UNIVERSITÉ DU QUÉBEC À MONTRÉAL

TRANSFORMATION AUTOMATIQUE D'ARBRES SYNTAXIQUES AVEC SABLECC

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

OPTION INFORMATIQUE SYSTÈME

PAR

AGBAKPEM KOMIVI KEVIN

Avril 2006

Remerciements

Je tiens à témoigner toute ma reconnaissance à mon directeur de recherche *Étienne Gagnon*, professeur au département d'informatique de l'*UQÀM*, pour le soutien technique et financier qu'il m'a apporté pendant toute la durée de ma maîtrise. *Étienne* a toujours été là pour m'aider à mener à bien ce travail. Merci *Étienne*.

Je souhaite aussi remercier les membres du groupe de recherche *Sable* de l'université *McGill* et les membres du groupe de recherche *LATECE* de l'*UQÀM* pour m'avoir aidé d'une manière ou d'une autre durant ces années et, surtout, pour avoir contribué à créer une ambiance de travail propice. Je tiens particulièrement à remercier *Hafedh Mili*, professeur au département d'informatique de l'*UQÀM* et directeur du *LATECE* qui au début de mon projet a mis à ma disposition tout le matériel nécessaire à mon travail.

Je tiens aussi à remercier les membres de la communauté *SableCC* qui m'ont largement aidé à améliorer la qualité de l'outil développé grâce aux commentaires et suggestions pertinents dont ils m'ont fait part.

Je tiens aussi à remercier M. *Roger Villemaire*, professeur et directeur du programme de maîtrise en informatique de l'*UQÀM* pour le travail qu'il accomplit.

Je ne pourrais finir cette liste sans dire un grand merci aux membres de ma famille qui m'ont soutenu moralement et financièrement pendant ces années et sans qui la réalisation de ce travail n'aurait pas été possible. Je pense particulièrement à *Boniface Koudjonou* qui a bien voulu faire la révision linguistique de ce document et à ma mère Mme *Agbakpem Amavi*. Un gros merci à ceux là.

À tous, Que Dieu le tout puissant vous bénisse.

Table des matières

Liste des figures

Liste des tableaux

Résumé

Dans la partie frontale d'un compilateur, il est souvent usuel de transformer l'arbre syntaxique concret correspondant à la grammaire utilisée lors de l'analyse syntaxique en un arbre simplifié appelé arbre syntaxique abstrait qui retient uniquement l'essentiel des éléments de l'arbre syntaxique concret et qui par ailleurs est plus approprié pour les étapes ultérieures de la compilation. La réalisation d'une telle transformation avec les générateurs de compilateurs traditionnels peut nécessiter d'écrire une quantité considérable de code répétitif et très enclin aux erreurs dans la grammaire.

Dans ce mémoire, nous introduisons un nouveau mécanisme pour spécifier des transformations d'un arbre syntaxique concret vers un arbre syntaxique abstrait durant l'analyse syntaxique. Doté d'une syntaxe intuitive, ce mécanisme permet la spécification de transformations ascendantes de manière efficace et déterministe. De plus, des vérifications effectuées sur ces transformations garantissent leur exactitude. Nous avons implanté ce mécanisme à l'intérieur de l'outil SableCC, un générateur de compilateurs. Le résultat est un outil capable de générer des analyseurs lexicaux et syntaxiques dans lesquelles sont enchâssées des actions de construction d'arbres syntaxiques et des classes utilitaires (incluant des visiteurs) à partir de simples spécifications grammaticales.

Mots clés : Arbre syntaxique concret ; Arbre syntaxique abstrait ; Générateur de compilateurs ; Analyse syntaxique

Abstract

In the front-end of a compiler, it is customary to transform the concrete syntax tree matching the grammar used for parsing into a simpler abstract syntax tree that only retains the essential elements of the concrete syntax tree and that is more suitable for performing later compilation stages. Using traditional compiler tools, this process can involve writing a significant amount of repetitive and error prone actions in the grammar file.

In this thesis, we introduce a new mechanism for specifying parse-time concrete to abstract syntax tree transformations. Using an intuitive syntax, this mechanism allows for the specification of efficient, deterministic, bottom-up transformations which are subjected to correctness verification. We have implemented this mechanism into the SableCC compiler generator to automatically and robustly generate lexers, parsers with embedded abstract syntax tree construction actions, and additional utility classes (including visitors) from simple grammar specifications.

À ma mère

À ma tante Andrée Kuevidjen

Chapitre I

INTRODUCTION

1.1 Contexte

Le nombre de langages d'ordinateur existant aujourd'hui est assez énorme et continue sans cesse de croître. Les langages machines sont utilisés dans plusieurs domaines pour plusieurs buts différents. Plusieurs types de langages existent. Ils vont des langages de programmation traditionnels comme C, C++ et Java en allant aux langages de *markup* comme HTML et XML ou encore aux langages de modélisation comme UML. L'augmentation sans cesse croissante du nombre de langages machines est certes due à des besoins mais aussi à la facilité de développement des outils de traitement de ces langages, tels que les compilateurs et interpréteurs. Le développement de ces outils est aujourd'hui en grande partie réalisé par des générateurs de compilateurs. Un générateur de compilateur transforme la spécification grammaticale en un compilateur pour le langage décrit par cette spécification, évitant ainsi au programmeur la lourde tâche d'écrire le code des différentes parties comme l'analyseur lexical et l'analyseur syntaxique.

Un générateur de compilateur a donc pour objectif de faciliter le développement du compilateur. Un compilateur est souvent organisé sous forme de phases successives qui opèrent toutes sur une représentation abstraite du langage (Appel, 1998) provenant de la phase précédente s'il y a lieu[1]. Le nombre de phases du compilateur dépend en général de son envergure. Toutefois, certaines phases comme l'analyse lexicale, et

[1]La toute première phase du compilateur opère sur le programme source ou un texte du langage.

l'analyse syntaxique constituent les phases les plus souvent retrouvées dans les compilateurs (Appel, 1998). L'arbre syntaxique est la représentation abstraite d'une phrase du langage issue de la phase d'analyse syntaxique. Il s'agit d'un arbre dont les feuilles contiennent presque tous les mots du programme source à analyser et dont la structure des noeuds internes est conforme à la grammaire. Souvent, l'arbre syntaxique tel qu'il est construit par l'analyseur syntaxique correspond à un arbre concret, c'est-à-dire un arbre dérivé directement de la grammaire d'analyse syntaxique. Cependant, un arbre simplifié est souvent plus approprié pour les phases ultérieures du compilateur ou de l'interpréteur (Appel, 1998; Aho, Sethi et Ullman, 2000) qui l'utilisent comme représentation abstraite. En effet, l'arbre concret est un arbre qui renferme trop d'informations et dont la structure est peu adaptée pour les phases ultérieures de la compilation ou de l'interprétation alors que l'arbre simplifié est dénudé de toute information inutile et est dotée d'une structure adaptée au traitement requis par ces étapes. La nécessité de transformer l'arbre syntaxique concret en un arbre simplifié mieux adapté aux étapes de la compilation ou de l'interprétation appelé arbre syntaxique abstrait naît alors.

Dans les environnements de développement de compilateurs traditionnels, alors que la construction de l'arbre syntaxique concret n'est pas toujours effectuée de manière automatique, celle de l'arbre syntaxique abstrait est presque inexistant ou incomplet sinon exige un travail répétitif souvent enclin aux erreurs.

L'objectif de notre travail de recherche effectué dans le cadre de la maîtrise en informatique a donc été de proposer une approche de transformation automatique des arbres syntaxiques concrets vers des arbres syntaxiques abstraits. Notre approche, se basant sur la spécification par l'utilisateur des transformations à l'intérieur de la grammaire, repose sur une syntaxe simple et concise qui permet de réaliser les transformations d'arbres de manière ascendante lors de la phase d'analyse syntaxique. Cette approche est déterministe et nous y appliquons des vérifications sémantiques afin d'en garantir l'exactitude. Nous avons implanté à l'intérieur de l'outil SableCC, un générateur de compilateurs, cette approche de simplification d'arbres syntaxiques.

1.2 Contributions principales de ce mémoire

Les contributions principales de ce mémoire sont :

- la séparation entre la **grammaire de l'analyse syntaxique** et la **grammaire de l'arbre syntaxique** ;
- l'introduction d'une syntaxe simple et intuitive pour la transformation d'un arbre syntaxique concret (*CST*) vers un arbre syntaxique abstrait (*AST*) basée sur cinq primitives de transformations ;
- l'aspect déterministe des transformations c'est-à-dire la garantie d'obtenir le même arbre (unicité) peu importe l'ordre d'application des transformations individuelles ;
- la construction directe de l'arbre syntaxique abstrait lors de l'analyse syntaxique, ce qui permet une réduction considérable de l'espace mémoire utilisé assurant ainsi une meilleure performance ;
- les vérifications sémantiques effectuées sur les transformations spécifiées par l'usager. Ces vérifications constituent la garantie de l'exactitude de transformations qui seraient acceptées par l'outil ;
- la gestion transparente des grammaires EBNF (*Extended Backus Naur Form*) vis à vis de l'usager. Ceci permet à l'usager d'écrire des grammaires concises sans se soucier du fait que l'algorithme d'analyse syntaxique ne supporte que les grammaires BNF (*Backus-Naur Form*) ;
- la résolution automatique de certains conflits par une méthode basée sur l'inclusion automatique et transparente de productions. Cette fonctionnalité fournit à l'usager une méthode transparente de résolution de conflits qui a l'avantage, contrairement à d'autres méthodes, de ne rien changer au langage reconnu par la grammaire ;
- la mise à disposition de la communauté SableCC du résultat du développement du module des transformations d'arbres syntaxiques à l'intérieur de l'outil SableCC ainsi que diverses ressources, y compris des fichiers d'exemples.

1.3 Organisation du mémoire

Ce mémoire est organisé de la manière suivante : le chapitre 2 décrit l'outil de génération de compilateurs et d'interpréteurs SableCC auquel nous avons intégré les transformations d'arbres syntaxiques. Ce chapitre constitue une base sur laquelle repose le reste de la présentation et contient l'essentiel des prérequis nécessaires pour la lecture de ce mémoire. Dans le chapitre 3, nous présentons les différentes conditions à remplir pour bénéficier des transformations d'arbres syntaxiques et présentons aussi la syntaxe de ces transformations. Le chapitre 4 décrit les vérifications sémantiques effectuées pour les transformations d'arbres syntaxiques. Le chapitre 5 présente une première esquisse de l'implantation en montrant les changements internes effectués aux grammaires d'entrée pour supporter l'algorithme de l'analyse syntaxique et le chapitre 6 complète cette présentation en décrivant en détail l'algorithme utilisé lors de l'implantation des transformations d'arbres syntaxiques. Dans le chapitre 7, nous présentons une approche de résolution automatique de certains conflits que nous avons introduits. Cette approche se base sur l'inclusion des productions impliquées dans le conflit dans la grammaire d'analyse syntaxique pour tenter de les résoudre. Le chapitre 8 présente les différents tests réalisés à la fois pour prouver la validité des transformations mais aussi tester la robustesse du logiciel "à toutes épreuves". Le chapitre 9 présente les différents travaux reliés et enfin le chapitre 10 présente nos conclusions et montre d'éventuelles pistes futures à explorer.

Chapitre II

NOTIONS PRÉLIMINAIRES (L'ENVIRONNEMENT SABLECC)

Ce chapitre présente le logiciel SableCC. Dans un premier temps, nous proposons une brève description du logiciel et de son fonctionnement. Ensuite, nous présentons le fichier de spécification qui constitue l'entrée du logiciel et les différents composants générés par SableCC à partir de ce fichier. L'accent est principalement mis sur le composant correspondant à l'arbre syntaxique car il sera au coeur du travail effectué dans ce mémoire. Pour finir, nous présentons l'architecture interne du logiciel SableCC et faisons un parallèle entre cette architecture et le fichier de spécification.

2.1 Introduction à SableCC

SableCC est un environnement orienté objet pour le développement de compilateurs et d'interpréteurs. Ce développement peut être un processus très long et fastidieux. Le présent mémoire n'a pas la prétention de décrire toutes les étapes que comporte ce processus. Néanmoins, certaines des étapes de ce processus sont supportées par SableCC et seront très brièvement[1] décrites dans ce chapitre.
Un compilateur ou interpréteur compile ou interprète un programme écrit dans un langage de programmation. Compiler le programme, souvent écrit en langage de haut niveau (compréhensible par l'humain), signifie le traduire en instructions machines pouvant être

[1]Se reporter à (Gagnon, 1998) pour plus de détails sur SableCC.

exécutées par le processeur de la machine. Par contre, interpréter un programme suppose que la tâche d'exécution est laissée à l'interpréteur. Dans ce cas, le programme n'est pas traduit en instructions machines. Le compilateur ou l'interpréteur est en général propre à un langage de programmation particulier. Un compilateur est décrit comme étant constitué des différentes parties suivantes (Aho, Sethi et Ullman, 2000) :

- l'analyseur lexical ;
- l'analyseur syntaxique ;
- l'analyseur sémantique ;
- le générateur de code intermédiaire ;
- l'optimiseur de code ;
- le générateur de code ;

SableCC crée certaines des parties de ce compilateur. Pour ce faire, il prend en entrée un fichier contenant la description du langage pour lequel le compilateur doit être écrit et produit en sortie certaines des parties du compilateur. Le fichier d'entrée est appelé *fichier de spécification du langage*. À partir de ce fichier, SableCC génère tout un cadre de travail incluant un *analyseur lexical* et un *analyseur syntaxique*. Développer un compilateur avec SableCC requiert :

1. d'écrire un fichier de spécification définissant le langage pour lequel le compilateur doit être écrit ;
2. de donner ce fichier à SableCC qui génère certaines parties du compilateur ;
3. de compléter le compilateur en se servant des fichiers générés par SableCC et en écrivant le reste qui, d'après la description des différents composants selon (Aho, Sethi et Ullman, 2000), serait l'analyseur sémantique, le générateur de code intermédiaire, l'optimiseur de code et le générateur de code.

La figure 2.1 résume les différentes étapes de développement d'un compilateur ou d'un interpréteur avec SableCC.

La définition suivante a pour but de permettre la compréhension du paragraphe à suivre.

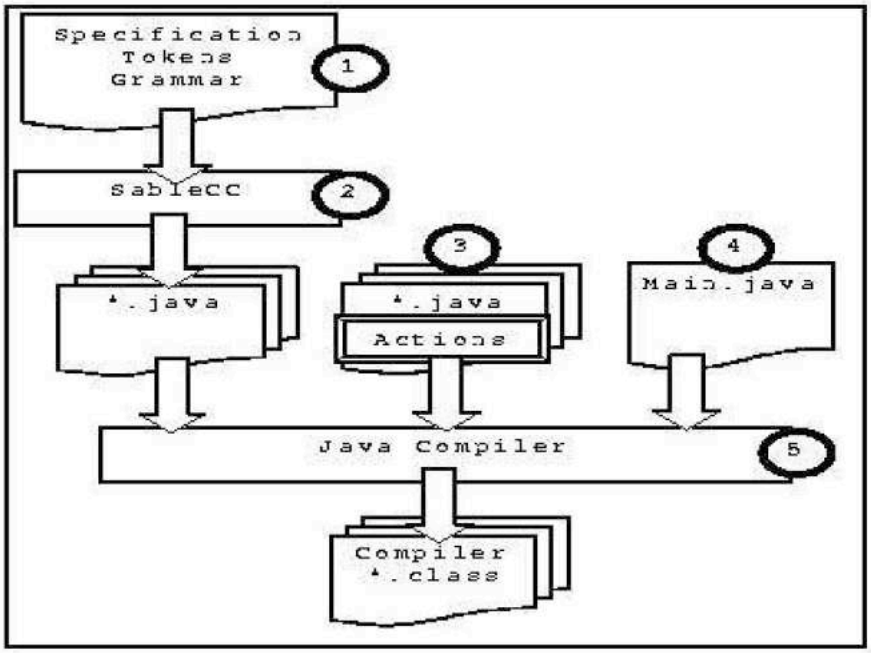

Figure 2.1 Étapes de développement d'un compilateur avec SableCC - source : (Gagnon, 1998)

Définition 2.1 (Notation Backus-Naur) *Une notation Backus-Naur (BNF ou* `Backus Naur Form` *en anglais) ou grammaire non contextuelle est le quadruplet* $G = (V_N, V_T, P, S)$ *où* V_N *un ensemble des symboles non-terminaux,* V_T *est un ensemble de jetons appelés symboles terminaux,* $P \epsilon V_N \times (V_N U V_T)^*$ *est un ensemble de productions constituées d'un non-terminal appelé partie gauche et d'une séquence de non-terminaux et/ou de terminaux appelés partie droite, et* $S \in V_N$ *est le symbole de départ ou l'axiome (Gagnon, 1998).*

2.2 Le fichier de spécification

Nous allons d'abord décrire les différentes sections composant ce fichier. Ensuite, nous parlerons de la version "évoluée" de la notation BNF dont se sert SableCC pour définir une des sections de ce fichier et, pour finir, des règles de nomenclature à respecter pour cette même section.

2.2.1 Les différentes sections

Le fichier de spécification contient les règles grammaticales et lexicales devant être satisfaites par les programmes compatibles avec le langage. Les règles grammaticales fixent la structure générale des programmes c'est-à-dire la forme des constructions qui les composent. Les règles lexicales quant à elles définissent les différents mots faisant partie du langage, mots qu'on peut donc retrouver dans les programmes de ce langage. Il existe deux sortes de mots : les mots prédéfinis, appelés mots-clés, et les identificateurs. Les identificateurs ne sont pas connus à l'avance. Seul le *patron*[2] (*modèle*) qu'ils doivent satisfaire est défini. Le fichier de spécification est organisé sous forme de sections. Il y en a cinq principales : `Helpers`, `States`, `Tokens`, `Ignored Tokens` et `Productions`. Une dernière section appelée `Package` sert à spécifier le répertoire de destination du cadre de travail généré par SableCC. Chacune des cinq premières sections est reliée soit aux règles lexicales, soit aux règles grammaticales. Ainsi, les sections `Helpers`,

[2]Expressions régulières servant à représenter les jetons.

`Tokens` et `States` constituent la définition lexicale et les sections `Ignored Tokens` et `Productions` correspondent à la définition grammaticale. Les mots pouvant faire partie du langage sont définis grâce à ce qu'on appelle des *jetons* (ou *tokens* en anglais). Tous les *jetons* doivent être définis dans la section `Tokens`. La section `Helpers` contient des "*sous-jetons*" dont on peut se servir pour définir des *jetons* plus facilement. Par exemple, dans cette section, on peut définir un *sous-jeton lettre* qui correspond aux lettres minuscules [`a..z`] et par la suite utiliser *lettre* dans la définition de plusieurs *jetons* dans la section `Tokens`. Les *jetons* et les *sous-jetons* sont définis en utilisant les expressions régulières. L'expression régulière utilisée pour définir un *jeton* est appelée modèle du *jeton*. La section `States` quant à elle définit des états pouvant être utilisés lors de la définition des *jetons*. La section `Productions` correspond à la définition de la *grammaire* du langage. Cette *grammaire* est souvent appelée *grammaire non contextuelle* ou *notation Backus-Naur (BNF)*. Conformément à la définition 2.1, une *grammaire* est donc constituée d'une série de *productions*. Une *production* est à son tour constituée d'un ou de plusieurs symboles terminaux ou non terminaux. La définition d'une *production* dans le fichier de spécification de SableCC suit la syntaxe suivante :

$$\texttt{prod} \;=\; \texttt{elem}_1\ \texttt{elem}_2\ \texttt{elem}_3\ \ldots\ \texttt{elem}_N;$$

où $\texttt{elem}_1$, $\texttt{elem}_2$, $\texttt{elem}_3$, ..., $\texttt{elem}_N$ sont soit des symboles terminaux (*jetons*), soit des symboles non terminaux (*productions*).

