**[X:S (, X:S)*]**]**] **is** <E> **endfunc**	F : S (, S)* -> S
function F ([**in**]V^{in}:S^{in}| **out** V^{out}:S^{out} (, [**in**]V^{in}:S^{in}| **out** V^{out}:S^{out})*) [**raises [**[X:S (, X:S)*]**]**] **is** <E> **endfunc**	F : S^{in} (, S^{in})* -> OUT_F OUT_F (∗!*constructor*∗) : S^{out} (, S^{out})* -> OUT_F

Déclaration E-LOTOS	Traduction en clause **sorts** LOTOS
function F ([**in**]V^{in}:S^{in}| **out** V^{out}:S^{out} (, [**in**]V^{in}:S^{in}| **out** V^{out}:S^{out})*) [**raises [**[X:S (, X:S)*]**]**] **is** <E> **endfunc**	OUT_F

<table>
<tr><th>Déclaration E-Lotos</th><th>Traduction en clause eqns Lotos</th></tr>
<tr><td>function F [([in]V^{in}:S^{in} (, [in]V^{in}:S^{in})*)]: S
[raises [[X:S (, X:S)*]]]
is
<E>
endfunc</td><td>forall V^{in}:S^{in} (, V^{in}:S^{in})* ofsort S
Union disjointe des variables « in »</td></tr>
<tr><td>function F
([in]V^{in}:S^{in}| out V^{out}:S^{out} (, [in]V^{in}:S^{in}| out V^{out}:S^{out})*)
[raises [[X:S (, X:S)*]]]
is
<E>
endfunc</td><td>forall V^{in}:S^{in} (, V^{in}:S^{in})* ofsort S
Union disjointe des variables « in »
ofsort OUT_F
F (V^{in} (, V^{in})*) = $Trad_{lotos}$(<E>) ;</td></tr>
</table>

B.5 Traduction des filtres

Filtre E-Lotos	Traduction en Lotos
any	*Impossible*
? V	V:S_V S_V est le type de la variable V
? V **as** <P>	*Impossible*
! <VE>	$Trad_{lotos}$(<VE>)
? C (<RP>)	C $Trad_{lotos}$(<RP>)
<P> **of** S	$Trad_{lotos}$(<P>) **of** S

record–pattern E-LOTOS	Traduction en LOTOS
V => <P> (, V => <P>)*	*Pas encore implémenté*
...	*Pas encore implémenté*
V => <P> (, V => <P>)*...	*Pas encore implémenté*
<pattern> (, <pattern>)*	$Trad_{lotos}$(<P>) ($Trad_{lotos}$(<P>))*

B.6 Traduction des valeurs

value–expression E-LOTOS	Traduction en LOTOS
K Constante entière	K
V Nom de variable	V
C (<RVE>) <RVE>= record–value–expression	C ($Trad_{lotos}$(<RVE>))

record–value–expression E-LOTOS	Traduction en LOTOS
V => <VE> (, V => <VE>)*	*Pas encore implémenté*
[<VE> (, <VE>)*]	[$Trad_{lotos}$(<VE>) (, $Trad_{lotos}$(<VE>))*]

B.7 Traduction des comportements

Comportement E-Lotos	Traduction en comportement Lotos
G [<P>] [@<P>] [**[**<E>**]**]	G [$Trad_{lotos}$(<P>)] *Impossible* *Impossible*
i	**i ; stop** Si **i** est en partie droite d'une composition sequentielle **i** Si **i** est en partie gauche d'une composition sequentielle
exit (any S)	**exit (any** S)
exit (<RE>)	**exit** (<RE>)
signal X [<RE>]	*Pas encore implémenté*
block	*Impossible*
wait (<E>)	*Impossible*
stop	**stop**
<B>**;** <B>	$Trad_{lotos}$(<B>) **;** $Trad_{lotos}$(<B>)
<B>**[]** <B>	$Trad_{lotos}$(<B>) **[]** $Trad_{lotos}$(<B>)
<B>[> <B>	$Trad_{lotos}$(<B>) [> $Trad_{lotos}$(<B>)

<table>
<tr><th>Comportement E-LOTOS</th><th>Traduction en comportement LOTOS</th></tr>
<tr><td><B>|| <B></td><td>Trad$Trad_{lotos}$(<B>) || $Trad_{lotos}$(<B>)</td></tr>
<tr><td><B>||| <B></td><td>$Trad_{lotos}$(<B>) ||| $Trad_{lotos}$(<B>)</td></tr>
<tr><td><B>|[[G (, G)*]]| <B></td><td>$Trad_{lotos}$(<B>) |[G (, G)*]| $Trad_{lotos}$(<B>)</td></tr>
<tr><td>hide [G:S (, G:S)*]
   in <B>
endhide</td><td>(hide G (, G)*
   in $Trad_{lotos}$(<B>)
)</td></tr>
<tr><td>choice <P> [] <B>endch</td><td>(choice $Trad_{lotos}$(<P>) [] <B>)</td></tr>
<tr><td><P> := <E></td><td>Impossible</td></tr>
<tr><td>par
   [($G\#K_{integer}$ (, $G\#K_{integer}$)*)]
   <B>
endpar</td><td>Pas encore implémenté</td></tr>
<tr><td>trap
   (exception <X>[<P>:S] is <B>)*
   [exit [([in]V^{in}:S^{in} (, [in]V^{in}:S^{in})*)]is <B>]
   in <B>
endtrap</td><td>Pas encore implémenté</td></tr>
<tr><td>if <E> then <B>
   elsif <E>$_1$ then <B>
   ⋮
   elsif <E>$_n$ then <B>
   else <B>
endif</td><td>([$Trad_{lotos}$(<E>)] -> $Trad_{lotos}$(<B>)
   [] [$Trad_{lotos}$(<E>$_1$)] -> $Trad_{lotos}$(<B>)
   ⋮
   [] [$Trad_{lotos}$(<E>$_n$)] -> $Trad_{lotos}$(<B>)
   [] [not (<E>$_1$ or ⋯ or<E>$_n$)] ->
   $Trad_{lotos}$(<B>)
)</td></tr>
</table>

B.8 Traduction des expressions

expression E-Lotos	Traduction en Lotos
`<VE>`	$Trad_{lotos}$(`<VE>`) A condition que `<VE>` soit un identificateur de variable
O (`<IOP>`) [[`<XP>`]] O est un identificateur d'opération	*Pas encore implémenté*
`<E>` = `<E>`	$Trad_{lotos}$(`<E>`) = $Trad_{lotos}$(`<E>`)

Bibliographie

[ASU91] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs, Principes techniques et outils.* InterEditions, 1991.

[BD85] Pierre Boullier and Philippe Deschamp. PARADIS, un système de paragraphage dirigé par la syntaxe. Technical Report, INRIA Rocquencourt (France), Novembr 1985.

[BD88] Pierre Boullier and Philippe Deschamp. Le système SYNTAX. Manuel d'utilisation et de mise en œuvre sous UNIX. Technical Report, INRIA Rocquencourt (France), September 1988.

[CGM+96] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, octobre 1996. Full version available as INRIA Research Report RR-2958.

[CH94] R. Cohen and E. Harry. Automatic generation of near-optimal linear-time translators for non-circular attribute grammars. 6th ACM symp. on principles of Progr. Languages, San Antonio, TX, 121–131, 1994.

[Cho56] N. Chomsky. *Three Models for the Description of Language.* IRE Transactions on Information Theory IT-2, 1956.

[Fan72] Isu Fang. *FOLDS, a Declarative Formal Language Definition System.* PhD thesis, PhD thesis, report STAN-CS-72-329, Comp.Sc.Dept.,Stanford Univ., Décembre 1972. Résumé dans : Séminaires Structure et Programmation des Calculateurs 1973, ed. M. Kronental and Bernard Lorho, INRIA, Rocquencourt, pp. 275-290 (1973).

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and

Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, août 1996.

[FGM+92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A Toolbox for the Verification of LOTOS Programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, mai 1992.

[Gar89a] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), novembre 1989.

[Gar89b] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, décembre 1989.

[Gar90] Hubert Garavel. CÆSAR Reference Manual. Rapport SPECTRE C18, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, novembre 1990.