Une *production* correspond en réalité à la définition d'une ou de plusieurs règles grammaticales (ou syntaxiques). Une *production* peut avoir plus d'une définition. Dans ce cas, les différentes définitions sont séparées par le caractère "`|`". Les différentes définitions d'une même *production* sont appelées *alternatives de la production*. Ainsi, si on considère la *production* de la figure 2.2, elle est constituée de deux *alternatives*. La première est constituée des symboles terminaux `var`, `plus` et `nombre`. Le mot entre les accolades `{}` correspond au nom de cette *alternative*. Ainsi le nom de la première *alternative* de cette *production* est donc `plus`. La deuxième *alternative* de nom `moins` est elle constituée des symboles `var`, `moins` et `nombre` (on suppose que `var`, `plus`, `moins` et `nombre` sont des *jetons* et qu'ils ont été définis dans la section appropriée du fichier de spécification,

c'est- à-dire la section `Tokens`). La section `Ignored Tokens` regroupe les *jetons* ne pouvant pas être utilisés lors de la définition grammaticale mais qui peuvent quand même apparaître dans les programmes du langage. Quand les mots correspondant à ces *jetons* sont vus dans les programmes du langage, ils sont tout simplement ignorés. C'est pour cette raison que ces *jetons* sont appelés *jetons ignorés*. Les *jetons ignorés* doivent être définis dans la section `Ignored Tokens` sous forme de listes séparées par des virgules (**,**). Un exemple de *jeton* souvent mis dans cette section est le *jeton* `blanc`. Il représente les caractères *espace*, *tabulation*, *retour à la ligne*, *saut de page*, etc., en somme, tous les caractères destinés à l'espacement du programme. Un autre exemple de ce type de *jeton* est aussi le *jeton* `commentaire`. Une dernière section appelée `Package` permet de spécifier le répertoire dans lequel seront mis les fichiers générés par SableCC. Un exemple est `org.sablecc`. Dans un tel cas de figure, les fichiers générés par SableCC seront mis dans le sous répertoire relatif *org/sablecc/*.

```
expression  =  {plus}    var  plus  nombre
            |  {moins}   var  moins  nombre
            ;
```

Figure 2.2 Exemple de définition d'une production

2.2.2 Notation "EBNF" supportée par SableCC

SableCC supporte plus que la simple *notation BNF*. En effet, il est permis d'utiliser des opérateurs $*$, $+$ et ? pour faciliter la définition des *productions*. Ces opérateurs peuvent apparaître après les symboles terminaux ou non terminaux dans une *alternative* et ont une signification spéciale. Cette version plus évoluée se situe entre la *notation BNF* et la *notation BNF étendue EBNF (Extended Backus Naur Form)*. En effet, certains des opérateurs de la *notation EBNF* ne sont pas supportés, par exemple, les parenthèses `( )` dont le rôle consiste à regrouper les éléments. La signification des différents opérateurs supportés par SableCC est présentée ci-dessous.

L'opérateur ? : Il permet d'exprimer la présence optionnelle du symbole qu'il suit.

Exemple 2.1

$\texttt{prod} = \texttt{elem}_1\ \texttt{elem}_2?\ ;$

équivaut à :

$\texttt{prod}\ =\ \texttt{elem}_1$

$\quad |\ \texttt{elem}_1\ \ \texttt{elem}_2\ ;$

L'opérateur + : Il permet d'exprimer la possibilité d'une occurrence multiple du symbole qu'il suit.

Exemple 2.2

$\texttt{prod}\ =\ \texttt{elem}_1\ \texttt{elem}_2+\ ;$

équivaut à :

$\texttt{prod}\ =\ \texttt{elem}_1\ \$\texttt{elem}_2$

$\$\texttt{elem}_2 = \texttt{elem}_2$

$\quad |\ \ \$\texttt{elem}_2\ \texttt{elem}_2\ ;$

L'opérateur * : Cet opérateur peut être réécrit en utilisant les deux premiers ci-dessus. En effet, $\texttt{elem}* \Rightarrow (\texttt{elem}+)?$, c'est-à-dire $(\texttt{elem} + |\ )$.

Exemple 2.3

$prod\ =\ elem_1\ elem_2*\ ;$

équivaut à :

$\texttt{prod}\ =\ \texttt{elem}_1\ \texttt{elem}_2+$

$\quad |\ \ \texttt{elem}_1\ ;$

qui à son tour équivaut à :

$\texttt{prod}\ =\ \texttt{elem}_1\ \$\texttt{elem}_2$

$\quad |\ \ \texttt{elem}_1\ ;$

$\$\texttt{elem}_2 = \texttt{elem}_2$

$\quad |\ \ \$\texttt{elem}_2\ \texttt{elem}_2\ ;$

L'opérateur * permet d'exprimer la possibilité d'une occurrence multiple du symbole qui le précède mais aussi l'absence de ce symbole.

Remarque importante : SableCC transforme à l'interne la *grammaire* "EBNF" en son équivalent BNF en opérant les transformations décrites ci-dessus. Au final, la *grammaire* avec laquelle travaille SableCC est une *grammaire* BNF correspondant à l'équivalent EBNF spécifiée par l'usager.

2.2.3 Restrictions et règles de nomenclature

La première règle de nomenclature en vigueur dans SableCC exige que les *alternatives* d'une même *production* aient des noms différents. La deuxième spécifie que si deux éléments à l'intérieur d'une même *alternative* sont de même type, un nom différent doit leur être donné pour les différencier. Pour illustrer le problème, supposons par exemple qu'on ait une *alternative* additionnelle dans la *production* `expression` de la figure 2.2 qui se définit ainsi : `{mult} nombre mult nombre`.
SableCC oblige dans ce cas de renommer un des deux éléments `nombre` conformément aux règles de nomenclature. Donner un nom à un élément se fait de la manière suivante : `[nom_donné] :element`. Ainsi, en réécrivant `{mult} [gauche] :nombre mult [droite] :nombre`, l'*alternative* `mult` se conforme alors aux règles de nomenclature de SableCC. La nouvelle *production* `expression` résultante serait donc :

```
expression = {plus} var plus nombre
           | {moins} var moins nombre
           | {mult} [gauche] :nombre mult [droite] :nombre
           ;
```

2.3 Les différents composants générés

Rappelons qu'à partir du fichier de spécification, SableCC génère les composants suivants :

- *un analyseur lexical :* il s'agit de la partie du compilateur qui découpe le flot de caractères composant le programme source d'entrée (le texte) en *jetons*, conformément à ceux qui sont définis dans le fichier de spécification. Ces *jetons*

sont aussi appelés *unités lexicales*. À partir du programme d'entrée, l'*analyseur lexical* fournit une série de *jetons*.

- *un analyseur syntaxique :* cette partie du compilateur vérifie que le programme texte en entrée est conforme à la structure fixée par la *grammaire* en s'assurant du respect des règles grammaticales décrites par les *alternatives* des *productions*. Il fonctionne en étroite collaboration avec l'*analyseur lexical* car c'est avec les *jetons* fournis par l'*analyseur lexical* que se fait la vérification du respect des règles grammaticales. En fait, les *jetons* fournis par l'*analyseur lexical* servent d'éléments de base pour l'*analyseur syntaxique*.
- *un ensemble de classes permettant de construire un arbre syntaxique et de parcourir cet arbre*. L'*arbre syntaxique* est un arbre dont les feuilles contiennent presque[3] tous les mots du programme source à compiler et dont la structure des noeuds internes reflète de façon exacte la *grammaire*.

2.3.1 L'analyseur lexical

2.3.1.1 Rôle

Le rôle de l'*analyseur lexical* consiste à valider lexicalement le texte ou programme en entrée, c'est-à-dire s'assurer que tous les mots de ce texte quels qu'ils soient (entièrement[4] visibles ou non) correspondent à une des *unités lexicales* définies dans la section `Tokens`. Dans le fichier de spécification de SableCC, les expressions régulières sont utilisées pour définir les *unités lexicales* suivant la syntaxe :

```
identificateur de l'unité lexicale suivi du signe égal (=) et d'une
expression régulière.
```

[3]Les *jetons* ignorés n'y sont pas inclus.

[4]Certains caractères à l'oeil sont représentés de la même façon. L'espace blanc et le caractère retour chariot par exemple.

L'identificateur de l'*unité lexicale* est tout simplement appelé *unité lexicale*. Par contre, l'expression régulière définissant les mots que peut reconnaître l'*unité lexicale* est appelée modèle (Aho, Sethi et Ullman, 2000). Donc, la définition d'une *unité lexicale* peut être réécrite de la manière suivante :

```
unité lexicale = modèle
```

Exemple : `class = 'class'`;
Dans l'exemple ci-dessus, `class` représente l'*unité lexicale*, `'class'` correspond au modèle de l'*unité lexicale* et ce modèle ne permet d'écrire qu'un seul mot qui est "`class`". Il s'agit dans ce cas-ci d'un mot prédéni.

2.3.1.2 Principe de fonctionnement de l'analyseur lexical

À partir de la section `Tokens`, plus précisement des modèles définis pour les *jetons*, SableCC crée un ou plusieurs automates finis déterministes[5]. Ensuite, il génère l'*analyseur lexical* en se servant de ces automates. Lorsqu'il est invoqué, l'*analyseur lexical* lit le texte en entrée, caractère par caractère, et s'arrête dès qu'il reconnaît un mot satisfaisant le modèle d'une des *unités lexicales* définies dans la section `Tokens`. Il retourne alors l'*unité lexicale* associée au modèle reconnu. Et le même processus reprend jusqu'à ce que la fin de fichier soit atteinte ou qu'un mot incorrect, c'est-à-dire un mot qui ne correspond à aucune des *unités lexicales*, soit détecté par l'*analyseur lexical*.
Les fichiers de l'*analyseur lexical* sont générés dans un sous-répertoire appelé `lexer` qui est créé par SableCC. La classe *Lexer* dans le fichier `Lexer.java` contient l'essentiel du code ; autrement dit le code nécessaire pour effectuer l'*analyse lexicale*.

2.3.2 L'analyseur syntaxique

Il convient ici de définir tout d'abord certains concepts.

Dans la *grammaire* de la figure 2.3, on suppose que les symboles en majuscules

[5]Une définition peut être trouvée dans (Aho, Sethi et Ullman, 2000).

programme	=	DEBUT corps FIN
corps	=	ACC_G instruction ACC_D
instruction	=	{assign} [id_g]: ID EGAL [id_d]: ID ;
	|	{comparaison} [id_g]: ID EGAL_EGAL [id_d]: ID ;

Figure 2.3 Exemple de grammaire

sont des terminaux et que les symboles en minuscules sont des non-terminaux.

Les définitions dans le reste de ce chapitre sont toutes tirées de (Aho, Sethi et Ullman, 2000).

Définition 2.2 (Dérivation) *On appelle dérivation d'un non-terminal S, le processus durant lequel un non terminal S en partie gauche est remplacé par la chaîne en partie droite de la production de S. Par exemple, si on considère la grammaire de la figure 2.3, une dérivation du symbole* `programme` *serait* `programme` $\Rightarrow$ `DEBUT corps FIN` *qui se lit "programme se dérive en DEBUT corps FIN". De la même manière, la séquence de symboles* `DEBUT corps FIN` *peut aussi être dérivée et produire la séquence suivante :* `DEBUT corps FIN` $\Rightarrow$ `DEBUT ACC_G` *instruction* `ACC_D FIN`*. On a donc en réalité une succession de dérivations qui sont :* `programme` $\Rightarrow$ `DEBUT corps FIN` $\Rightarrow$ `DEBUT ACC_G instruction ACC_D FIN` *qui peut être réécrite de la manière suivante :* `programme` $\overset{*}{\Rightarrow}$[6] `DEBUT ACC_G instruction ACC_D`*. Donc, une dérivation peut aussi être faite sur le résultat d'une autre dérivation. De façon plus abstraite, on dit que* `aAb` $\Rightarrow$ `a`β`b` *si* `A` $= \beta$ *est une production et* `a` *et* `b` *sont des chaînes arbitraires de symboles grammaticaux (terminaux ou non terminaux).*

Définition 2.3 (Langage engendré par une grammaire) *Soit une grammaire G et S son axiome. Le langage engendré par une grammaire $L(G)$ est l'ensemble des pro-*

[6]Le signe $*$ au dessus de $\Rightarrow$ signifie que la dérivation se fait zéro ou plusieurs fois.

grammes pouvant être écrits pour cette grammaire. Un programme est souvent appelé phrase du langage. Le langage engendré par une grammaire peut donc être défini en utilisant des dérivations successives de l'axiome de la grammaire.

Définition 2.4 (Proto-phrase d'une grammaire) *Si G est une grammaire, on appelle proto-phrase de G, le résultat de zéro, ou de plusieurs dérivations de l'axiome de la grammaire S. Une proto-phrase peut contenir des symboles non terminaux. Par exemple, la chaine* `DEBUT corps FIN` *est une proto-phrase de la grammaire de la figure 2.3. Une* ***phrase*** *est une proto-phrase sans non terminal.*

Définition 2.5 (Arbre syntaxique) *Étant donné une grammaire non contextuelle ou notation Backus-Naur, un arbre syntaxique est un arbre possédant les propriétés suivantes :*

1. *la racine est étiquetée par l'axiome ;*
2. *Chaque feuille est étiquetée par une unité lexicale ou par epsilon ϵ ;*
3. *chaque noeud intérieur est étiqueté par un non-terminal ;*
4. *si A est le non-terminal étiquetant un noeud intérieur et si les étiquettes des fils de ce noeud sont, de gauche à droite, X_1, X_2, ..., X_n, alors $A = X_1\ X_2\ ...\ X_n$ est une production. Ici, X_1, X_2, ..., X_n représentent soit un symbole non-terminal, soit un symbole terminal. Un cas particulier est $A = \epsilon$ qui signifie qu'un noeud étiqueté A a un seul fils étiqueté ϵ.*

En guise d'exemple, l'arbre syntaxique correspondant à la phrase "`DEBUT ACC_G ID EGAL ID ACC_D FIN`*" pour la grammaire de la figure 2.3 est représentée sur la figure 2.4.*

2.3.2.1 Rôle

L'*analyseur syntaxique* généré par SableCC fait la validation grammaticale du programme ou texte d'entrée. Pour ce faire, il se base sur les règles grammaticales spécifiées dans la *grammaire*. Comme précédemment mentionné, une règle grammaticale

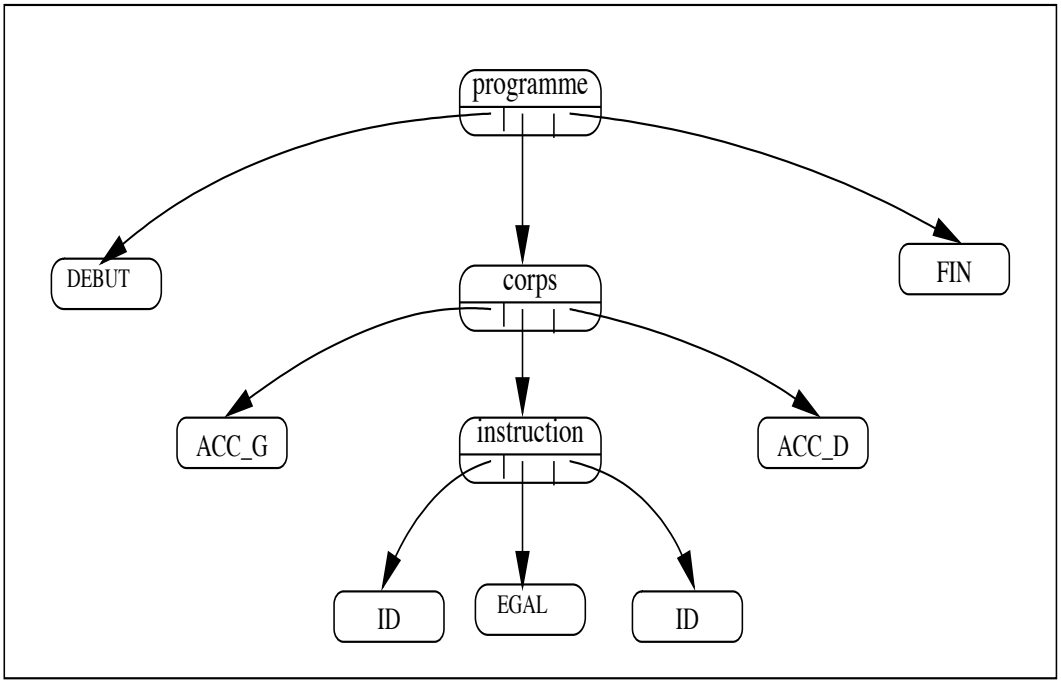

Figure 2.4 Arbre syntaxique de la phrase `DEBUT ACC_G ID EGAL ID ACC_D FIN`

correspond à une *alternative* d'une *production*. L'*analyseur syntaxique* ne lit pas directement le fichier en entrée. Il reçoit un flot d'*unités lexicales* par l'intermédiaire de l'*analyseur lexical* et c'est à partir de ces *unités lexicales* que sont appliquées les règles de validation.

2.3.2.2 Principe de fonctionnement

Un exemple de dérivation du symbole non terminal `programme` de la *grammaire* de la figure 2.3 serait : `programme` ⇒ `DEBUT corps FIN` ⇒ `DEBUT ACC_G instruction ACC_D FIN` ⇒ `DEBUT ACC_G ID EGAL ID ACC_D FIN`. La phrase `DEBUT ACC_G ID EGAL ID ACC_D FIN` est une phrase valide conformément à cette *grammaire*. Les symboles constituant cette phrase correspondent aux symboles fournis par l'*analyseur lexical*. Pour valider une phrase à partir d'une règle, il suffit donc de trouver une dérivation du symbole non-terminal gauche de la règle qui corresponde à cette phrase. L'*analyseur syntaxique* commence la validation grammaticale par la règle de l'axiome de la *gram-*

maire. Elle correspond à une des *alternatives* de la première *production* de la section `Productions` dans le fichier de spécification de SableCC. Le but est de trouver une dérivation de l'axiome qui corresponde à la séquence d'*unités lexicales* venant de l'*analyseur lexical*. Si une telle dérivation existe et est possible, alors l'analyse est dite réussie, sinon l'analyse est dite échouée et s'arrête. En plus de faire la validation grammaticale du programme d'entrée, l'*analyseur syntaxique* de SableCC construit automatiquement un *arbre syntaxique* pour ce programme. Nous allons maintenant décrire les classes (code Java) des arbres syntaxiques construits par SableCC et voir comment l'*analyseur syntaxique* construit ces arbres. Mais avant cela, nous allons parler de la particularité des arbres syntaxiques générés par SableCC : *le typage*.

2.3.3 Les arbres syntaxiques typés de SableCC

Pour une phrase donnée, l'*arbre syntaxique* équivalent peut être associé à une dérivation particulière de l'axiome de la *grammaire* qui est constituée d'une ou de plusieurs séquences de remplacement. À partir de cette séquence de remplacement, on peut déduire la forme générale de l'*arbre syntaxique* ainsi que les différents noeuds et feuilles qui la composent. Il est assez facile de voir que l'*arbre syntaxique* de la figure 2.4 est construit en utilisant la dérivation suivante :

`programme` ⇒ `DEBUT corps FIN` ⇒ `DEBUT ACC_G instruction ACC_D FIN`
⇒ `DEBUT ACC_G ID EGAL ID ACC_D FIN`

Tous les symboles non-terminaux de la *grammaire* de la figure 2.3 ayant servi à dériver cette phrase font partie d'une *production* ayant une seule *alternative* sauf le symbole `instruction`. Le symbole `instruction` a en effet deux *alternatives* différentes. Cela suppose donc au moment de la dérivation de ce non-terminal qu'il faut choisir une des deux *alternatives* de cette *production*. Dans le cas de la dérivation ayant servi pour la construction de l'arbre de la figure 2.4, l'*alternative* `assign` a été choisie. L'*arbre syntaxique* de la figure 2.4 ne réflète pas ce choix car elle ne fait pas apparaître laquelle des *alternatives* est choisie au niveau du noeud *instruction*. Mais en réalité, les arbres syntaxiques construits par les *analyseurs syntaxiques* générés par SableCC in-

tègrent l'information permettant de connaître exactement l'*alternative* choisie pour un remplacement lors d'une dérivation. En effet, pour deux dérivations différentes du même symbole non-terminal utilisant deux *alternatives* différentes de la même *production*, deux noeuds de types différents sont construits au niveau de l'*arbre syntaxique*. On dit alors que l'*arbre syntaxique* est strictement typé ou fortement typé. Un exemple d'un tel arbre sera présenté dans la section suivante.