[Gar94a] Hubert Garavel. On the Introduction of Gate Typing in E-LOTOS. Rapport SPECTRE 94-3, VERIMAG, Grenoble, février 1994. Annex D of ISO/IEC JTC1/SC21/WG1 N1314 Revised Draft on Enhancements to LOTOS and Annex B of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[Gar94b] Hubert Garavel. Six improvements to the process part of LOTOS. Rapport SPECTRE 94-7, VERIMAG, Grenoble, juin 1994. Annex K of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[Gar95a] Hubert Garavel. Contribution to the Design of Data Types in E-LOTOS. Rapport SPECTRE 95-10, VERIMAG, Grenoble, juillet 1995. Input document [OTT7] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Ottawa (Canada), July, 20–26, 1995.

[Gar95b] Hubert Garavel. On the Introduction of Gate Typing in E-LOTOS. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP International Workshop on Protocol Specification, Testing and Verification (Warsaw, Poland)*. IFIP, Chapman & Hall, juin 1995.

[Gar95c] Hubert Garavel. A Wish List for the Behaviour Part of E-LOTOS. Rapport SPECTRE 95-21, VERIMAG, Grenoble, décembre 1995. Input document [LG5] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

[Gar96] Hubert Garavel. An Overview of the Eucalyptus Toolbox. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, pages 76–88. University of Maribor, Slovenia, juin 1996.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. ACM SIGPLAN '84 Symp. on Compiler Construction Montréal, Notices 19, pp. 157–170, June 1984.

[GHL+92] R. W. Gray, V.P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. ELI : A complete, Flexible Compiler Construction System. *Communications of the ACM 35 (February 1992)*, pages 121–131, 1992.

[GJM+97] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. CADP'97 – Status, Applications and Perspectives. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, juin 1997.

[GP90] J-F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, december 1990.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, juin 1990.

[GS95a] Hubert Garavel and Mihaela Sighireanu. Defect Report concerning ISO Standard 8807 and Proposal for a Correct Flattening of LOTOS Parameterized Types. Rapport SPECTRE 95-11, VERIMAG, Grenoble, juillet 1995. Input document [OTT6] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Ottawa (Canada), July, 20–26, 1995.

[GS95b] Hubert Garavel and Mihaela Sighireanu. French-Romanian Comments regarding some Proposed Features for E-LOTOS Data Types. Rapport SPECTRE 95-19, VERIMAG, Grenoble, décembre 1995. Input document [LG3] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

[GS96a] Hubert Garavel and Mihaela Sighireanu. French-Romanian Integrated Proposal for the User Language of E-LOTOS. Rapport SPECTRE 96-05, VERIMAG, Grenoble, mai 1996. Input document [KC3] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Kansas City, Missouri, USA, May, 12–21, 1996.

[GS96b] Hubert Garavel and Mihaela Sighireanu. On the Introduction of Exceptions in LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the*

Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany), pages 469–484. IFIP, Chapman & Hall, octobre 1996.

[GT93] Hubert Garavel and Philippe Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In Rachida Dssouli and Gregor v. Bochmann, editors, *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'93 (Montréal, Canada)*, 1993.

[Jan85] H.-St Jansohn. Automated Generation of Optimized Code. *GMD-Bericht*, (154), 1985.

[JGL+95] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21 N10108 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, octobre 1995.

[Jou91] Jean-Philippe Jouve. *Réalisation du décompilateur d'arbres attribués du système* FNC-2 PPAT *(a Pretty–Printer for Attributed Trees)*. PhD thesis, Ecole Polytechnique., Juillet 1991. Rapport de stage d'option. Directeur de stage : Monsieur Jourdan INRIA.

[JP90] Martin Jourdan and Didier Parigot. Application development with the FNC–2 attribute grammar system. In *Dieter Hammer, editor, Compiler Compilers '90, Lecture Notes in Computer Science, pages 85-94. Springer-Verlag, New York-Heidelberg-Berlin*, 1990.

[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.

[JPJ+90] M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the fnc–2 attribute grammar system. Technical Report, INRIA Rocquencourt (France), Jun 1990.

[Knu68] D. E. Knuth. Semantics of context–free languages, June 1968., Mars 1968. Math. Systems Theory 2,1 pp. 127–145, correction : *Math. Systems Theory 5,1 pp. 95–96*.

[KW76] Ken Kennedy and S. K. Warren. Automatic generation of efficient evaluators for attribute grammars., Janvier 1976. 3rd ACM symp. on principles of Progr. Languages, Atlanta, Ge, 32–49.

[Lor74] Bernard Lorho. De la définition à la traduction des langages de programmation: méthode des attributs sémantiques. Master's thesis, Université Paul Sabatier, Toulouse, Novembr 1974.