2.3.4 Les classes de l'arbre syntaxique et le parcours

Comme nous venons de le voir, la construction de l'*arbre syntaxique* utilise la *grammaire* du langage, c'est-à-dire la section `Productions` du fichier de spécification de SableCC. En effet, à partir de cette section, SableCC génère les classes qui serviront à bâtir l'*arbre syntaxique* mais aussi celles qui seront utilisées pour définir des traverseurs (parcours) sur cet arbre. Nous allons maintenant décrire les différents types de classes générées ainsi que la nomenclature utilisée pour les générer.

2.3.4.1 Les classes de l'arbre syntaxique

En ce qui concerne les classes de l'*arbre syntaxique* uniquement, SableCC génère :

- une classe pour chacune des *productions* définies dans la section `Productions`. Le nom utilisé pour ces classes est formé par la concaténation d'un "`P`" suivi du symbole non terminal représentant la *production* mis dans un certain format ;
- une classe pour chacune des *alternatives* de cette *production*. Le nom des classes est formé par la concaténation d'un "`A`" suivi du nom de l'*alternative* et de celui de la *production*. Tous les deux mis dans un certain format ;
- une classe pour chacune des *unités lexicales* définies dans la section `Tokens`. Le nom de ces classes est formé par un "`T`" suivi du nom du symbole de l'*unité lexicale* encore une fois mis dans un certain format ;
- un certain nombre de classes utilitaires dont le but est de permettre la construction d'un arbre robuste.

Symbole ou nom d'alternative	Nom associé
programme	Programme
acc_g	AccG
if_then_else	IfThenElse

Tableau 2.1 Nomenclature des symboles après transformation par SableCC

Le format utilisé par SableCC pour les symboles et les noms d'alternatives pour les noeuds de l'*arbre syntaxique* consiste à convertir la première lettre de ces symboles en majuscule et si dans le symbole, il y a un caractère "_", ce caractère est supprimé et le caractère suivant ce symbole est aussi converti en majuscule. La table 2.1 montre des exemples de conversions.

Voici certaines des classes générées par SableCC pour la *grammaire* de la figure 2.3. Ces classes sont reliées entre elles par une hiérarchie sous-jacente qui est induite par la syntaxe Java.

- En général (à la fois pour les symboles non terminaux et terminaux) :

  `abstract class Node {}` et

  `abstract class Token extends Node {String text ;}`

- Pour les *unités lexicales* `debut`, `fin`, `acc_g`, `acc_d`, `egal`, `egal_egal`, `id` :

```
class TDebut extends Token {};
class TFin extends Token {};
class TAccG extends Token {};
class TAccD extends Token {};
class TEgal extends Token {};
class TEgalEgal extends Token {};
class TId extends Token {} .
```

- Pour les *productions* :
  - la *production* `programme` :

```
abstract class PProgramme extends Node {};
```

– l'unique *alternative* de cette *production* :

```
class AProgramme extends PProgramme {
      TDebut elem1 ;
      PCorps elem2 ;
      TFin elem3 ;
}
```

– la *production* `corps` :

```
abstract class PCorps extends Node {};
```

– l'unique *alternative* de cette *production* :

```
class ACorps extends PCorps {
      TACCG elem1 ;
      PInstruction elem2 ;
      TACCD elem3 ;
}
```

– la *production* `instruction` :

```
abstract class PInstruction extends Node {}
```

– l'*alternative* `{assign}` :

```
class AAssignInstruction extends PInstruction {
      TId elem1 ;
      TEgal elem2 ;
      TId elem3 ;
}
```

– l'*alternative* `{comparaison}` :

```
class AComparaisonInstruction extends PInstruction
      TId elem1 ;
      TEgalEgal elem2 ;
      TId elem3 ;
}
```

- Les classes `TDebut`, `TFin`, `TAccG`, `TAccD`, `TId`, `TEgal`, `TEgalEgal` pour les *unités lexicales* de la *grammaire*. Ces classes héritent toutes de la classe `Token` qui contient un champ texte de type chaîne de caractère (`String`) servant à contenir le mot du texte d'entrée ou du programme correspondant à l'*unité lexicale* (le lexème).
- Les classes `PProgramme`, `PCorps` et `PInstruction` pour les trois *productions* `programme`, `corps` et `instruction` de la *grammaire*. Ces classes sont abstraites, ce qui signifie qu'on ne peut pas créer des instances de ces classes. En effet, l'*arbre syntaxique* est une structure en mémoire, donc constituée de noeuds, instances de classes qui sont construits à l'aide de constructeurs associés à ces classes. Ces classes ne peuvent donc pas être utilisées pour construire les noeuds d'un tel arbre.
- Les classes `AProgramme`, `ACorps`, `AAssignInstruction` et `AComparaisonInstruction` pour les différentes *alternatives* de ces *productions*. Ces classes serviront à créer des noeuds de l'*arbre syntaxique*. C'est la raison pour laquelle elles contiennent des champs qui correspondent aux différents symboles constituant l'*alternative*. En effet, la classe `ACorps` correspondant à l'*alternative*

  ```
  corps = ACC_G instruction ACC_D
  ```

  comporte trois champs dont les types correspondent aux symboles dans l'alternative.

Toutes les classes générées par SableCC pour la construction de l'*arbre syntaxique* n'ont pas été décrites ici pour ne pas alourdir le texte et aussi parce qu'elles ne sont pas toutes pertinentes pour la compréhension du reste. Pour avoir une idée complète de ces classes, le lecteur est invité à consulter le contenu du package `node` du code source de SableCC disponible gratuitement à l'adresse suivante : `http://sablecc.org`.

2.3.4.2 Arbre syntaxique fortement typé

En réalité, l'*arbre syntaxique* de la figure 2.4 n'est pas tout à fait l'arbre construit par l'*analyseur syntaxique* de la *grammaire* de la figure 2.3. En effet, conformément à la

description donnée pour les classes de l'*arbre syntaxique* dans le paragraphe ci-dessus, le type des noeuds de l'*arbre syntaxique* est plus spécifique. Nous avions dans un premier temps représenté l'arbre de cette façon pour permettre une compréhension rapide de la façon dont il était construit. Le vrai arbre résultant de l'analyse syntaxique de la phrase `DEBUT ACC_G ID EGAL ID ACC_D FIN` est représenté à la figure 2.5.

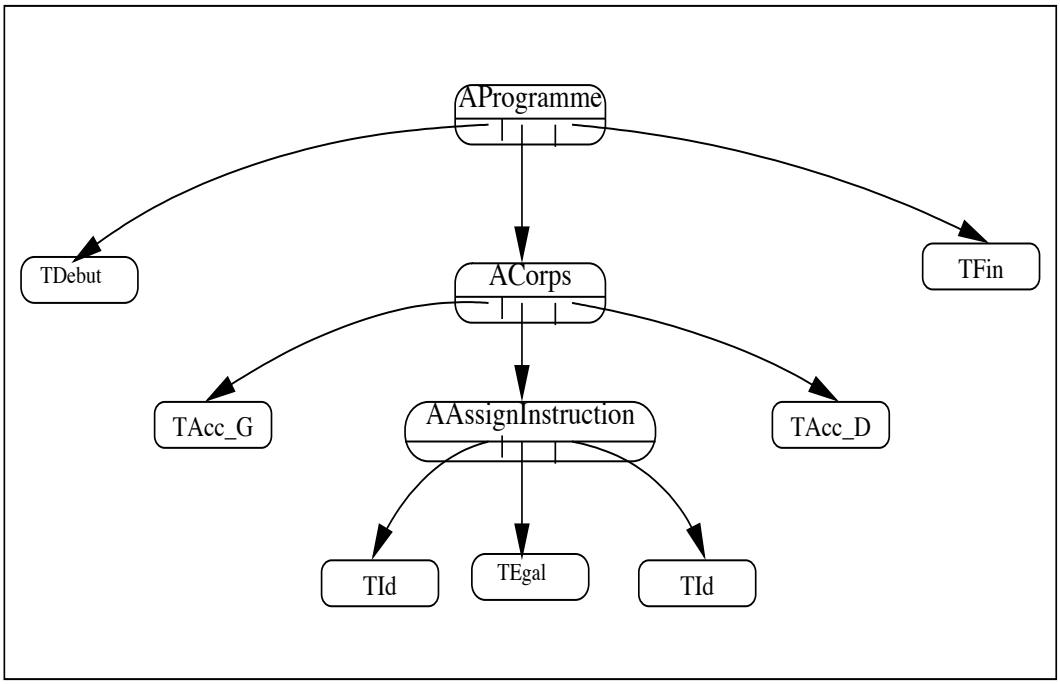

Figure 2.5 Arbre syntaxique "style-SableCC" de la phrase `DEBUT ACC_G ID EGAL ID ACC_D FIN`

2.3.4.3 Les différents types de parcours de l'arbre syntaxique

SableCC définit deux types de parcours sur les *arbres syntaxiques*.

1. Parcours en profondeur normal (en traitant les fils d'un noeud, de la gauche vers la droite.
2. Parcours en profondeur inverse (en traitant les fils d'un noeud, de la droite vers la gauche).

Ainsi, en utilisant ces parcours pour faire un affichage préfixe (racine et ensuite enfants) de l'arbre de la figure 2.5, le premier parcours donnerait la chaîne :

(`AProgramme TDebut ACorps TAccG AAssignInstruction TId Tegal TId TAccD TFin`)

tandis que le deuxième parcours donne :

(`AProgramme TFin TCorps TAccD AAssignInstruction TId TEgal TId TAccG TDebut`).

SableCC fournit deux classes qui définissent ces deux types de parcours. Il s'agit de la classe `DepthFirstAdapter` pour le parcours en profondeur normal et de la classe `ReverseDepthFirstAdapter` pour le parcours en profondeur inverse. Ces deux classes servent à faire le parcours de l'*arbre syntaxique* résultant de l'analyse syntaxique. Elles peuvent être utilisées pour effectuer des actions diverses. Avant de donner la structure générale de ces classes, rappelons un peu la correspondance entre la *grammaire* et l'*arbre syntaxique*.

2.3.4.4 Correspondance entre la grammaire et l'arbre syntaxique

Étant donné une *grammaire* G, nous avons montré d'une part qu'une phrase du langage correspond forcément à une séquence de dérivation particulière du symbole de départ et, d'autre part, qu'il y avait une correspondance directe entre l'*arbre syntaxique* et cette dérivation. Par ailleurs, une dérivation étant composée de symboles non terminaux et de symboles terminaux, il s'avère donc que l'*arbre syntaxique* est uniquement composée de noeuds créés à partir de symboles terminaux et non terminaux de la *grammaire*. Plus exactement, l'*arbre syntaxique* est composé de noeuds créés uniquement à partir d'*alternatives* de *productions* de la *grammaire* et de *jetons*. Donc pour une *grammaire* donnée, les différents types de noeuds pouvant faire partie de l'*arbre syntaxique* constituent un ensemble fini et les éléments de cet ensemble peuvent être tous déduits de cette même *grammaire*[7]. Il apparaît donc que pour définir un parcours sur

[7]Ici, il s'agit non seulement de la *grammaire* mais aussi des unités lexicales.

un *arbre syntaxique* issu du langage décrit par cette *grammaire*, il suffit de définir un parcours pour tous les noeuds possibles pouvant appartenir aux arbres syntaxiques des programmes de ce langage et ce en conformité avec la structure de la grammaire. Et c'est sur ce principe que se base SableCC pour créer les classes `DepthFirstAdapter` et `ReverseDepthFirstAdapter`. La définition des classes de parcours s'appuie sur le patron de conception visiteur présenté dans (Gamma et al., 1995) car il répond bien aux besoins de ce type de problème. SableCC utilise une version étendue de ce patron. Ce concept est plus élaboré par (Gagnon, 1998). Une description plus détaillée de certaines des classes de parcours de l'*arbre syntaxique* sera donnée dans les passages suivants.

2.3.4.5 Structure générale de la classe DepthFirstAdapter

Xxx représente une *alternative* de la *grammaire*

```
class DepthFirstAdapter ... {
      //Pour la racine de l'arbre
      void caseStart(Start node) {
           inStart(node) ;
           node.getAXxx().apply(this) ; //premier fils du noeud Start
                                node.getEOF().apply(this) ; //deuxi
           outStart(node) ;
      }
      void inStart(Start node) {
           defaultIn(node) ;
      }
      void outStart(Start node) {
           defaultOut(node) ;
      }
      //Pour les autres noeuds de l'arbre
      //La portion de code ci-dessous est dupliquée pour toutes les
      //alternatives de la grammaire
```

```
        void caseXxx(Xxx node) {
            //Avant le traitement des fils du noeud Xxx
            inXxx(node) ;
            node.getYyy().apply(this) ; //premier fils du noeud Xxx
            node.getZzz().apply(this) ; //premier fils du noeud Xxx
            //Apres le traitement des fils du noeud Xxx
            outXxx(node) ;
        }
        void inXxx(Xxx node) {
            defaultIn(node) ;
        }
        void outXxx(Xxx node) {
            defaultOut(node) ;
        }
...
        void defaultIn(Node node) {}
        void defaultOut(Node node) {}
        //Le code ci-dessous est aussi dupliquee pour tous les
        //jetons du fichier de specification
        void caseTToken1(TToken1 node) { }
}
```

Les méthodes `inXxx(...)` et `outXxx(...)` : le parcours proprement dit est effectué dans la méthode `caseXxx(...)`. En effet, dans cette méthode qui correspond au traitement d'un noeud particulier de l'*arbre syntaxique*, on va chercher les fils de ce noeud et on applique récursivement le parcours sur ces fils. On note aussi qu'une méthode `inXxx(...)` est appelée avant le traitement des noeuds fils et qu'une autre, `outXxx(...)`, est appelée après ce traitement.

La méthode `inXxx(...)` permet donc d'effectuer un traitement avant la visite des noeuds fils alors que la méthode `outXxx(...)` permet d'effectuer un traitement après

la visite des noeuds.

2.3.4.6 Classe de parcours pour la *grammaire* de la figure 2.3

La classe DepthFirstAdapter qui serait générée pour la *grammaire* de la figure 2.3, serait la suivante :

```
class DepthFirstAdapter ... {
    //Pour la racine de l'arbre
    void caseStart(Start node)
    {
       inStart(node) ;
       node.getPProgramme().apply(this) ; //premier fils du noeud Start
       node.getEOF().apply(this) ; //deuxieme fils du noeud Start
       outStart(node) ;
    }
    void inStart(Start node)
    { defaultIn(node) ; }
    void outStart(Start node)
    { defaultOut(node) ; }
    //Pour les autres noeuds de l'arbre
    //La portion de code ci-dessous est dupliqué pour
    toutes alternatives de la grammaire
    void caseAProgramme(AProgramme node) {
       in(node) ;
       //premier fils du noeud AProgramme
       node.getDebut().apply(this) ;
       //deuxieme fils du noeud AProgramme
       node.getCorps().apply(this) ;
       //troisieme fils du noeud AProgramme
```

```
        node.getFin().apply(this);
        outAProgramme(node);
    }
    void inAProgramme(AProgramme node)
    { defaultIn(node); }
    void outAProgramme(AProgramme node)
    { defaultOut(node); }

    void defaultIn(Node node) {}
    void defaultOut(Node node) {}
...
}
```

2.3.4.7 Utilisation des classes de parcours

Rappelons que les classes `DepthFirstAdapter` et `ReverseDepthFirstAdapter` définissent respectivement un parcours en profondeur et un parcours en profondeur inverse sur l'*arbre syntaxique*. Pour utiliser ces parcours afin d'accomplir une action particulière sur l'*arbre syntaxique*, il suffit de définir une sous-classe d'une de ces deux classes et de redéfinir les méthodes correspondant aux noeuds au niveau desquels on souhaite effectuer l'action. Pour illustrer le principe, prenons comme exemple l'*arbre syntaxique* de la figure 2.5 et supposons qu'on souhaite, lors d'un parcours en profondeur, faire un traitement (affichage) préfixe sur cet arbre. Rappelons qu'un traitement préfixe sur un sous-arbre de racine étiquetée `Root` ayant deux fils étiquetés `LeafA` et `LeafB` donnerait dans cet ordre : 1 - `traiter`[8](`Root`), 2 - `traiter`(`LeafA`), 3 - `traiter`(`LeafB`).
Disposant de la classe `DepthFirstAdapter` fournie par SableCC avec toutes les fonc-

[8]traiter(...) correspondrait à une méthode effectuant une action quelconque, par exemple un affichage.

tionnalités incluses, réaliser un tel affichage revient à faire les actions suivantes pour tous les noeuds correspondant à des *alternatives* de la *grammaire* :

- dans un premier temps, redéfinir la méthode `inXxx(...)` et y afficher la racine du sous-arbre enraciné ;
- et dans un second temps, redéfinir la méthode `outXxx(...)` et y afficher les noeuds fils qui sont des *jetons*. Seul les noeuds fils qui sont des *jetons* doivent être affichés car les autres noeuds fils seront visités lors de la récursion.

Le code de la classe effectuant le parcours préfixe est présenté ci-dessous :

```
class ParcoursPrefixe extends DepthFirstAdapter {
      void inAProgramme(AProgramme node)
      { System.out.println("AProgramme") ; }

      void outAProgramme(AProgramme node)
      { System.out.println("TBegin TEnd") ; }
      void inACorps(ACorps node)
      { System.out.println("ACorps") ; }

      void outACorps(ACorps node)
      { System.out.println("TAccG TAccD") ; }

      void inAAssignInstruction(AAssignInstruction node)
      { System.out.println("AAssignInstruction") ; }

      void outAAssignInstruction(AAssignInstruction node)
      { System.out.println("ID EGAL ID") ; }

      void inAComparaisonInstruction(AComparaisonInstruction node)
      { System.out.println("AComparaisonInstruction") ; }
```

```
        void outAComparaisonInstruction(AComparaisonInstruction node)
        { System.out.println("ID EGALEGAL ID") ; }
    }
```

En résumé, pour une *grammaire* donnée G, SableCC génère deux classes (`DepthFirstAdapter` et `ReverseDepthFirstAdapter`) effectuant le parcours en profondeur et le parcours en profondeur inverse des arbres syntaxiques correspondant aux phrases du langage décrit par cette *grammaire*. Pour effectuer des actions au niveau de ces arbres syntaxiques, il suffit de définir une sous-classe d'une de ces classes et mettre le code approprié dans la ou les méthode(s) correspondant au(x) noeud(s) de l'arbre sur lesquels on désire effectuer l'action.

2.4 Architecture interne de SableCC

Nous avons présenté jusque là le fonctionnement du logiciel SableCC à travers ses différentes entrées et sorties. Nous avons vu que l'entrée de SableCC est un fichier de spécification organisé sous forme de sections. La sortie, quant à elle, est un cadre de travail regroupant un *analyseur lexical*, un *analyseur syntaxique* et des classes pour la construction d'un *arbre syntaxique* et son parcours. Nous allons à présent décrire l'architecture interne du logiciel SableCC. Notre objectif est de montrer que le logiciel SableCC lui-même est formé de différents composants semblables à ceux qu'il génère.

2.4.1 SableCC, un compilateur de compilateurs

Comme nous l'avons précédemment mentionné, le logiciel SableCC est un environnement pour le développement des compilateurs et des interpréteurs. Il s'agit d'un compilateur de compilateurs (*Compiler Compiler* en anglais). En effet, suivant sa structure, le logiciel SableCC peut être vu comme étant un compilateur qui prend en entrée un programme qui est en réalité un fichier de spécification, et génère en sortie le cadre de travail précédemment décrit. Le fichier de spécification reçu par le logiciel SableCC

en entrée doit respecter un format bien précis. Quand ce fichier est donné en entrée au logiciel SableCC, celui-ci vérifie qu'il respecte un format imposé par des règles et génère les différents composants précédemment décrits. La vérification des règles à l'intérieur du logiciel est faite par un *analyseur syntaxique*. Cet *analyseur syntaxique* utilise au préalable un *analyseur lexical* qui découpe le fichier d'entrée en *unités lexicales* qui servent à réaliser l'analyse syntaxique. À la suite de l'analyse syntaxique, le programme en entrée se retrouve dans un *arbre syntaxique* à partir duquel les différents composants sont générés. À la lumière des ces informations, on comprend donc mieux pourquoi l'architecture interne de SableCC est identique à un certain point à celle des compilateurs ou interpréteurs qu'il permet de développer.

2.4.2 La grammaire des fichiers d'entrée de SableCC

SableCC reçoit en entrée des fichiers de spécification qui doivent respecter un certain format. La vérification du respect de ce format est assurée par l'*analyseur syntaxique* qui utilise des règles définies dans une *grammaire*. Cette *grammaire* se trouve dans un fichier. Appelons ce fichier, *fichier de spécification des fichiers de spécification de SableCC*. Ce fichier est défini suivant la même syntaxe que les fichiers de spécifications de SableCC, c'est-à-dire un fichier organisé sous forme de sections, dans laquelle la section `Productions` correspond à la *grammaire*. Le fichier des fichiers de spécification de SableCC est donc aussi un fichier de spécification de SableCC car il suit la même structure et a la même syntaxe. Le symbole de départ de cette *grammaire* est `grammar` et la *production* de ce symbole est :

```
grammar = P.package? P.helpers? P.states? P.tokens? ign_tokens?
P.productions? P.ast?;
```

Les différents éléments de l'*alternative* de cette *production* sont des symboles non terminaux. Ils désignent des *productions* définissant les différentes sections des fichiers de spécification de SableCC. La version complète de la *grammaire* et du fichier de spécification se trouve en Appendice A de ce mémoire. Dans le reste de ce mémoire, nous utiliserons le terme **méta-fichier de spécification de SableCC** pour désigner le fi-

chier de spécification des fichiers de spécification de SableCC.

2.5 Résumé

L'environnement SableCC a été présenté dans ce chapitre. Dans un premier temps, une brève description du logiciel a été donnée en montrant ses entrées et sorties. Dans les sorties, c'est-à-dire le code généré, l'accent a été principalement mis sur la composante concernant la construction de l'*arbre syntaxique* et son parcours. Il a également été montré comment pouvait être utilisée la composante de parcours de l'*arbre syntaxique* pour faire l'affichage préfixe sur l'*arbre syntaxique* d'un petit programme inventé. Cette mise en contexte assez complète sur SableCC permet d'aborder les différentes étapes du travail des transformations automatiques d'arbres syntaxiques en commençant par la syntaxe de spécification de ces transformations dans le prochain chapitre.

Chapitre III

SYNTAXE DES TRANSFORMATIONS D'ARBRES SYNTAXIQUES

Dans ce chapitre, la syntaxe pour spécifier les transformations d'arbres syntaxiques sera présentée. Après une mise en situation sur les transformations d'arbres syntaxiques, nous présenterons les modifications apportées au *méta-fichier de spécification* de SableCC pour prendre en compte ces transformations. Ensuite, nous verrons les différentes étapes nécessaires pour bénéficier des transformations d'arbres syntaxiques du point de vue de l'usager et enfin nous parlerons des différents types de transformations. Nous nous servirons d'exemples de grammaire ou d'extraits de grammaire pour décrire les différents types de transformations.

3.1 Mise en situation

Les transformations d'arbres syntaxiques de SableCC ont pour objectif de permettre la construction d'un arbre syntaxique abstrait. Un arbre syntaxique abstrait n'a pas nécessairement la même structure que celle induite par la grammaire du langage et ne retient pas nécessairement tous les mots du programme d'entrée. Nous savons que les analyseurs syntaxiques générés par SableCC construisent automatiquement un arbre syntaxique lorsqu'ils analysent un programme donné. La structure de ces arbres syntaxiques est déduite automatiquement de la grammaire du langage par une dérivation particulière du symbole de départ (Aho, Sethi et Ullman, 2000). Les arbres ainsi produits reflètent de façon exacte la grammaire et retiennent, comme déjà mentionné

dans le chapitre 2, tous les mots du langage à l'exception de ceux correspondant à des jetons ignorés. Étant donné qu'un arbre syntaxique abstrait pourrait ne pas contenir certains mots et ne pas avoir la même structure que la grammaire, il convient alors pour construire de tels arbres d'indiquer leur structure et la manière dont ils doivent être construits.

Les arbres syntaxiques abstraits sont toujours construits à partir du programme analysé, et ce programme fait partie du langage décrit par la grammaire. Pour construire l'arbre syntaxique abstrait, il faut donc récupérer les mots ou certains mots du programme d'entrée et les inclure dans la nouvelle structure décrivant cet arbre. L'ajout des transformations d'arbres syntaxiques de SableCC a donc consisté à mettre en place l'infrastructure nécessaire pour accomplir cette tâche. Concrètement, une nouvelle section a été ajoutée aux fichiers de spécification pour décrire la structure de l'arbre syntaxique et la section `Productions` existante a été modifiée pour y inclure une syntaxe permettant une construction d'un arbre syntaxique abstrait à partir du programme analysé. La nouvelle section rajoutée s'appelle `Abstract Syntax Tree`. Ces changements ont bien entendu été faits au niveau du *méta-fichier de spécification* de SableCC pour permettre la prise en charge par le logiciel des nouveaux types de fichier de spécification.

3.2 Les modifications au méta-fichier de spécification de SableCC

Nous avons déjà vu que les transformations d'arbres syntaxiques doivent être indiquées dans le fichier de spécification à l'aide d'une nouvelle syntaxe. Nous avons aussi vu dans le chapitre 2 que c'est la grammaire du *méta-fichier de spécification* de SableCC qui définit la syntaxe à utiliser pour les fichiers de spécification. Dès lors qu'une nouvelle syntaxe a été rajoutée aux fichiers de spécification pour pouvoir indiquer les transformations d'arbres syntaxiques, il a fallu modifier le *méta-fichier de spécification* de telle sorte que soit supportée cette nouvelle syntaxe. Essentiellement, les modifications apportées au *méta-fichier de spécification* concernent l'ajout de nouveaux jetons dans la section `Tokens` et l'ajout de nouvelles productions dans la section `Productions` (cela correspond à un grossissement de la grammaire). Ces modifications ont pour but

de décrire la nouvelle section `Abstract Syntax Tree` mais aussi la syntaxe de transformation de la section `Productions` à présent adoptée dans les fichiers de spécification de SableCC.

3.2.1 La nouvelle section Abstract Syntax Tree

Dans le *méta-fichier de spécification*, la nouvelle section `Abstract Syntax Tree` est décrite par de nouvelles productions dont la principale est la production de nom `ast`. Cette production principale permet de décrire une section qui contient une grammaire. La syntaxe utilisée pour cette grammaire est la même que celle de la grammaire de la section `Productions`, c'est à dire une notation "*EBNF*" style-SableCC. La différence entre ces deux sections tient au fait que la grammaire de cette nouvelle section sert uniquement à construire l'arbre syntaxique. En effet, c'est à partir des différentes productions et alternatives de cette section que seront déduites les différents types de données servant à créer les noeuds de l'arbre syntaxique. Un extrait du *méta-fichier de spécification* montrant certaines des nouvelles productions ajoutées pour décrire la section `Abstract Syntax Tree` est présentée à la figure 3.1.

La particularité de cette section tient au fait que les différents symboles grammaticaux[1] pouvant y apparaître doivent correspondre à des productions définies dans cette même section. Ainsi, si par exemple `prod = symb_term symb_non_term ;` est une production de cette section, et que `symb_term` est un symbole terminal et `symb_non_term` un symbole non terminal, il faut que `symb_non_term` soit une production définie dans cette même section.

3.2.2 Modification de la définition d'une production

Diverses modifications ont été apportées à la définition d'une production. Elles visent principalement à permettre la construction de l'arbre syntaxique abstrait à partir

[1]symbole non terminal

```
...
Productions
...

grammar        =
               P.package?  P.helpers?  P.states?  P.tokens
               ign_tokens? P.productions?   P.ast?;
ast            =
               abstract syntax tree [prods]: ast_prod+;
ast_prod       =
               id equal [alts]:ast_alts semicolon;
ast_alts       =
               ast_alt [ast_alts]:ast_alts_tail*;
ast_alts_tail  =
               bar ast_alt;
ast_alt        =
               alt_name? [elems]:elem*;
...
```

Figure 3.1 Extrait du méta-fichier de spécification montrant le symbole `ast` rajouté à la production `grammar` et les productions subséquentes

du programme analysé. Nous savons maintenant (chapitre 2) que l'analyse syntaxique est faite à partir de la grammaire de la section `Productions` et que la construction de l'arbre syntaxique abstrait est essentiellement basée sur la section `Abstract Syntax Tree`. Nous savons aussi que la construction de l'arbre syntaxique se fait lors de l'analyse syntaxique. La grammaire de la section `Productions` devient donc la grammaire d'analyse syntaxique et celle de la section *Abstract Syntax Tree*, la grammaire de construction de l'arbre syntaxique. La structure de l'arbre syntaxique abstrait n'étant plus déduite de la grammaire d'analyse syntaxique, il faut donc un mécanisme permettant la construction de l'arbre lors de l'analyse syntaxique. C'est donc dans cette optique que se situent les transformations introduites dans cette section. Les transformations mises en place au niveau des productions sont de deux sortes. Dans un premier temps, pour chaque production, il faut définir le ou les noeud(s) éventuel(s) de l'arbre syntaxique à construire. Ensuite, étant donné que le langage est en réalité décrit par

les alternatives des productions, il faut indiquer pour chacune des alternatives de ces productions la manière de construire ce(s) noeud(s) là. Pour ce faire, nous avons mis en place une transformation pour les productions et une autre pour les alternatives.

3.2.2.1 Les transformations de production

La construction de l'arbre syntaxique est un processus progressif qui se fait lors de l'analyse syntaxique (Gagnon et Hendren, 1998). Pendant cette construction, on ne dispose pas nécessairement d'un arbre mais d'une forêt d'arbres. Ce n'est qu'à la fin de ce processus que l'arbre syntaxique final est construit et retourné par l'analyseur syntaxique. La transformation de production indique donc les différents noeuds à construire ou à retenir pour la production actuelle. Cette transformation se présente sous la forme d'une liste de noeuds. Elle est exprimée de la manière suivante :

$$\{\rightarrow \texttt{prod\_transform}_1[*/?/+]\ \texttt{prod\_transform}_2[*/?/+]\ \ldots\ \texttt{prod\_transform}_N[*/?/+]\}$$

où $\texttt{prod\_transform}_{1,2,\ldots,N}$ est soit un symbole terminal, soit un symbole non terminal. Dans le cas d'un symbole non terminal, ce symbole doit être défini dans la section `Abstract Syntax Tree`. La nouvelle production `prod_tranform` a été ajoutée à la grammaire du méta-fichier de spécification de SableCC pour permettre de supporter cette nouvelle syntaxe au niveau des fichiers de spécification. Un extrait du *méta-fichier de spécification* montrant cette production est presentée à la figure 3.2

3.2.2.2 Les transformations d'alternative

À ce niveau, le travail consiste à construire ou à retenir pour chaque alternative les différents éléments susceptibles d'être inclus dans l'arbre syntaxique en construction en conformité avec la transformation de production. En effet, les noeuds construits à ce niveau-ci doivent correspondre en nombre et en type à ceux de la transformation de production. Nous reviendrons plus longuement sur cet aspect dans le chapitre consacré aux vérifications sémantiques à effectuer sur les transformations d'arbres syntaxiques. La transformation d'alternative se présente sous la forme suivante :

```
...
Productions
...

prod              =
                  id  prod_transform?   equal alts semicolon;
prod_transform    =
                  l_brace arrow[elems]:elem* r_brace;
elem              =
                  elem_name?  specifier? id  un_op?;
...
```

Figure 3.2 Extrait du méta-fichier de spécification de SableCC montrant le symbole ajouté à la production `prod` pour permettre la spécification des transformations de production

$\{\rightarrow$ `alt_transform`$_1$ `alt_transform`$_2$ ... `alt_transform`$_N\}$.

Au niveau du *méta-fichier de spécification*, la production `alt_transform` est la principale d'une série de plusieurs productions qui ont été rajoutées pour la prise en compte de cette syntaxe au niveau des fichiers de spécification de SableCC. La figure 3.3 montre un extrait du méta-fichier avec la production `alt_transform`.

3.3 Etapes nécessaires pour les transformations d'arbres syntaxiques avec SableCC

Pour bénéficier des transformations d'arbres syntaxiques, il faut d'abord définir une section `Abstract Syntax Tree` dans laquelle on spécifie la structure de l'arbre syntaxique et ensuite rajouter des transformations aux productions de la section `Productions` pour permettre que l'arbre syntaxique construit après analyse syntaxique du programme soit conforme à la section `Abstract Syntax Tree`. Comme déjà mentionné, les transformations à ajouter aux productions de la section Productions sont de deux ordres, les transformations de production et les transformations d'alternatives associées à cette production.

```
...
Productions
...

alt                 =
                    alt_name? [elems]:elem*  alt_transform?;
alt_transform       = l_brace arrow[terms]:term r_brace

term                =
                    {new} new prod_name l_par params? r_par |
...                 {list} l_bkt list_of_list_term? r_bkt   |
                    {simple} specifier? id simple_term_tail?  |
                    {null} null                               ;
list_of_list_term   =
                    list_term [list_terms]:list_term_tail*;
list_term           =
                    {new} new prod_name l_par params? r_par |
                    {simple} specifier? id simple_term_tail?  ;
list_term_tail      =
                    comme list_term;
...
```

Figure 3.3 Extrait du méta-fichier de spécification de SableCC montrant le symbole ajouté à la production `alt` pour permettre la spécification des transformations d'alternatives

3.3.1 La section Abstract Syntax Tree

Dans cette section, on doit retrouver la structure générale qu'on souhaite donner au nouvel arbre syntaxique. Cette structure est exprimée sous forme de productions. Les alternatives de ces productions correspondent à des noeuds de l'arbre syntaxique et les différents éléments de ces alternatives aux fils de ces noeuds. C'est à partir de cette section que SableCC génère le cadre de travail permettant de construire l'arbre syntaxique. La figure 3.4 montre un exemple d'une section `Abstract Syntax Tree` qui va servir à représenter des expressions arithmétiques simples.

Il est intéressant de s'attarder quelques instants sur la production `exp` de cette

```
Abstract Syntax Tree

        grammaire = exp*;
        exp =
            {plus}      [g]:exp [d]:exp   |
            {moins}     [g]:exp [d]:exp   |
            {div}       [d]:exp [d]:exp   |
            {mult}      [g]:exp [d]:exp   |
            {nombre}    nombre            ;
```

Figure 3.4 Section `Abstract Syntax Tree` d'un fichier de spécification de SableCC

section représentée à la figure 3.4. Cette production comporte cinq alternatives qui peuvent chacune représenter des noeuds de l'arbre syntaxique. Les principaux noeuds correspondent aux opérations arithmétiques simples habituelles que l'on connaît avec les opérateurs `+`, `-`, `*` et `/`. Cependant les alternatives correspondant à ces opérations n'incluent pas ces opérateurs. En effet, l'alternative `plus` de la production `exp` n'est constituée que de deux éléments qui correspondent en fait aux opérandes de l'opérateur arithmétique binaire `+`. Ceci est dû au fait que l'arbre syntaxique, par sa structure, renferme déjà assez d'informations et la présence de l'opérateur n'est donc pas nécessaire.

3.3.2 La transformation de production

Une transformation de production indique ce qui doit être retenu de la production, plus précisément la portion du programme d'entrée qui pourrait être incluse dans l'arbre syntaxique. Elle intervient au niveau de la section `Productions`. Elle permet d'indiquer globalement ce qui devrait être retenu de la production. Elle doit ensuite être complétée par les transformations d'alternatives. La transformation de production doit être spécifiée juste après le nom de la production. Les différents éléments pouvant la constituer doivent être soit des symboles non terminaux correspondant à des productions définies dans la section `Abstract Syntax Tree` soit des symboles terminaux[2].

[2]Rappel : tous les symboles terminaux sont définis dans la section `Tokens`.

La syntaxe des transformations de productions incluent aussi les opérateurs *****, + et **?** qui peuvent optionnellement s'ajouter aux différents éléments de la transformation de production. Nous allons maintenant indiquer la signification de ces opérateurs :

l'opérateur `?` : il sert à indiquer que l'élément à retenir de la production pourrait être absent. Dans ce cas, quand on essaie d'atteindre le noeud dans l'arbre syntaxique, ce dernier correspond à une référence vers un objet *null* ;

l'opérateur `+` : il sert à indiquer qu'il s'agit d'une occurrence *multiple* du symbole qui le précède. La transformation d'alternative associée devrait donc refléter cela. La liste dans ce cas-ci ne peut pas être vide ;

l'opérateur `*` : de la même manière que l'opérateur `+`, il permet de spécifier une liste. La seule différence tient au fait que la liste peut être vide dans ce cas-ci.

Voici quelques exemples de transformation de production :

```
grammaire {→ exp*} = ... ;
exp       {→ exp } = ... ;
facteur   {→ exp } = ... ;
```

Il est évident que seul un extrait des transformations est montré dans les exemples ci-dessus. Nous allons nous servir d'une simple grammaire permettant de spécifier des expressions arithmétiques simples et la transformer pour construire un arbre syntaxique abstrait. La grammaire en question est presentée à la figure 3.5

L'objectif de la transformation est de pouvoir construire un arbre syntaxique abstrait qui serait conforme à la grammaire de la section `Abstract Syntax Tree` de la figure 3.4 à partir d'expressions arithmétiques spécifiées avec la grammaire de la figure 3.5.

Les transformations de production à opérer sur les différentes productions de la section `Productions` de la figure 3.5 seraient les suivantes :

```
grammaire        {→ grammaire } = ... ;
liste_exp        {→ exp* }      = ... ;
suite_liste_exp  {→ exp}        = ... ;
```

Tokens	**Productions**
par_g = ´(´ ; par_d = ´)´ ; plus = ´+´; moins = ´–´; mult = ´*´ ; div = ´/´ ; virg = ´,´; blanc = (´ ´ | 13 | 10)+ ; nombre = [0 .. 9]+; **Ignored Tokens** blanc ;	grammaire = liste_exp ; liste_exp = exp suite_liste_exp* ; suite_liste_exp = virg exp ; exp = {plus} exp plus facteur | {moins} exp moins facteur | {facteur} facteur ; facteur = {mult} facteur mult terme | {div} facteur div terme | {terme} terme ; terme = {nombre} nombre | {exp} par_g exp par_d ;

Figure 3.5 Section `Productions` de la grammaire d'expressions arithmétiques "simples"

```
exp          {→ exp }     = ... ;
facteur      {→ exp }     = ... ;
terme        {→ exp }     = ... ;
```

Les productions se transforment toutes soit en un symbole `grammaire`, soit en un symbole `exp` ou encore en une liste de symboles de type `exp` c'est-à-dire `exp*`. Il faut rappeler que les éléments des transformations de production, s'ils sont des symboles non terminaux, doivent correspondre à des productions définies dans la section `Abstract Syntax Tree`. `grammaire` et `exp` étant deux productions définies dans cette section, les transformations ci-dessus sont donc valides. Une fois les transformations de productions en place, il convient maintenant de définir des transformations d'alternatives de manière

appropriée pour permettre la construction de l'arbre syntaxique.

3.3.3 La transformation d'alternative

C'est la transformation d'alternative qui définit en réalité le type de construction à réaliser pour l'arbre syntaxique abstrait. En effet, cet arbre syntaxique étant construit à partir du programme d'entrée et la structure du programme étant représentée par la grammaire, plus précisément les alternatives des productions, c'est donc au niveau des alternatives de ces productions qu'il convient de spécifier ce qu'il faut retenir dans l'arbre syntaxique abstrait. Pour ce faire, nous avons défini cinq types de transformations au niveau des alternatives pour permettre de construire l'arbre syntaxique abstrait :

- la rétention d'éléments déjà existant dans la forêt d'arbres de l'arbre syntaxique en construction : ***simple*** ;
- la création de nouveaux noeuds de l'arbre syntaxique en construction à partir de noeuds déjà existants ou de jetons (tokens) : ***New*** ;
- la création de liste de noeuds (alternatives) ou de feuilles (jetons) : **list** ;
- l'élimination d'un certain type de noeuds : ***vide*** ;
- un joker pour le remplacement d'un symbole en cas de pénurie ou son élimination : ***Null***.

Nous allons maintenant présenter les transformations d'alternatives individuellement. Nous nous appuierons sur des exemples concrets pour permettre une meilleure compréhension. Nous allons nous servir comme base d'une grammaire sans aucune transformation et allons y insérer progressivement des transformations pour faire en sorte que l'arbre syntaxique construit soit un arbre syntaxique abstrait.