[Lor77] Bernard Lorho. *Semantic Attributes Processing in the system DELTA.* A. Ershov Cornelius H. A. Koster.(eds.), 1977. Methods of Algorithmic Language Implementation.

[Lut97] Bas Luttik. Description and Formal Specification of the Link Layer of P1394. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, juin 1997.

[Mat97] Radu Mateescu. Vérification de systèmes répartis : l'exemple du protocole BRP. *Technique et Science Informatiques*, 16(6):725–751, juin 1997.

[Par97] Terence John Parr. *Language Translation Using PCCTS & C++ : A Reference Guide.* Automata, February 1997.

[PRDJ95] D. Parigot, G. Roussel, E. Duris, and M. Jourdan. Les grammaires attribuées : un langage fonctionnel déclaratif. In *journées du* GDR *de programmation, Grenoble, November 1995*, 1995.

[Queer] Juan Quemada, editor. Committee Draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG7 Project 1.21.20.2.3, 1997 janvier.

[RD94] R.Wilhem and D.Maurer. *Les compilateurs,théorie, construction, génération.* MASSON, 1994.

[SG95a] Mihaela Sighireanu and Hubert Garavel. Application of the Proposed E-LOTOS Datatype Language to the Description of OSI and ODP Standards. Rapport SPECTRE 95-17, VERIMAG, Grenoble, décembre 1995. Input document [LG2] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

[SG95b] Mihaela Sighireanu and Hubert Garavel. Defect Report concerning the LOTOS Description of OSI TP Protocol. Rapport SPECTRE 95-18, VERIMAG, Grenoble, décembre 1995.

[SG95c] Mihaela Sighireanu and Hubert Garavel. A Proposal for the Data Type Part of E-LOTOS Applicable to the Formal Description of OSI and ODP Standards. Rapport SPECTRE 95-20, VERIMAG, Grenoble, décembre 1995. Input document [LG4] of the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

[SG96] Mihaela Sighireanu and Hubert Garavel. E-LOTOS User Language. Rapport SPECTRE 96-06, VERIMAG, Grenoble, octobre 1996. In ISO/IEC JTC1/SC21 Third Working Draft on Enhancements to LOTOS (1.21.20.2.3). Output document of the edition meeting, Kansas City, Missouri, USA, May, 12–21, 1996.

[SM97] Mihaela Sighireanu and Radu Mateescu. Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop* *on Applied Formal Methods in System Design (Zagreb, Croatia)*, juin 1997. Full version available as INRIA Research Report RR-3172.

[Viv96] Bruno Vivien. Etude du système SYNTAX/FNC-2 pour la génération de compilateurs. Mémoire de probatoire en informatique, CNAM, Grenoble, juin 1996.

Index

Résumé :

Le développement de systèmes informatiques complexes induit des exigences fortes en termes de logiciels certifiés. Dans ce contexte, on assiste a l'émergence de nouvelles méthodes de génie logiciel, basées sur les techniques de descriptions formelles et de vérification de programmes.

Dans les années 1980 et au début des années 1990, trois techniques de descriptions formelles (LOTOS, ESTELLE et SDL) ont été définies au sein des organismes de normalisation. Plus récemment, afin de surmonter les diverses lacunes de ces trois techniques, l'ISO/IEC a entrepris une révision majeure du langage LOTOS, qui doit déboucher sur un nouveau langage appelé E-LOTOS.

Notre travail concerne l'étude et la réalisation d'un prototype de compilateur (nommé TRAIAN) pour le langage E-LOTOS. La définition de ce langage n'étant pas encore stabilisée, et pour obtenir un premier prototype de compilateur facilement modifiable, nous avons choisi de baser nos développements sur l'outil SYNTAX/FNC-2, un système moderne pour la génération de compilateurs.

La production du compilateur TRAIAN repose sur les grammaires BNF et sur l'utilisation de plusieurs langages spécialisés pour la description d'arbres abstraits, la construction d'arbres, l'écriture des grammaires attribuées et la production de code cible.

Comme sous-produits de notre travail, deux paragrapheurs pour LOTOS et E-LOTOS ont été réalisés.

Mots-clés :

compilation, génie logiciel, grammaires attribuées, ingénierie des protocoles, méthodes formelles.

Keywords:

compilation, protocol engineering, attribute grammars, software engineering, formal methods.