3.3.3.1 La retention d'éléments déjà existants dans la forêt d'arbres de l'arbre syntaxique en construction : *simple*

Ce type de transformation est la plus simple possible qu'on puisse avoir. En effet, son rôle consiste tout simplement à retenir un noeud tel quel dans une alternative et à l'ajouter aux noeuds déjà présents dans la forêt d'arbres. La syntaxe est la suivante :

```
ast_elem[.transform]
```

Il faut garder à l'esprit qu'il s'agit de transformation d'alternative. Dans ce cas, `ast_elem` est un élément de l'alternative courante. La partie optionnelle (`[.transform]`) n'est applicable que dans le cas où `ast_elem` désigne un symbole non terminal. En effet, `[.transform]` fait référence à la transformation du symbole `ast_elem`. En d'autres mots, si `ast_elem.transform` est une transformation d'alternative, il faut que `ast_elem` corresponde à une production dont la transformation de production comporte un élément dont le nom ou le type est `transform`.

Exemple 3.1

```
suite_list_exp                          {→ exp }        =
               virg exp                 {→ exp}          ;
...
terme                                   {→ exp }        =
               {exp} l_par exp r_par    { → exp}         ;

facteur                                 {→ exp }        =
               {terme} terme            {→ terme.exp}    ;
```

Dans l'exemple ci-dessus, pour l'alternative de la production *facteur*, la transformation d'alternative `terme.exp` désigne un élément de type `exp`. En effet, `terme` est un symbole non terminal designant la production de même nom dans la section `Productions`. Comme on peut le remarquer dans cet exemple, la production `terme` se transforme en un symbole `exp`, ce qui en fait un élément dont le type est `exp`.
Les autres exemples (production `suite_liste_exp` et `terme`) sont assez simples dans le sens que les transformations d'alternatives ne font que retenir un élément déjà présent dans l'alternative. Les autres éléments de l'alternative sont de ce fait ignorés.

Exemple 3.2

```
suite_list_exp                          {→ exp }                =
```

```
        virg exp                        {→ exp}                 ;
...
terme                                   {→ [une_exp] :exp }     =
        {exp} l_par exp r_par           {→ exp}                 ;

facteur                                 {→ exp }                =
        {terme} [un_terme] :terme       {→ un_terme.une_exp}    ;
```

Ce deuxième exemple constitue une réécriture de l'exemple 3.1 dans laquelle nous avons rajouté un nom à l'élément `exp` (`[une_exp] :exp`) de la transformation de la production `terme` et à l'élément `terme` (`[un_terme] :terme`) de l'alternative de la production `facteur`. Il a pour but d'illustrer à quel point une transformation d'alternative de type `simple` peut être relativement complexe. Comme mentionné précédemment, `un_terme.une_exp` est valide comme transformation d'alternative si et seulement si `un_terme` correspond à un symbole non terminal et que `une_exp` correspond à un des éléments de la transformation de la production correspondant à ce symbole non terminal. C'est le cas ici car `un_terme` est le nom donné à un élément d'alternative qui correpond au symbole non terminal `terme` et `une_exp` est le nom donné à un élément de la transformation de la production `terme` correspondant à ce non terminal.

3.3.3.2 La création de nouveaux noeuds : *New*

Le but de ce type de transformation est de créer un nouveau noeud. Le type de ce noeud doit correspondre à une des alternatives de la section `Abstract Syntax Tree` car c'est de cette section que sont déduits les noeuds devant être inclus dans l'arbre syntaxique à construire. La syntaxe de la transformation *New* est la suivante :

`New ast_production[.alternative] ( param`$_1$`, param`$_2$`, ..., param`$_N$ `)`

`ast_production` doit être une production de la section `Abstract Syntax Tree`. `alternative` correspond au nom donné à une des alternatives de cette production s'il y a lieu. Si l'alternative n'a pas de nom, la partie correspondant au nom de l'alternative

dans la création du noeud devient optionnel. C'est pour cela que dans la syntaxe ci-dessus, `.alternative` est placée entre crochets (`[ ]`).
$param_{1,2,...,N}$ correspondent aux paramètres nécessaires pour la construction du noeud. Ces paramètres doivent obéir à deux règles à savoir :

1. ils doivent correspondre en nombre et en type aux éléments de l'alternative de la production `ast_production` ;
2. les paramètres $param_{1,2,...,N}$ doivent être des transformations d'alternatives à leur tour. Cela signifie qu'on peut avoir une transformation `New` imbriquée dans une autre transformation `New` dans le genre :

   `New ast_production.alternative( New autre_ast_production.autre_alternative(...), ..., paramN).`

Le plus souvent, les transformations de type *New* sont combinées avec les transformations de type "simple", comme le montre l'exemple ci-après.

Exemple 3.3

```
exp                     {→ exp }        =
    {plus} exp plus facteur
                        {→ New exp.plus(exp, facteur.exp)} ;
...
terme                   {→ exp }        =
    {nombre} nombre     {→ New exp.nombre(nombre)} ;

facteur                 {→ exp }        =
    {mult} facteur mult terme
                        {→ New exp.facteur(facteur.exp, terme.exp)} ;
```

Dans l'exemple ci-dessus, on peut se rendre compte que les transformations d'alternative de type *New* ont été combinées avec des transformations de type *simple*. Décortiquons un peu l'exemple de la production *terme*, c'est-à-dire la transformation d'alternative `{→ New exp.nombre(nombre)}`. La transformation `exp.nombre` désigne

l'alternative `{nombre}` de la production `exp` dans la section *Abastract Syntax Tree*. En effet, cette alternative comporte un seul élément qui est `nombre`. C'est donc la raison pour laquelle le paramètre passé pour la construction du noeud est `nombre`, c'est-à-dire l'élément `nombre` dans l'alternative.

3.3.3.3 La création de liste de noeuds : *list*

Ce type de transformation permet de regrouper ensemble des éléments du même type dans une liste chainée. La syntaxe est la suivante :

`[` $list\_elem_1$ `,` $list\_elem_2$ `, ...,` $list\_elem_N$ `]`

Les éléments de la liste sont à leur tour des transformations d'alternative. Ils peuvent être soient des transformations de type "`simple`" ou encore de type "`New`". Tous les éléments de cette liste doivent avoir le même type, ce qui signifie qu'ils doivent représenter le même jeton (token) ou encore être du type de la même production.

Exemple 3.4

```
liste_exp                        {→ exp* }        =
          exp suite_liste_exp*   {→ [exp, suite_liste_exp.exp] ;
...
suite_liste_exp                  {→ exp }         =
          virg exp               {→ exp} ;
```

L'exemple ci-dessus de la transformation de type *"list"* regroupe les éléments de type `exp` de l'alternative de la production `liste_exp`. En effet, cette alternative par sa structure permet d'écrire "`exp ,exp ,exp ,... ,exp`", c'est-à-dire une série d'expressions séparées par des virgules. La transformation de type "`simple`" `suite_liste_exp.exp` correspond à une liste de symboles de type `exp` car dans l'alternative de la production `liste_exp`, l'élément `suite_liste_exp` est suivi d'un opérateur *, ce qui en fait une liste. La transformation `{→ [exp, suite_liste_exp.exp]` permet donc de regrouper le premier élément de type `exp` avec la séquence des éléments `exp` et ce pour en faire

une liste.

3.3.3.4 L'élimination d'un certain type de noeud : *vide*

Cette transformation permet d'éliminer complètement une production. En effet, elle peut être utile si on veut se débarasser d'un certain nombre de constructions de la grammaire. La syntaxe est la suivante :

{→}

Étant donné que les transformations d'alternatives doivent être conformes aux transformations de productions, les transformations de productions correspondantes doivent aussi être **{→ }**. Dans la grammaire de la figure 3.5, il n'y a pas de productions que l'on souhaiterait éliminer donc notre exemple sera général.

Exemple 3.5

```
production          {→ } =
            {alt1} ... {→ }   |
            {alt2} ... {→ }   |
            ...
            {altN} ... {→ }   ;
```

3.3.3.5 Un joker pour le remplacement d'un symbole en cas de pénurie ou son élimination : *Null*

Ce type de transformation est un joker parce qu'il permet de substituer `Null` à un élément manquant. Il peut aussi être utilisé quand on désire annuler l'effet d'un élément. Prenons un exemple pour montrer ceci.

Exemple 3.6

```
terme                        {→ exp? }        =
      {nombre} nombre        {→ New exp.nombre(nombre) } |
      {exp} l_par exp r_par  {→ Null}
```

Productions

```
grammaire =
        liste_exp                       {–> New grammaire( [liste_exp.exp] )  } ;

liste_exp                               {–> exp* } =
        exp suite_liste_exp*            {–> [exp, suite_liste_exp.exp] } ;

suite_liste_exp                         {–> exp } =
        virg exp                        {–> exp } ;

exp =
        {plus} exp plus facteur         {–> New exp.plus(exp, facteur.exp) }     |
        {moins} exp moins facteur       {–> New exp.moins(exp, facteur.exp)  } |
        {facteur} facteur               {–> facteur.exp }                        ;

facteur                                 {–> exp } =
        {mult} facteur mult terme       {–> New exp.mult(facteur.exp, terme.exp) }  |
        {div} facteur div terme         {–> New exp.div(facteur.exp, terme.exp)  }  |
        {terme} terme                   {–> terme.exp                                ;

terme                                   {–> exp } =
        {nombre}  nombre                {–> New exp.nombre( nombre ) }   |
        {exp}  l_par exp r_par          {–> exp }                        ;
```

Abstract Syntax Tree

```
grammaire = exp*;
exp =
    {plus}      [g]:exp [d]:exp   |
    {moins}     [g]:exp [d]:exp   |
    {div}       [d]:exp [d]:exp   |
    {mult}      [g]:exp [d]:exp   |
    {nombre} nombre               ;
```

Figure 3.6 Section `Productions` et `Abstract Syntax Tree` de la grammaire d'expressions arithmétiques "simples" après l' ajout des transformations d'arbres syntaxiques

Dans le cas de l'exemple ci-dessus, la transformation de l'alternative `exp` renvoie un élément `Null`. En écrivant `terme.exp`, dépendemment de l'alternative de la production terme qui est réalisée, cette transformation peut désigner un noeud qui est une référence vers un objet `null`. Dans un tel cas, rien n'est inséré dans la forêt d'arbres en construction pour cette alternative. Ce type de transformation est différent de la transformation `vide` pour deux raisons. D'une part parce que son incidence est décidée au niveau de la transformation d'alternative et non de la transformation de production. D'autre part, dans le cas de la transformation `vide`, la production et donc toutes ses alternatives sont entièrement éliminées alors que dans le cas de la transformation `Null`, seule une partie des alternatives de cette production est éliminée (voir Exemple 3.7).

Exemple 3.7

```
facteur                    { → exp }        =
        {mult} facteur mult terme
                           {→ New exp.facteur(facteur.exp, Null) }  ;
```

Les sections `Productions` et `Abstract Syntax Tree` complètes résultant de la transformation d'arbres syntaxiques de la grammaire de la figure 3.5 sont présentées à la figure 3.6

3.4 Résumé

Nous avons présenté dans ce chapitre la syntaxe de définition des arbres syntaxiques abstraits. Nous avons dans un premier temps montré les modifications apportées au *méta-fichier de spécification* de SableCC notamment à la grammaire de SableCC pour permettre la prise en charge de la nouvelle syntaxe par le logiciel. Ensuite, nous avons présenté les différents types de transformations existantes et nous les avons illustrées à l'aide de divers exemples. Nous espérons que ce chapitre pourra servir de base à ceux qui désirent utiliser les transformations d'arbres syntaxiques de SableCC.

Chapitre IV

LES VÉRIFICATIONS SÉMANTIQUES EFFECTUÉES SUR LES TRANSFORMATIONS D'ARBRES SYNTAXIQUES

Dans ce troisième chapitre, les vérifications sémantiques faites sur les transformations d'arbres syntaxiques de SableCC sont introduites. Après une brève description de l'objectif de ces vérifications, les différentes vérifications sémantiques faites seront décrites en détail. Cette description est faite à travers les sections du *méta-fichier de spécification* sur lesquelles elles interviennent. Il s'agit plus précisément des sections `Productions` et `Abstract Syntax Tree`. Dans la dernière partie de ce chapitre, nous parlerons de la concordance de types qui est un aspect important des vérifications sémantiques comme nous le verrons plus tard ainsi que la vérification stricte de types effectuée en raison de la multiplicité[1] de certains opérateurs.

4.1 Objectif des vérifications sémantiques

Les vérifications sémantiques permettent à SableCC de garantir la validité des transformations d'arbres syntaxiques spécifiées par l'usager et de construire ainsi un arbre syntaxique valide conformément à ces transformations. Outre le souci d'éviter à l'usager une perte cruciale du temps en cas d'erreur, elles permettent de garantir l'intégrité de l'arbre créé dans le cadre défini par le logiciel. Les vérifications sémantiques effectuées permettent d'assurer le respect d'un certain nombre de règles dont :

[1] Présence des opérateurs `*`, `+` et `?` devant les symboles de la grammaire.

- la non-création de DAG ("*Direct Acyclic Graph*", c'est-à-dire graphe orienté sans cycle) qui garantit par la même occasion que le résultat de la transformation donne bien un arbre à la fin de l'analyse syntaxique ;
- la garantie que la racine de l'arbre sera un noeud créé à partir d'une des alternatives de la première production de la section `Abstract Syntax Tree` ;
- la non création et non insertion d'une liste hétérogène dans l'arbre résultant. Une liste hétérogène est une liste dont les éléments ne sont pas du même type, c'est à dire des noeuds construits à partir d'alternatives de différentes productions ou des jetons différents ;
- l'intégrité (ou non corruption) de l'arbre résultant, autrement dit, il y a une correspondance stricte entre l'arbre créé et les transformations spécifiées dans la section `Productions`. Aussi les noeuds et feuilles de l'arbre doivent absolument être construits en se basant sur les productions de la section `Abstract Syntax Tree`, plus précisément les alternatives de ces productions et les jetons de la grammaire.

La vérification du respect de ces propriétés est en grande partie faite au niveau de la section `Productions`, et ce, en étroite collaboration avec la section `Abstract Syntax Tree`.

4.1.1 Mécanisme utilisé pour effectuer les vérifications sémantiques

Nous avons vu dans le chapitre 2 que pour une grammaire donnée, SableCC génère un cadre de travail qui permet, entre autres, de construire un arbre syntaxique pour les programmes compatibles avec cette grammaire après analyse syntaxique de ce programme. Nous avons aussi vu que le logiciel SableCC lui-même est constitué d'un cadre de travail semblable à ceux qu'il génère pour les fichiers de spécification qu'il reçoit en entrée. Toujours en se référant au chapitre 2, après analyse lexicale et syntaxique d'un programme d'entrée par SableCC, ce programme se retrouve à peu de choses près dans l'arbre syntaxique résultant de l'analyse syntaxique. Les programmes d'entrée de SableCC étant des fichiers de spécification, et ces fichiers de spécification étant com-

posés de sections, ces sections se retrouvent donc dans l'arbre syntaxique résultant de l'analyse syntaxique par SableCC. Pour effectuer les vérifications sémantiques sur les sections `Abstract Syntax Tree` et `Productions` du fichier de spécification, les mécanismes disponibles dans le cadre de travail généré ont donc été utilisés, notamment, des traverseurs d'arbres (visiteurs) ont été définis. Ceux-ci parcourent l'arbre syntaxique et appliquent les règles induites par les vérifications sémantiques à la grammaire.
Nous avons vu dans le chapitre 2 que la définition d'un visiteur avec SableCC se fait en définissant une sous-classe de la classe `DepthFirstAdapter` ou `ReverseDepthFirstAdapter`. C'est ainsi qu'a été défini le visiteur `ResolveAstIds` qui effectue les vérifications sémantiques pour la section `Abstract Syntax Tree`. Cette classe se trouve dans le fichier `ResolveAstIds.java`. Pour les vérifications sémantiques de la section `Productions`, deux visiteurs ont été définis. Le premier, `ResolveProdTransformIds` s'intéresse aux vérifications sémantiques de la partie `prod_transform` d'une production, autrement dit, les transformations des productions et le deuxième, `ResolveTransformIds`, fait les vérifications pour la partie `alt_transform`, c'est à dire les transformations des alternatives de cette même production. Ces deux visiteurs sont respectivement dans les fichiers `ResolveProdTransformIds.java` et `ResolveTransformIds.java`.

4.2 Vérifications sémantiques concernant la section Abstract Syntax Tree

Dans le *méta-fichier de spécification*, la section `Abstract Syntax Tree` contient les productions dont les noeuds serviront à déterminer la structure de l'arbre syntaxique. Les vérifications sémantiques de cette section sont exactement les mêmes que celles effectuées pour la section `Productions` lorqu'elle est dépourvue de toute syntaxe de transformations d'AST. Les vérifications sémantiques effectuées sur la section `Productions` dépourvue de transformations d'arbres syntaxiques ne seront pas abordées car elles étaient déjà en place.

La section `Abstract Syntax Tree` est composée d'une ou de plusieurs productions. Chacune de ces productions est à son tour constituée d'une ou de plusieurs al-

ternatives (règles) qui sont elles aussi constituées d'un ou plusieurs symboles pouvant être soit des productions ou des jetons éventuellement optionnels ($elem_N$?), soit des listes ($elem_N$ +) probablement vides ($elem_N$ *) de productions ou de jetons. Rappelons qu'une production est définie comme suit :

$$
\begin{aligned}
\texttt{prod} = \{\texttt{nom\_alternative}_1\}\ \texttt{elem}_1\ \texttt{elem}_2\ \ldots\ \texttt{elem}_{\texttt{M}_1} \\
\mid\ \{\texttt{nom\_alternative}_2\}\ \texttt{elem}_1\ \texttt{elem}_2\ \ldots\ \texttt{elem}_{\texttt{M}_2} \\
\mid\ \ldots. \\
\mid\ \{\texttt{nom\_alternative}_\texttt{N}\}\ \texttt{elem}_1\ \texttt{elem}_2\ \ldots\ \texttt{elem}_{\texttt{M}_\texttt{N}} \\
;
\end{aligned}
$$

où $\texttt{M}_1$, $\texttt{M}_2$, ..., $\texttt{M}_\texttt{n}$ correspondent respectivement au nombre d'éléments des alternatives $\texttt{nom\_alternative}_1$, $\texttt{nom\_alternative}_2$, ..., $\texttt{nom\_alternative}_\texttt{N}$ de la production `prod` et $\texttt{elem}_\texttt{X}$ (`X`, un entier) correspond soit à un jeton, soit à un nom de production définie dans la section `Abstract Syntax Tree`.

Les vérifications sémantiques s'appliquent à tous les constituants d'une production et permettent, conformément aux propriétés précédemment citées, d'assurer le respect des règles listées ci-dessous. Quand une de ces règles est violée, SableCC appelle une méthode `error(...)` ou `errorX(...)` où `X` représente un numéro arbitraire attribué à l'erreur dans le code source. Cette méthode lance une exception permettant ainsi d'arrêter l'exécution de SableCC. Un message d'erreur approprié est alors imprimé sur la sortie standard indiquant à l'usager la source du problème à l'intérieur du fichier de spécification.

Les règles à respecter au niveau de la section `Abstract Syntax Tree` sont les suivantes :

Règle 1 : s'assurer qu'aucune production n'est définie plus d'une fois et que les différentes alternatives d'une même production ont toutes des noms différents. Par défaut, le nom de l'alternative est composé d'un "`A`" majuscule concaténé avec le

nom situé à l'intérieur des accolades (`{}`) au début de l'alternative s'il y a lieu mis dans un certain format, concaténé aussi avec le nom de la production également selon un certain format. Le même type de vérification est effectué sur les éléments composant une alternative. Ainsi dans l'exemple 4.1 ci-dessous, vu que le nom de la première production (`prod`) est le même que celui de la deuxième et que dans l'exemple 4.2, le nom de la première alternative (`ANomAltProd`) après avoir été converti conformément à la nomenclature en vigueur dans SableCC est aussi le même que celui de la deuxième alternative, il s'agit d'un cas de violation de cette règle. Il en est de même pour les éléments $\texttt{elem}_1$ dans l'exemple 4.2.

Exemple 4.1

$\texttt{prod} = \texttt{elem}_1\ \texttt{elem}_2\ \texttt{elem}_3\ ;$

$\texttt{prod} = \texttt{autre\_elem}_1\ \texttt{autre\_elem}_2\ ;$

Exemple 4.2

```
prod = {nom_alt} elem1 elem2 elem1 |
       {nom_alt}  ...              ;
```

Quand ce type d'erreur survient, la méthode `void error(Token, String)` ou encore `void error(String)` de la classe `ResolveAstIds` est ainsi appelée levant une exception `RuntimeException` avec un message d'erreur approprié.

Règle 2 : s'assurer que tous les éléments apparaissant dans les alternatives sont définis comme production dans la section `Abstract Syntax Tree` ou encore comme jetons dans la section `Tokens`. Dans le cas contraire, une erreur est signalée en appelant la méthode `void error(Token, String)`.

Pour l'exemple 4.3 si $\texttt{elem}_1$ n'est pas défini ni dans la section `Tokens`, ni dans la section `Abstract Syntax Tree`, une erreur est signalée à l'utilisateur.

Exemple 4.3

$\texttt{prod} = \texttt{elem}_1\ \texttt{elem}_2\ ;$

Règle 3 : s'assurer, si un élément est reconnu comme étant un jeton, qu'il n'est pas ignoré. Autrement dit, il ne figure pas dans la liste des jetons dans la section

`Ignored Tokens`. La méthode `void error(Token, String)` est appelée dans le cas où cette condition n'est pas respectée. En effet, étant donné que les jetons considérés comme jetons ignorés ne sont pas traités par l'analyseur syntaxique, ils ne peuvent pas apparaître dans l'arbre syntaxique et par conséquent dans les productions des sections `Abstract Syntax Tree` ni `Productions`.

Exemple 4.4

```
...
Ignored Tokens
   elem, ... ;
Abstract Syntax Tree
   ...
   prod = T.elem ... ;
   ...
```

Règle 4 : s'assurer que s'il y a ambiguïté à propos du type d'un élément, un spécificateur du genre "`T.`" ou "`P.`" puisse lever l'ambiguïté sans équivoque. Ce genre d'erreur peut survenir si une production définie dans la section `Abstract Syntax Tree` porte le même nom qu'un jeton. Rappelons que les jetons sont définis dans la section `Tokens`.

Exemple 4.5

```
...
Tokens
   ...
   switch = 'switch' ;
   ...
Abstract Syntax Tree
   ...
   switch = ... ;
   ...
   instr = {switch} switch ... ;
```

Règle 5 : s'assurer que le mot clé `class` ne soit pas le nom d'un élément d'une alternative. Cette contrainte est imposée pour éviter la confusion entre la méthode `getClass()` de la classe `Object` du langage `Java` et qui est donc héritée par défaut par toutes les classes en `Java` et la méthode du même nom qui résulterait si un symbole était nommé `class`. En effet, comme vu dans le chapitre 2, le cadre de travail de SableCC permet de générer automatiquement des classes pour représenter les différentes alternatives des productions définies dans la section `Abstract Syntax Tree`. Ainsi, si un élément avait comme nom `class`, la méthode d'accès qui en résulterait serait `getClass()`. Ceci aurait pour conséquence la redéfinition de la méthode `getClass()` de la classe `Object`. Ce qui n'est pas permis par le langage car cette méthode est déclarée avec le mot clé `final`, ce qui signifie en `Java` qu'elle ne peut pas être redéfinie.

Exemple 4.6

```
...
Abstract Syntax Tree
   ...
   prod = class ... ;
   ...
```

Une solution qui permettrait d'éviter ce genre de problème serait simplement d'ajouter :

```
prod = [classe] :class ... ;
```

4.3 Vérifications sémantiques concernant la section Productions

Les vérifications sémantiques dans cette section doivent être effectuées sur les transformations de productions et les transformations d'alternatives.

4.3.1 Vérification sur les productions

La syntaxe de spécification d'une transformation de production est la suivante :

prod { → prod_transform$_1$ prod_transform$_2$ ... prod_transform$_N$ }

où prod_transform$_1$, prod_transform$_2$, ..., prod_transform$_N$ correspondent à des jetons non ignorés ou encore à des symboles (de production) de la section Abstract Syntax Tree.

Les vérifications sémantiques à effectuer sur cette partie concernent d'une part l'existence de chacun de chacun des symboles prod_transform$_X$ (X étant un entier) dans les sections appropriées de la grammaire, et d'autre part, s'assurer de la non ambiguïté des noms donnés à ces symboles.

Pour la première de ces vérifications, il faut déterminer le type du symbole. S'il s'agit d'un jeton, il faut ensuite s'assurer que ce jeton n'est pas ignoré. Cependant, s'il s'agit d'une production, il faut s'assurer que cette production est définie dans la section Abstract Syntax Tree. Si le symbole n'apparaît dans aucune des deux sections Tokens et Abstract Syntax Tree, alors il s'agit d'une erreur qui doit être signalée.

La seconde de ces vérifications consiste à s'assurer que tous les symboles ont des noms différents. Prenons l'exemple de la production avec la transformation suivante :

```
Tokens
    A = ... ;
    b = ... ;
Productions
    prod {→ A b A } = .... ;
    ...
Abstract Syntax Tree
    b = ... ;
```

Il y a violation de l'unicité des noms car le jeton A y apparaît deux fois. Pour éviter ce genre d'erreur, il suffit de renommer le deuxième symbole A avec le mécanisme de nommage des symboles offert par SableCC. La production deviendrait donc :

```
prod {→ A b [autre_a] :A} = .... ; .
```

La deuxième erreur présente dans cette transformation concerne le type de l'élément `b`. En effet, cet élément est à la fois défini comme jeton mais aussi comme symbole de production. Il y a donc ambiguïté sur le type de `b`. Pour lever cette ambiguïté, il suffit de préciser le type de l'élément `b` en la faisant précéder d'un spécificateur (`T.` ou `P.`). La transformation de production se réécrirait de la manière suivante si le b est un jeton :

```
prod {→ A T. b [autre_a] :A} = .... ;
```

et s'il s'agit d'un symbole de production,

```
prod {→ A P. b [autre_a] :A} = .... ; .
```

4.3.2 Vérifications sur les alternatives

Il y a cinq types de transformations d'alternative. Les vérifications sémantiques s'appliquent donc à ces différentes transformations.

4.3.2.1 La récupération à l'intérieur de la forêt d'arbres d'un noeud ou d'une feuille déjà existante (*simple*)

Rappelons que la syntaxe de ce type de transformation est :

```
symb_grammaire[.prod_transform]
```

Un noeud déjà existant peut être soit un jeton, soit une production. Dans le cas d'une production, SableCC offre la possibilité de ne retenir que certains des éléments de cette production. Plus exactement, il s'agit de certains des éléments de la transformation de cette production. C'est ce à quoi sert la partie optionnelle de la syntaxe de spécification des transformations d'alternative de type `simple ([.prod_transform] )`.
Les vérifications sémantiques à effectuer sur ce type de transformation sont :

1. la vérification de l'existence du noeud dans l'alternative courante. Dans la production, cela se traduit par la vérification que dans l'alternative courante (c'est-à-dire à gauche de la transformation de l'alternative), il existe bel et bien un symbole correspondant à celui désigné par le noeud.

   Exemple 4.7

```
simple_term_tail   {→ id}
        = dot id   {→ id}.
```

Ici, l'élément `id` qui apparaît dans la transformation de l'alternative (`{→ id}`) existe bel et bien à gauche de cette transformation, c'est-à-dire que `id` est un symbole de l'alternative courante (`= dot id`). Cette condition est donc respectée ;

2. dans le cas où la partie optionnelle `prod_transform` existe, il faut s'assurer dans un premier temps que `symb_grammaire` ne désigne pas un jeton (car les jetons ne sont pas transformés) et dans un second temps, vérifier que dans la section `Productions`, il existe une production dont le type correspond à celui de `symb_production` et que cette production contient dans sa transformation de production un symbole dont le nom est le même que `prod_transform`.

   Exemple 4.8

   ...

```
Productions
  ...
  prod_name
       = id prod_name_tail ?
                    {→ New prod_name(id, prod_name_tail.id)} ;
  prod_name_tail  {→ id}
       = dot id
                    {→ id} ;
```

   Dans l'exemple 4.8, tout d'abord, il faut vérifier que `prod_name_tail` est une production, ce qui est le cas. Ensuite, on fait la vérification suivante : `id` est-il un des éléments de la transformation de la production `prod_name_tail` ? Ceci est également vrai. La vérification restante concerne la concordance de types, c'est-à-dire s'assurer que le type de `prod_name_tail.id` correspond au type du deuxième élément de l'alternative sans nom de la production *prod_name* de la section `Abstract Syntax Tree`, c'est-à-dire `id` ;

3. dans le même ordre d'idée, si la spécification de la transformation se limite uniquement à un identificateur, il faut s'assurer que l'identificateur désigne un jeton.

Sinon, si l'identificateur désigne une production, il faut s'assurer que la production ne comporte aucune transformation.

Exemple 4.9

```
prod_name_tail {→ id}
          = dot id
                {→ id.id} ;
```

La transformation d'alternative de l'exemple 4.9 est incorrecte car `id` est un jeton et on sait que les jetons ne peuvent être transformés.

Exemple 4.10

```
reg_exp                          {→ concat∗}
   = concat [concats] :reg_exp_tail*
                                 {→ [concat, concats.other]} ;
reg_exp_tail                     {→ concat}
   = bar concat                  {→ concat} ;
```

Dans l'exemple 4.10, dans la transformation `concats.other`, `concats` désigne une réalisation de la liste `reg_exp_tail*`, or la transformation de la production `reg_exp_tail` ne contient aucun élément dont le nom est `other`. Par conséquent, la transformation `concats.other` est invalide.

4.3.2.2 La création d'un nouveau noeud (*New*)

Rappel : La création d'un nouveau noeud fait intervenir le nom du noeud à créer. Ce nom est composé d'un identificateur et éventuellement d'un point (`.`) suivi d'un autre identificateur. Le premier identificateur fait référence au symbole d'une production, alors que le deuxième identificateur désigne l'alternative correspondante de cette production par son nom s'il y a lieu. Il est important de noter que cette production doit faire partie de la section `Abstract Syntax Tree`.

Exemple 4.11

```
Productions
```

```
  ...
  helper_def                    {→ helper_def}
    = id equal reg_exp semicolon
                                {→ New helper_def(id, reg_exp) } ;
  ...
Abstract Syntax Tree
  ...
  helper_def = id reg_exp ;
  ...
```

Les vérifications requises pour cette transformation d'alternative sont les suivantes :

1. S'assurer qu'il existe bien une production de la section `Abstract Syntax Tree` dont le nom est le même que le premier identificateur. De plus, dans le cas de la présence du deuxième identificateur, il faudrait aussi s'assurer que cette production contient une alternative dont le nom correspond à cet identificateur. S'il n'y a pas de deuxième identificateur, il faut quand même s'assurer que cette production contient une alternative sans nom.
2. Vérifier le nombre de paramètres (`New prod.alt(`$\texttt{param}_1$, $\texttt{param}_2$, `...,` $\texttt{param}_N$`)`) en le comparant avec le nombre d'éléments de l'alternative de la production correspondante (`prod = {alt_name}` $\texttt{elem}_1$ $\texttt{elem}_2$ `...` $\texttt{elem}_M$`)`. Si `N=M` alors la règle concernant le nombre de paramètres est respectée sinon celle-ci est violée.
3. Vérifier la concordance des types entre les paramètres du `New` et les éléments de l'alternative dans la section `Abstract Syntax Tree` ($\texttt{param}_1$ avec $\texttt{elem}_1$, $\texttt{param}_2$ avec $\texttt{elem}_2$ et ainsi de suite jusqu'à $\texttt{param}_N$ avec $\texttt{elem}_M$`)`. La méthode utilisée par SableCC sera discutée plus précisément dans la partie de ce chapitre réservée à la concordance de types (section 4.4).

Dans l'exemple 4.11 d'où est tirée la transformation d'alternative (`New`) `New helper_def(`

`id, reg_exp)`, il n'y a pas de deuxième identificateur, donc il s'agit du premier cas de figure. Il est aussi facile de constater qu'il existe bel et bien une production `helper_def` dans la section `Abstract Syntax Tree`, et que cette production a une alternative sans nom (`helper_def = id reg_exp ;`). Cette alternative est effectivement constituée de deux éléments, ce qui permet de respecter la condition sur le nombre de paramètres. Les paramètres se trouvent à l'intérieur des parenthèses, juste après le mot-clé `New` et le nom de la production `helper_def`. Pour la concordance de type, cela se fait en se basant sur l'ordre d'apparition des paramètres. La règle générale consiste donc à s'assurer que le type des paramètres correspond dans l'ordre aux types des éléments dans l'alternative choisie. Dans l'exemple 4.11, il s'agit de comparer les paramètres `id` et `reg_exp` aux éléments `id` et `reg_exp` de l'alternative. Dans ce cas-ci, la concordance de type est respectée. En effet, le type de `id`, élément d'alternative mais aussi paramètre de création de nouveau noeud est `TId` car `id` est bien défini dans la section `Tokens` et pas dans la section `Productions`. Celui de `reg_exp`, dans les deux cas, est aussi `PRegExp` encore une fois parce que `reg_exp` n'est défini que dans la section `Productions`. Il faut reconnaître que déterminer le type des différents paramètres de la création du nouveau noeud et des éléments de l'alternative était simple pour cet exemple. Cela peut être un peu plus fastidieux dans certains cas.

4.3.2.3 La création d'une liste d'éléments ([...])

Les listes créées par SableCC sont des listes homogènes, c'est-à-dire que les éléments contenus dans la liste doivent être du même type. Donc, une des vérifications importantes qui est faite à ce niveau consiste à s'assurer que tous les éléments à l'intérieur de la liste (en fait entre les crochets [ ]) sont du même type. L'autre vérification importante consiste à s'assurer que le type des éléments à l'intérieur de la liste correspond effectivement au type spécifié par la transformation de production correspondante.

4.3.2.4 La transformation vide

Cette transformation intervient quand on veut se débarasser complètement d'une production. Elle permet d'éliminer une production et ainsi de ne plus pouvoir y faire référence en utilisant son symbole dans le reste des transformations.

Exemple 4.12

```
token_def
          = state_list ? id equal reg_exp look_ahead ? semicolon
                            {→ New token_def(state_list, id, reg_exp) } ;
look_ahead                  {→ }
          = slash reg_exp   {→ } ;
```

L'exemple 4.12 ci-dessus est tiré de la définition des jetons dans le logiciel SableCC. Cette définition de jeton permet à l'usager de définir des jetons en spécifiant un "`lookahead`". L'incidence du lookahead permet de ne reconnnaître un jeton que si ce dernier est suivi d'un mot qui correspond à l'expression régulière représentée par le `lookahead`. Mais dans le logiciel SableCC, le `lookahead` n'est pas mis en oeuvre, ce qui fait qu'en réalité, cette fonctionnalité n'est pas offerte. Cependant, sa spécification est tout de même permise par la syntaxe. On peut donc décider de tout simplement éliminer la production qui représente le `lookahead` dans l'arbre syntaxique, car elle n'apporte aucune information supplémentaire. C'est ce qui est fait avec la transformation (`vide`) de la production `look_ahead` de l'exemple 4.12. Ceci implique que si le symbole `look_ahead` apparaît comme élément dans une autre alternative (comme dans l'alternative sans nom de la production `token_def`), elle ne peut pas être référencée dans la transformation d'alternative associée car elle est considérée comme n'ayant jamais existée.

4.3.2.5 L'élimination d'un noeud (*Null*)

Rappel : Cette transformation peut être considérée comme un *Joker*. En effet, elle peut être utilisée partout où peut l'être une transformation du type `simple` ou

```
New.
```

La vérification sémantique s'appliquant dans le cas de cette transformation concerne principalement la concordance des types. Il faut, entre autre, s'assurer que `Null` n'est pas utilisé pour une liste car la transformation pour indiquer une liste vide est `[ ]` et non `[ Null ]` . La raison de cette interdiction tient au fait que les listes utilisées par SableCC ne peuvent pas contenir la référence `null`.

Exemple 4.13

```
prod_name_tail                                {→ id? }
          = {brouillon} dot par_g id par_d  {→Null}
          | {reel}      dot id              {→ id }
          ;
```

Dans l'exemple 4.13, on veut éliminer l'alternative `{brouillon}` de la production. Étant donné que la production dans son ensemble se transforme, il faut absolument spécifier une transformation pour toutes les alternatives y compris l'alternative `{brouillon}`. Le `Null` nous permet donc dans ce cas d'éliminer l'alternative en ne retenant rien.

Exemple 4.14

```
list_term
        = {new} new prod_name  l_par params? r_par
           {→ New list_term.new(prod_name, l_par, [params.list_term])}
        | {simple} specifier? id simple_term_tail?
           {→ Null } ;
```

Dans cet autre exemple 4.14, on remplace une alternative pour laquelle on aurait normalement créé un nouveau noeud (`New list_term.simple(...)`) par une transformation `Null`.

4.4 Vérification de la concordance de types

Il s'agit d'une des plus importantes vérifications sémantiques effectuées pour les transformations d'arbres syntaxiques. Effectuée en dernier, c'est cette ultime vérification qui garantit la validité des transformations spécifiées par l'utilisateur et s'assure ainsi de la robustesse et de l'intégrité de l'arbre syntaxique généré. Cette vérification intervient pour tous les types de transformations. Mieux encore, elle permet également dans le cas de combinaisons de plusieurs types de transformations de s'assurer que la transformation résultante est valide. Cette vérification intervient entre les transformations d'alternatives et de productions mais aussi à l'intérieur d'une même transformation d'alternative.

```
prod                    {–> prod_transform1 prod_transform2 ... prod_transformN }

      = {alt1}  elem1 elem2 ... elemN
                        {–> alt1_transform1 alt1_transform2 ... alt1_transformN }

      | {alt2} elem1 elem2 ... elemN
                        {–> alt2_transform1 alt2_transform2 ... alt2_transformN }
```

Figure 4.1 Prototype de définition d'une production avec transformation

Considérant la production de la figure 4.1, la concordance de types entre la transformation d'une production et ses transformations d'alternatives consiste à s'assurer que $\texttt{prod\_transform}_1$ est du même type que $\texttt{alt1\_transform}_1$ et $\texttt{alt2\_transform}_1$, et que $\texttt{prod\_transform}_2$ est du même type que $\texttt{alt1\_transform}_2$ et $\texttt{alt2\_transform}_2$, et ainsi de suite jusqu'à la concordance entre $\texttt{prod\_transform}_N$, et $\texttt{alt1\_transform}_N$ et $\texttt{alt2\_transform}_N$. Quand $\texttt{prod\_transform}_X$ représente un jeton, alors la concordance de types impose que les `alt_transform` correspondants représentent aussi le même jeton ou le joker `Null`. Mais quand il s'agit d'une production de la section `Abstract Syntax Tree`, il suffit que les transformations d'alternatives associées correspondent à une des alternatives de cette production. La concordance de types à travers les différents types de transformations est détaillée ci-après.

4.4.1 Récupération d'éléments à l'intérieur de la forêt d'arbres d'un noeud ou d'une feuille existante (*simple*)

La syntaxe d'une telle transformation est fidèlement décrite par les exemples 4.8 et 4.9. Dans l'exemple 4.9, il s'agit de s'assurer que le `id` de la transformation d'alternative `{→ id}` correspond au `id` de la transformation de production. Pour l'exemple 4.8 reproduit dans la figure ci-bas, la détermination du type de la transformation `tail.id`, suppose de regarder la transformation de production de l'élément dont le nom est `tail`. Ici, il se trouve qu'il s'agit de la production `prod_name_tail`. Sa transformation de production contient effectivement un élément dont le nom est `id`. `id` est un jeton donc le type de `tail.id` est donc `TId`. Pour la concordance de types, en ce qui concerne les paramètres du `New`, se référer au point 4.4.2.

```
Productions

    ...

    prod_name
            = id  [tail]:  prod_name_tail?          {–> New prod_name(id, tail.id)} ;

    prod_name_tail                                  {–> id};
            = dot  id                               {–> id};

Abstract Syntax Tree

    prod_name = [name]:  id  [alt_name]:  id?;
```

Figure 4.2 Extrait d'une grammaire de SableCC mettant en oeuvre une transformation de type `simple`

4.4.2 La création d'un nouveau noeud (*New*)

La syntaxe du `New` est la suivante : `New ast_prod.alt(`$\texttt{param}_1$, $\texttt{param}_2$`, ...,` $\texttt{param}_N$`)`. Tel que mentionné précédemment, il faut que `ast_prod` soit une production

de la section `Abstract Syntax Tree` et que cette production comporte une alternative dont le nom est `alt`. Cette alternative doit avoir le même nombre d'éléments que le nombre de paramètres de la transformation, c'est-à-dire que obligatoirement : `ast_prod = {alt}` $\texttt{elem}_1$ $\texttt{elem}_2$... $\texttt{elem}_N$. Ensuite, pour la concordance de types, il faut que $\texttt{elem}_1$ soit du même type que $\texttt{param}_1$, $\texttt{elem}_2$ du même type de $\texttt{param}_2$, et ainsi de suite jusqu'à $\texttt{elem}_N$.

La transformation d'alternative `New prod_name(id, tail.id)` dans l'exemple 4.10 respecte les contraintes citées ci dessus car `prod_name` est bien une production définie dans la section `Abstract Syntax Tree`. De plus, cette production a une seule alternative sans nom. Cette alternative est constituée de deux éléments de type `id`, ce qui correspond aussi aux paramètres de la transformation d'alternative.

4.4.3 La création d'une liste d'éléments (*list*)

La syntaxe de création de liste : [$\texttt{list\_elem}_1$, $\texttt{list\_elem}_2$, ..., $\texttt{list\_elem}_N$]. Les validations sémantiques à effectuer consistent à :

1. s'assurer que tous les éléments $\texttt{list\_elem}_X$ (`X` un entier) sont du même type.
2. s'assurer qu'aucun des éléments n'est le joker `Null` car `Null` ne peut apparaître dans la liste.
3. s'assurer de la correspondance stricte avec l'opérateur `+` ou `*`. En d'autres termes, en considérant la production de l'exemple 4.15, la transformation d'alternative ([$\texttt{elem}_1$ $\texttt{elem}_2$]) est correcte étant donné que la transformation de production (`prod_transform+`) à laquelle elle est associée a un opérateur `+`.

   Exemple 4.15

   `prod` {→ `prod_transform+` }

   = $\texttt{elem}_1$ $\texttt{elem}_2$ {→ [$\texttt{elem}_1$ $\texttt{elem}_2$] }

4.4.4 La transformation vide

Si une transformation de production indique une transformation vide, il faut simplement vérifier que toutes les alternatives de cette production indiquent aussi une transformation vide.

4.4.5 L'élimination d'un noeud (*Null*)

Rien de particulier à ce niveau si ce n'est qu'il faut s'assurer que l' élément dans la transformation de production ou dans l'alternative de la section `Abstract Syntax Tree` auquel on associe le *Null* est suivi par un opérateur `?`.

4.5 Vérification stricte imposée par la multiplicité des opérateurs (`?`, `*` et `+`)

En vue de garantir la construction d'un arbre syntaxique conforme aux multiplicités dues à la présence éventuelle d'opérateurs `?`, `*` ou `+` dans la grammaire de la section `Abstract Syntax Tree`, nous avons mis en place des vérifications sémantiques strictes par rapport à ces opérateurs. L'objectif est de faire en sorte que les noeuds de l'arbre syntaxique réflètent vraiment la spécification de la grammaire par rapport à ces opérateurs. Lorsque les opérateurs `*` et `+` apparaissent devant un symbole au niveau de la grammaire de l'arbre syntaxique, cela implique la présence dans l'arbre syntaxique d'une liste contenant des noeuds correspondant à ce symbole. Quand il s'agit de l'opérateur `*`, cette liste peut être vide alors que dans le cas de l'opérateur `+`, elle doit absolument contenir des noeuds. Un traitement similaire est effectué en présence de l'opérateur `?`. En effet, la présence de cet opérateur devant un symbole dans la grammaire de l'arbre syntaxique signifie que le noeud correspondant dans l'arbre syntaxique peut être une référence vers l'objet `null`. Dans le cas contraire (si aucun opérateur n'apparait devant le symbole), un noeud `non null` doit être construit pour ce symbole et inséré dans l'arbre syntaxique. Ces vérifications interviennent au niveau de la spécification des transformations dans la section `Productions` entre les transformations de produc-

tions et les transformations d'alternatives associées, tout ceci en étroite collaboration avec les productions de la section `Abstract Syntax Tree`. La vérification stricte se fait plus exactement entre l'opérateur `?` et l'absence d'opérateurs et entre l'opérateur `*` et l'opérateur `+`.

4.5.1 L'opérateur ? et l'absence d'opérateur

En cas d'absence d'opérateur, il faut s'assurer que le noeud de l'arbre syntaxique qui sera construit pour la transformation d'alternative spécifiée ne sera pas `null`. Prenons des exemples pour clarifier les choses.

Exemple 4.16

```
Productions
  ...
  prod_name_tail   {→ id}    /* transformation de production */
     = dot id      {→ id} ;  /* transformation d'alternative */
  ...
```

Dans ce premier exemple, la transformation d'alternative `id` correspond à la transformation de production `id`. Cette transformation de production n'est suivie d'aucun opérateur, donc la transformation d'alternative `id` associée est correcte. Par contre, si la transformation d'alternative avait été `Null`, ceci serait incorrect car la transformation de production `id` n'est pas suivie d'un opérateur `?`.

Exemple 4.17

```
Productions
  ...
  prod_name                                {→ id [tail] :id }
            = id [tail] :prod_name_tail ?  {→ id tail.id)} ;
  prod_name_tail                           {→ id}
```

```
            = dot id                        {→ id} ;
```

Dans ce deuxième exemple, la transformation d'alternative `tail.id` correspondant à la transformation de production `[tail] :id` n'est pas correcte car `tail.id` peut occasionner la création d'un noeud null dans l'arbre syntaxique. En effet, *tail* est le nom donné au symbole `prod_name_tail` dans l'alternative de la production *prod_name*. Or ce symbole est optionnel, ce qui veut dire que la production dont elle fait partie sera transformée à l'interne par SableCC et deviendra :

```
prod_name                               {→ id [tail] :id }
          = id [tail] :prod_name_tail   {→ id tail.id)}
          | id                          {→ id Null)} ;
prod_name_tail                          {→ id}
          = dot id                      {→ id} ;
```

Comme précédemment mentionné, `Null` ne peut être accepté comme transformation d'alternative que si le symbole de la transformation de production associé est suivi d'un opérateur `?`, ce qui n'est plus vrai dans ce cas. Pour que cette transformation soit valide, il faut que la transformation de la production `prod_name` soit : `{→ id [tail] :id ? }`

Exemple 4.18

```
Productions
  ...
  prod_name
            = id [tail] :prod_name_tail ?  {→ New prod_name(id, tail.id)} ;
  prod_name_tail                           {→ id}
            = dot id                       {→ id} ;
Abstract Syntax Tree
  prod_name = [name] :id [alt_name] :id ;
```

Ce dernier exemple sur l'absence d'opérateur fait intervenir la création d'un noeud de l'arbre syntaxique. Nous avons illustré dans l'exemple 4.18 que la transformation d'alternative `tail.id` peut être transformée en `Null` par SableCC. Dans une telle situation, la transformation d'alternative `New prod_name(id, tail.id)` peut devenir `New prod_name(id, Null)`, ce qui la rendrait invalide car le deuxième élément `id` de l'alternative de la production `prod_name` dans la section `Abstract Syntax Tree` n'est pas suivi par un opérateur `?`.

4.5.2 L'opérateur + et l'opérateur *

Il faut s'assurer en cas de présence de l'opérateur `+` que la liste associée ne peut pas être vide. Cette vérification implique que la liste ne soit pas `[ ]` mais il faut aussi que dans le cas où il y a des éléments dans cette liste que tous ces éléments ne soient pas supprimés par une transformation interne éventuelle de SableCC.

Exemple 4.19

```
pkg_name                                    {→ pkg_id+ } =
    pkg_id [pkg_ids] :pkg_name_tail* semicolon
                                            {→ [ ] } ;
pkg_name_tail                               {→ pkg_id } =
    dot pkg_id                              { → pkg_id } ;
```

La transformation `[ ]` de l'exemple 4.19 est invalide car la transformation de l'alternative de la production `pkg_name` est une liste vide alors que la transformation de production associée est suivie de l'opérateur `+`. Cet exemple aurait été valide si l'opérateur avait plutôt été `*`.

Exemple 4.20

```
pkg_name                                {→ pkg_id+ } =
    pkg_id [pkg_ids] :pkg_name_tail* semicolon
```

```
                                        { → [pkg_id, pkg_ids.pkg_id] } ;
pkg_name_tail                           {→ pkg_id } =
    dot pkg_id                          {→ pkg_id } ;
```

Par contre, l'exemple 4.20 est valide malgré le fait que dans la transformation d'alternative `[pkg_id, pkg_ids.pkg_id]`, `pkg_ids.pkg_id` pourrait être supprimée lors de transformations internes de SableCC. En effet, `pkg_ids` est le nom donné au symbole `pkg_name_tail` avec l'opérateur `*`. Or le caractère optionnel que confère cet opérateur au symbole `pkg_name_tail` amènera SableCC à supprimer `pkg_ids.pkg_id` dans la transformation d'alternative `[pkg_id, pkg_ids.pkg_id]` lors des transformations internes sur la grammaire. La transformation deviendra `[pkg_id]`, ce qui reste valide malgré tout. Le résultat des transformations internes effectuées par SableCC sur l'exemple 4.20 donne :

```
pkg_name                {→ pkg_id+ }
        = pkg_id semicolon
                        {→ [pkg_id] }
        | pkg_id [pkg_ids] :$pkg_name_tail semicolon
                        {→ [pkg_id, pkg_ids.pkg_id] } ;
pkg_name_tail           {→ pkg_id }
        = dot pkg_id    {→ pkg_id } ;
$pkg_name_tail          {→ pkg_id* }
        ={terminal} pkg_name_tail
                        {→ [pkg_name_tail.pkg_id] }
        |{nonterminal} $pkg_name_tail pkg_name_tail
                        {→ [pkg_name_tail.pkg_id
                        $pkg_name_tail.pkg_id] } ;
```

4.6 Résumé

Les vérifications sémantiques que le logiciel SableCC effectue sur les grammaires d'entrée ont été présentées. Dans un premier temps, les motivations de telles vérifications sémantiques ont été données et ensuite, les différentes sections de la grammaire concernées par ces vérifications ont été décrites. Il a également été montré en quoi ces vérifications sémantiques étaient importantes pour l'intégrité de l'arbre syntaxique construit.

Chapitre V

SUPPORT DE LA SYNTAXE EBNF

Les analyseurs syntaxiques générés par SableCC à partir d'un fichier de spécification font l'analyse syntaxique des programmes compatibles à la grammaire de la section `Productions` de ce fichier de spécification. Nous avons vu dans le chapitre 2 que l'analyse syntaxique se faisait grâce à une dérivation de l'axiome de la grammaire contenu dans la section `Productions`. Il y a de ce fait une correspondance entre cette grammaire et l'analyseur syntaxique.

L'algorithme sous-jacent à l'analyseur syntaxique ne prend que des grammaires dont la syntaxe de spécification est basée sur une notation BNF. Or la spécification de la section `Productions` des fichiers de spécification de SableCC qui sert à la génération des analyseurs syntaxiques accepte la syntaxe EBNF "type-SableCC". Dès lors que l'algorithme interne de l'analyseur syntaxique est basé sur une notation BNF mais que la section `Productions` peut être spécifiée en utilisant une notation "EBNF" *type-SableCC*, il a fallu transformer les constructions EBNF de la section `Productions` en constructions BNF. Donc, pour une grammaire contenant des constructions EBNF, SableCC crée à l'interne une grammaire BNF équivalente et c'est à partir de cette grammaire que l'analyseur syntaxique est généré. Dans la première partie de ce chapitre, nous allons faire un bref rappel sur la syntaxe EBNF utilisée pour les grammaires de SableCC et nous parlerons aussi des différentes transformations internes effectuées pour chacun des opérateurs de la notation EBNF supportée par SableCC. Ensuite dans une deuxième partie, nous montrerons l'incidence de ces transformations internes sur les transformations d'alternatives

et enfin nous concluerons.

5.1 La syntaxe EBNF (Extended Backus Naur Form)

Comme le nom l'indique, EBNF est une extension de la notation BNF. Elle permet l'écriture de grammaires non-contextuelles (par exemple, les grammaires de langage de programmation). Elle permet d'utiliser les opérateurs d'expressions régulières pour simplifier certaines notations. Les opérateurs supportés sont : `*`,`+`, `?`,`()`. Ces opérateurs permettent de facilement spécifier un certain nombre de constructions. SableCC ne supporte que les opérateurs `*`, `+` et `?`. En effet, plutôt que de faciliter la tâche de l'utilisateur, l'utilisation des parenthèses peut parfois s'avérer très délicat et se révéler une source de confusion d'un point de vue lisibilité. Prenons pour exemple la production suivante :

```
p = (a a | b)* | (c d)+.
```

On peut certes déterminer le nombre d'alternatives de cette production mais pas de manière triviale. Pour preuve, la décomposition ci-dessous permet de les trouver en trois étapes, ce qui est long et non trivial.

```
  p =       (a a | b )*          |
            (c d)+               ;
============>
  p =       {first} new1_p*      |
            {second} new2_p+     ;
  new1_p =  {first} a a          |
            {second} b           ;
  new2_p =  c d                  ;
```

Nous allons maintenant présenter les différents types d'opérateurs ainsi que les transformations internes impliquées par ces opérateurs au niveau de SableCC.

5.1.1 L'opérateur ?

L'opérateur `?` permet d'exprimer la présence optionnelle du symbole qui le précède. Ainsi la production suivante :

```
prod = a b ?
```

peut être réexprimée de la manière suivante :

```
prod = {one} a   |
       {two} a b ;
```

Pour chacun des opérateurs `?` apparaissant dans la section `Productions` du fichier de spécification, SableCC fait donc une transformation semblable.

5.1.2 L'opérateur +

L'opérateur `+` exprime une possible dérivation multiple du symbole le précédant. En d'autres termes,

```
prod = a b+ ;
```

peut être réexprimée en :

```
prod = a $b ;
$b = {non_terminal} $b b |
     {terminal}     b    ;
```

Il faut noter ici l'ajout de la production `$b`. Cette production récursive gauche[1] permet de réaliser la dérivation multiple du symbole `b`. Donc à chaque fois que l'opérateur `+` apparaît après un symbole de la grammaire, une production dont le nom est formé par la concaténation de `$` et de son symbole est rajoutée à la liste des productions de la grammaire, si elle n'existe pas déjà.

---

[1] $b Dire qu'une production est récursive gauche signifie qu'elle réapparait dans une de ces alternatives et ce, tout de suite après le signe d'égalité (`$b = $b ...`).

5.1.3 L'opérateur *

L'opérateur * est en quelque sorte une combinaison des deux opérateurs précédents. En effet,

```
prod = a b* ;
```

est équivalent à

```
prod = a (b+) ?
```

Plus précisement

```
prod = {one} a b+ |
       {two} a    ;
```

se transforme à son tour en

```
prod = {one} a $b |
       {two} a    ;
```

avec ajout de la production `$b`, comme dans le cas de l'opérateur `+`.

5.2 Modifications subies par les transformations d'alternatives

Les transformations d'alternatives spécifiées par l'usager ne tiennent évidemment pas compte des transformations internes effectuées par le logiciel SableCC vu que ces dernières sont faites après. Ces transformations internes peuvent dans certains cas invalider les transformations d'alternatives initialement spécifiées. Pour éviter ce genre de situation, SableCC ajuste les transformations d'alternatives par rapport aux transformations internes qu'il effectue pour les opérateurs. Nous allons montrer, à travers les différents types d'opérateurs supportés, les ajustements apportés aux transformations d'alternatives correspondantes.

5.2.1 L'opérateur ?

Soit le fichier de spécification suivant de SableCC :

```
Tokens
   a = 'a' ;
   b = 'b' ;
   c = 'c' ;
Productions
   prod              {→ ast_prod}
        = a b? c     {→ New ast_prod(b, c)} ;
Abstract Syntax Tree
   ast_prod = b? c ;
```

La production `prod` ci-dessus sera transformée par SableCC et donnera finalement :

```
prod                  {→ ast_prod } =
      {aprod1} a c    {→ New ast_prod(Null, c) } |
      {aprod2} a b c  {→ New ast_prod(b, c) }    ;
```

Le résultat de cette transformation peut être décomposé en deux étapes. La première correspond à la duplication de l'alternative dans laquelle apparaît l'opérateur `?`. Ensuite, pour être compatible avec SableCC, on donne un nom différent aux deux alternatives (`aprod1` et `aprod2`). On applique alors la transformation de l'opérateur `?`, ce qui a pour effet d'éliminer l'élément `b?` dans la première alternative et d'éliminer le `?` de l'élement `b?` dans la deuxième. Le résultat est donc :

```
prod                  {→ ast_prod} =
      {aprod1} a c    {→ New ast_prod(b, c)}  |
      {aprod2} a b c  {→ New ast_prod(b, c)}  ;
```

Mais la production ci-dessus est incorrecte car la transformation d'alternative de l'alternative `{aprod1} a c {→ New ast_prod(b, c) }` comporte encore un `b` alors que cet élément a été supprimé de l'alternative (`... a c`). La deuxième étape de la transformation va donc consister à mettre à jour les transformations d'alternatives. Pour ce faire, on parcourt les transformations de cette alternative et on remplace toutes

les occurences de l'élément supprimé par `Null`. C'est ainsi que l'alternative `{aprod1} a c {→ New ast_prod(b, c) }` devient `{aprod1} a c {→ New ast_prod(Null, c) }`. Dans le cas où l'opérateur apparaît plusieurs fois dans l'alternative, on répète les étapes précédentes autant de fois que nécessaire. Prenons un exemple pour illustrer ce cas.

Production originale :

```
prod                    {→ ast_prod} =
      a b? c?           {→ New ast_prod(b, c)} ;
```

Traitement de `b?` :

1. duplication de l'alternative et mise en oeuvre du `b?`

```
prod                      {→ ast_prod} =
      {aprod1} a   c? {→ New ast_prod(b, c)}  |
      {aprod2} a  b c? {→ New ast_prod(b, c)}  ;
```

2. remplacement du terme `b` par `Null` dans les transformations des alternatives dans lesquelles `b` a été éliminé

```
prod                      {→ ast_prod} =
      {aprod1} a   c? {→ New ast_prod( Null , c)}  |
      {aprod2} a b c? {→ New ast_prod(b, c)}       ;
```

Traitement de `c?` :

1. duplication de l'alternative et mise en oeuvre du `c?`

```
prod                       {→ ast_prod}=
      {aprod11} a          { → New ast_prod(Null, c)}  |
      {aprod12} a c        { → New ast_prod(Null, c)}  |
      {aprod21} a b        { → New ast_prod(b, c)}     |
      {aprod22} a b c      { → New ast_prod(b, c)}     ;
```

2. remplacement du terme `c` par `Null` dans les transformations des alternatives dans lesquelles `c` a été éliminé

```
prod                    {→ ast_prod}=
      {aprod11} a       {  → New ast_prod(Null,   Null )}  |
      {aprod12} a   c   {  → New ast_prod(Null, c)}        |
      {aprod21} a b     {  → New ast_prod(b,  Null )}      |
      {aprod22} a b c   {  → New ast_prod(b, c)}           ;
```

5.2.2 L'opérateur +

Un traitement plus élaboré est requis pour cet opérateur. En effet, étant donné qu'une nouvelle production est ajoutée pour réaliser l'occurrence multiple, les actions à effectuer sont plus nombreuses. Montrons cela par un exemple. Considérons l'extrait de grammaire suivant :

```
...
Productions
   prod            {→ ast_prod}
        = a b+     {→ New ast_prod([b])} ;
Abstract Syntax Tree
   ast_prod = b+ ;
```

Deux cas de figure se présentent selon le type du symbole `b` :

- soit b est une production sans transformation de production ou un jeton ;
- soit b est une production avec transformation.

5.2.2.1 Production sans transformation de production ou jeton

Dans le premier cas de figure, la transformation de cette production donnerait :

```
prod                          {→ ast_prod }
      = a $b                  {→ New ast_prod([$b.b])} ;

$b                            {→ b+}
      = {nonterminal} $b b {→ [$b.b b] }
      |  {terminal} b         {→ [b] } ;
```

Jusque là rien de nouveau en ce qui concerne le remplacement dans l'alternative de `b+` par `$b` et l'ajout de la nouvelle production `$b`. Ce qui a changé, ce sont les transformations d'alternative (ajout et modification). En effet, la transformation d'alternative `{→ New ast_prod([b])}` a été changée et a été remplacée par `{→ New ast_prod([$b.b])}`. Le terme `b` est devenu `$b.b`. `$b` est la nouvelle production rajoutée pour exprimer l'occurence multiple de `b`. La transformation de production de la production `$b` est `{→ b+ }`, ce qui fait de `$b.b` un terme qui désigne une liste de symboles `b`.
En ce qui concerne les alternatives de la production `$b`, la première est récursive gauche et permet la dérivation multiple du `b` et la seconde peut être en quelque sorte considérée comme condition d'arrêt de la dérivation multiple. Les transformations d'alternatives, quant à elles, permettent de vraiment de construire la liste souhaitée qui sera éventuellement incluse dans l'arbre syntaxique.

- La première (`{nonterminal}`) `{→ [$b.b b] }` construit une nouvelle liste en regroupant la liste précédemment construite (`$b.b`) avec un nouveau symbole `b`.
- La seconde `{→ [b] }` construit une liste avec un seul symbole `b`. L'analyse d'une séquence de `b` commence toujours par la deuxième alternative (`({terminal}`).

En réalité quand une séquence de symboles b est rencontrée dans le programme d'entrée, cette séquence est analysée en se basant sur la production `$b`. Pour avoir une idée de ce qui se passe exactement, examinons d'un peu plus près le comportement de la pile d'analyse syntaxique pour une séquence de trois symboles `b` en entrée.

Le fonctionnement de l'analyseur syntaxique sera vu plus tard. Pour l'instant, nous allons nous concentrer sur les actions effectuées par l'analyseur syntaxique en

(pile d'analyse, entrée)	Action	Liste créée
1 (... 0, bbb EOF)	décaler b	
2 (... 0 b 1, bb EOF)	réduction par $b = b	[b]
3 (... 0 $b 2, bb EOF)	décaler b	
4 (... 0 $b 2 b 3, b EOF)	réduire par $b = $b b	[b b]
5 (... 0 $b 2, b EOF)	décaler b	
6 (... 0 $b 2 b 5, EOF)	réduire par $b = $b b	
7 (... 0 $b 2, EOF)	accepter et arrêter l'analyse	[b b b]

Tableau 5.1 Pile d'analyse de la séquence `bbb` par l'analyseur syntaxique

fonction du symbole en entrée. La séquence à analyser est `bbb`. Cette séquence est découpée par l'analyseur lexical et envoyée sous forme de jetons individuels à l'analyseur syntaxique. À la vue du premier symbole `b`, l'analyseur effectue une action *"décaler"*, ce qui a pour effet de mettre ce symbole sur la pile. Ensuite, à la vue du deuxième symbole `b`, l'analyseur décide d'effectuer une action *"réduire"* et choisit comme alternative de production, l'alternative `{terminal}` de la production `$b`. Une action *"réduire"* correspond au dépilement d'un certain nombre de symboles et au réempilement d'autres symboles. Dans notre cas, `b` est dépilé et mis dans une liste qui est réempilée. La liste ainsi créée provient de la transformation de l'alternative `terminal : {→ [b]}`. C'est cette alternative qui a été utilisée pour effectuer l'action *"réduire"*. A présent, en sommet de pile, il y a une liste contenant un symbole `b`. Le prochain symbole en entrée est `b`. Avec cette combinaison, une action *"décaler"* va être déclenchée par l'analyseur syntaxique entraînant l'empilement du symbole `b`. Il s'agit du deuxième symbole `b` de la séquence des trois symboles. Maintenant avec la liste `[b]` et le symbole `b` en sommet de pile, la prochaine action effectuée par l'analyseur syntaxique est une action *"réduire"* et l'alternative `{nonterminal}` de la production `$b` est cette fois-ci utilisée. Ce qui a pour effet de dépiler les deux éléments au sommet de la pile, c'est à dire le symbole `b` et la liste `[b]` et réempiler une liste contenant le symbole `b` précédemment dépilé et l'autre symbole `b` contenu dans la liste, elle aussi précédemment dépilée. Avec la nouvelle liste

en sommet de pile et le dernier symbole `b` dans l'entrée, l'analyseur syntaxique effectue les même actions que précédemment entraînant ainsi la création d'une liste contenant trois symboles `b` et son empilement.

Lors de l'analyse syntaxique de la séquence des trois symboles `b` dans la paragraphe ci-dessus, il est important de remarquer que pour le premier symbole, l'alternative `{terminal}` de la production `$b` a été utilisée alors que pour les deux autres symboles, c'est l'alternative `{nonterminal}` de cette production qui a été utilisée. Cela vient démontrer l'affirmation précedente selon laquelle l'analyse syntaxique d'une séquence (liste) d'un symbole `X` commence toujours par l'alternative `{terminal}` de la production `$X` rajoutée par SableCC pour réaliser cette séquence.
Dans le tableau 5.1, le symbole `EOF` dénote la fin de l'entrée. Comme mentionné précédemment, ce symbole est toujours rajouté par l'analyseur syntaxique à la fin de l'entrée normale pour pouvoir arrêter l'analyse syntaxique. Les points de suspension `...` dénotent le fond de la pile. Le contenu n'est pas pertinent dans notre cas.

5.2.2.2 Production avec transformation

Prenons un exemple pour illustrer ce cas. Soit l'extrait de grammaire ci-dessous :

```
...
Productions
   prod            {→ ast_prod}
         = a b+    {→ New ast_prod([b.ta], [b.tc])};
   b               {→ ta tc}
         = ta tb tc {→ ta tc };


Abstract Syntax Tree
   ast_prod = ta+ tc+;
```

Après transformation, on obtient le résultat suivant :

```
...
```

```
Productions
   prod                        {→ ast_prod}
         = a $b                {→ New ast_prod([$b.ta], [$b.tc])};

   b                           {→ ta  tc }
         =ta tb tc             {→ ta  tc };

   $b                          {→ ta*  tc*}
         = {nonterminal} $b b{→ [$b.ta b.ta]  [$b.tc b.tc] }
         |  {terminal} b       {→ [b.ta]  [b.tc]};
```

Ce qui change dans ce cas est le fait que la production `$b` rajoutée pour réaliser l'occurrence multiple de `b` ne crée plus une liste de symboles `b` vu que `b` n'existe plus mais plutôt une ou des listes des symboles faisant partie de la transformation de production de `b` (→ `ta tc`), c'est à dire une liste de `ta` et une liste de `tc` (`ta` et `tc` étant dans ce cas des jetons). Les transformations d'alternatives sont dans ce cas-ci aussi ajustées en conséquence.

En résumé, SableCC tranforme la grammaire en une *notation BNF*, c'est-à-dire dépourvue des opérateurs `?`, `*` et `+` tout en respectant les choix de l'utilisateur indiqués dans la spécification originale.

5.2.3 L'opérateur *

Comme nous l'avons déjà mentionné, cet opérateur est une combinaison des opérateurs `+` et `?` parce qu'il permet de réaliser une séquence éventuellement vide du symbole qui le précède. Par rapport au traitement effectué pour l'opérateur `+`, le seul changement concerne le fait que dans le cas de l'opérateur `*`, il faut aussi ajouter la possibilité d'absence du symbole. Ainsi, si on considère l'extrait suivant correspondant à un fichier de spécification :

```
...
Productions
```

```
prod            {→ ast_prod}
     = a b*     {→ New ast_prod([b.ta], [b.tc])};
b               {→ ta tc }
     = ta tb tc {→ ta tc };

Abstract Syntax Tree
ast_prod = ta* tc*;
```

La transformation associe dans un premier temps `b*` à `(b+)?`, c'est à dire `{b+ | }` et ensuite, remplace `b+` par `$b`. Ce qui implique une transformation en conséquence de l'alternative dans laquelle le remplacement de `b+` a lieu. De la même manière, il est aussi nécessaire de mettre à jour l'alternative "dupliquée" dans laquelle on a éliminé le symbole `b` pour réaliser l'absence du symbole `b`.

Ainsi la grammaire précédente subit les étapes suivantes de transformation (seule la section `Productions` est modifiée, raison pour laquelle nous ne reproduisons que cette section ici).

Transformation 1 : (remplacement de `b*` par `{b+ | }` et ensuite remplacement de `b+` par `$b` → `{$b| })` :

```
...
Productions
prod                        {→ ast_prod}
      = a $b                {→ New ast_prod([$b.ta], [$b.tc])};
      | a                   {→ New ast_prod([$b.ta], [$b.tc])};
b                           {→ ta tc }
      = ta tb tc            {→ ta tc };
$b                          tt {→ ta*  tc*}
      = {nonterminal} $b b  {→ [$b.ta b.ta]  [$b.tc b.tc] }
      |  {terminal} b       {→ [b.ta]  [b.tc]};
```

Transformation 2 : (remplacement de `b.ta` par `$b.ta` et de `b.tc` par `$b.tc` dans la première transformation d'alternative et élimination de `b.ta` et de `b.tc` dans la seconde

transformation d'alternative) :

```
...
Productions
prod                        {→ ast_prod}
     = a $b                 {→ New ast_prod([$b.ta], [$b.tc])};
     | a                    {→ New ast_prod([ ], [ ])};
b                           {→ ta tc }
     = ta tb tc             { → ta tc };
$b                          {→ ta*  tc*}
     = {nonterminal} $b b   {→ [$b.ta b.ta]  [$b.tc b.tc] }
     |  {terminal} b        {→ [b.ta]  [b.tc]};
```

Remarque importante : Dans la transformation d'alternative ci-dessus, quand un élément est manquant, on ne le remplace pas par un `Null` mais on le supprime tout simplement. Cela est dû au fait que le terme `Null` ne peut pas apparaître dans une transformation d'alternative de type "liste".

5.2.4 La combinaison d'opérateurs ?, * et + dans la même production

Supposons une production de la grammaire dans laquelle on retrouve les trois opérateurs ?, + et *. La transformation dans ces cas suit exactement les mêmes règles que ce que nous avons vu jusque là. Nous allons maintenant présenter un exemple de ce type de transformation. Soit l'extrait suivant correspondant à un fichier de spécification de SableCC :

```
...
Productions
   prod          {→ ast_prod}
        = ta? tb c* d e+
                 {→ New ast_prod([ta c.ta], [c.tc e.tc], [e.te])};

   c               {→ ta tc }
```

```
        = ta tb tc  {→ ta tc } ;

    e               {→ tc te }
        = tc td te  {→ tc te } ;

Abstract Syntax Tree
    ast_prod = ta* tc* te* ;
```

1. Dans un premier temps, on applique les transformations correspondant à `ta ?` :

```
prod        {→ ast_prod}
      = {aprod1} ta tb c* d e+
            {→ New ast_prod([ta c.ta], [c.tc e.tc], [e.te])} ;
      | {aprod2} tb c* d e+
            {→ New ast_prod([c.ta], [c.tc e.tc], [e.te])} ;

c                       {→ ta tc }
      = ta tb tc        {→ ta tc } ;

e                       {→ tc te }
            = tc td te  {→ tc te } ;
```

   Ceci a pour effet l'élimination de l'opérateur `?` devant le symbole `ta` dans la première alternative et l'élimination de l'élément `ta ?` dans la deuxième alternative (alternative dupliquée) ainsi que l'élimination du terme `ta` dans la transformation d'alternative.

2. Transformation de `c*` (ajout de la production `$c`) : `c*` → `(c+) ?`

```
prod        {→ ast_prod}
     = {aprod11} ta tb $c d e+
```

```
            {→ New ast_prod([ta $c.ta], [$c.tc e.tc], [e.te])};
      | {aprod12} ta tb d e+
            {→ New ast_prod([ta], [e.tc], [e.te])};
      | {aprod21} tb $c d e+
            {→ New ast_prod([$c.ta], [$c.tc e.tc], [e.te])};
      | {aprod22} tb d e+
            {→ New ast_prod([ ], [e.tc], [e.te])};

c                           {→ ta tc }
      = ta tb tc            {→ ta tc };

e                           {→ tc te }
      = tc td te            {→ tc te };

$c                          {→ ta* tc* }
      = {nonterminal} $c c  {→ [$c.ta c.ta] [$c.tc c.tc]}
      |  {terminal} c       {→ [c.ta] [c.tc] };
```

3. Transformation de `e+` (remplacement de `e+` par `$e` et ajout de la production `$e`) :

```
prod     {→ ast_prod}
      = {aprod11} ta tb $c d $e
          {→ New ast_prod([ta $c.ta], [$c.tc $e.tc], [$e.te])};
      | {aprod12} ta tb d $e
          {→ New ast_prod([ta], [$e.tc], [$e.te])};
      | {aprod21} tb $c d $e
          {→ New ast_prod([$c.ta], [$c.tc $e.tc], [$e.te])};
      | {aprod22} tb d $e
          {→ New ast_prod([ ], [$e.tc], [$e.te])};

c                           {→ ta tc }
```

```
            = ta tb tc          {→ ta tc } ;

e           {→ tc te }
            = tc td te          {→ tc te } ;

$c          {→ ta* tc* }
        = {nonterminal} $c c
                                {→ [$c.ta c.ta] [$c.tc c.tc]}
        |  {terminal} c
                                {→ [c.ta] [c.tc] } ;

$e                              {→ tc* te* }
        = {nonterminal} $e e    {→ [$e.tc e.te] [$e.tc e.te]}
        |  {terminal} e         {→ [e.tc] [e.te] } ;
```

Cette production (`prod`) en *notation* EBNF a occasionné la création de deux nouvelles productions (`$c` et `$e`) pour réaliser les occurrences multiples des symboles `c` et `e`. Elle a aussi occasionné la création de trois nouvelles alternatives. L'ancienne et originellement unique alternative de la production `prod` n'est pas en reste de toutes ces modifications car elle a en effet aussi subi des changements. La grammaire finalement obtenue est conforme à la grammaire originale en ce sens qu'elle peut analyser les mêmes programmes d'entrée mais aussi l'arbre syntaxique résultant de l'analyse syntaxique des programmes d'entrée reste celui spécifié originellement par l'auteur de la grammaire. Au vu de ce qui précède, les transformations internes faites par SableCC n'affectent d'aucune façon l'usager.

5.3 Résumé

Nous avons expliqué dans ce chapitre les transformations internes faites aux grammaires d'entrée de SableCC pour transformer la notation EBNF en notation BNF. Nous avons aussi vu que ces transformations sont dues au fait que l'algorithme d'analyse syn-

taxique mis en oeuvre dans SableCC ne supporte pas la notation EBNF. Nous avons dans un second temps présenté les ajustements automatiquement faits sur les transformations spécifiées par l'usager du fait du passage de la notation EBNF à une notation BNF.

Chapitre VI

ALGORITHME DE CONSTRUCTION DE L'ARBRE SYNTAXIQUE

Le but premier des transformations d'arbres syntaxiques de SableCC est de permettre la construction d'un arbre syntaxique abstrait, c'est-à-dire un arbre syntaxique ne retenant que certains des éléments du programme original. Comme nous l'avons déjà vu dans les chapitres précédents, SableCC génère le code de construction de l'arbre syntaxique en se basant sur la grammaire (section `Productions` du fichier de spécification). Dans le cas de SableCC3[1], seule la partie des transformations à l'intérieur de cette grammaire est prise en compte pour la construction de l'arbre syntaxique réduit dit *arbre syntaxique abstrait*. Dans ce chapitre, nous allons décrire l'algorithme utilisé pour la construction de cet arbre. Pour ce faire, nous parlerons du fonctionnement de l'analyseur syntaxique car c'est à la fin de cette phase que l'arbre syntaxique est construit. Nous verrons dans un premier temps le fonctionnement sans les transformations d'arbres syntaxiques et ensuite nous mettrons l'accent sur la manière dont les changements apportés par les transformations d'arbres syntaxiques affectent ce fonctionnement. Pour finir, nous décrirons la façon dont les différents noeuds constituant cet arbre sont construits et y sont insérés.

---

[1] Version de SableCC dans laquelle nous avons intégré les transformations d'arbres syntaxiques.

6.1 L'analyseur syntaxique sans les transformations d'arbres syntaxiques

Le rôle de l'analyseur syntaxique dans les compilateurs et dans les interpréteurs consiste à vérifier qu'un programme rédigé dans un langage de programmation en particulier est conforme à la grammaire de ce langage. La structure syntaxique d'un langage de programmation est décrite par une grammaire non contextuelle (en abrégé, grammaire), ou notation *BNF*(*Backus-Naur Form*). C'est donc à partir de cette grammaire qu'il convient de construire l'analyseur syntaxique. Il existe trois types généraux d'analyseurs syntaxiques (Aho, Sethi et Ullman, 2000) pour les grammaires :

1. les méthodes universelles comme celles de Coke-Younger-Kasami et d'Earley peuvent analyser une grammaire quelconque. Cependant l'utilisation de ces méthodes dans les compilateurs serait inefficace car elles procureraient d'assez faibles rendements (Aho, Sethi et Ullman, 2000) ;
2. les deux grandes méthodes utilisées en compilation sont soit descendantes soit ascendantes. Comme leur nom l'indique, les méthodes d'analyse ascendante sont basées sur un analyseur qui construit des arbres d'analyse en partant du bas (les feuilles) vers le haut (la racine) ;
3. les méthodes d'analyse descendante quant à elles utilisent un analyseur qui construit les arbres du haut vers le bas. Dans les deux cas, l'entrée de l'analyseur syntaxique est parcourue de la gauche vers la droite, un symbole à la fois.

Les méthodes d'analyse ascendante ou descendante les plus efficaces travaillent uniquement sur des sous-classes de grammaires, mais plusieurs de ces sous-classes, comme les grammaires *LL* (***L****eft to right scanning of the input constructing a* ***L****eft most derivation in reverse* ce qui signifie "*parcours de l'entrée de la gauche vers la droite en construisant une dérivation gauche inverse*") et *LR* (***L****eft to right Scanning ;* ***R****ight most derivation*, ce qui signifie "*parcours de l'entrée de la gauche vers la droite en construisant une dérivation droite inverse*"), sont suffisamment expressives pour décrire la plupart des constructions syntaxiques des langages de programmation. Les grammaires *LL* sont avantageuses en ce sens qu'il est plus facile d'écrire à la main des analyseurs capables

de les traiter alors que les analyseurs traitant les grammaires *LR* (plus générales que les grammaires *LL*) sont le plus souvent générés par des outils automatiques[2] (Aho, Sethi et Ullman, 2000).

SableCC traite les grammaires de la classe *LALR(1)* (***L**ook**A**head **LR***). Il s'agit d'une classe de grammaire qui est un sous-ensemble des grammaires *LR*. Ces grammaires sont intéressantes car le coût de l'analyse des programmes conformes à ces grammaires est un temps linéaire par rapport à la taille de l'arbre syntaxique généré mais aussi parce que ces analyseurs ont l'avantage de pouvoir être mis en oeuvre en utilisant des tables d'analyse réduites par rapport à celles des analyseurs *LR(1)*. De plus, ces grammaires permettent de décrire la plupart des constructions syntaxiques classiques des langages de programmation. Le "1" dans les classes de grammaires *LR(1)*, *LALR(1)* et *LL(1)* indique le nombre de symboles de prévisions utilisés pour prendre les décisions d'analyse lors de l'analyse syntaxique.

Un analyseur syntaxique *LALR(1)* est basé sur un principe de décalage-réduction. Ce type d'analyseur construit en sortie un arbre d'analyse pour le programme en entrée en commençant par les feuilles (le bas) et en remontant vers la racine (le haut).

La description complète du fonctionnement de ce type d'analyseur syntaxique n'étant pas le thème de ce mémoire, nous nous en tiendrons donc au mécanisme général.

6.1.1 Principe de fonctionnement

La figure 6.1 montre les différents composante d'un analyseur syntaxique. Comme on peut le remarquer, il est constitué d'un tampon d'entrée, d'une pile, d'un programme d'analyse et de tables d'analyse.

Le principe de fonctionnement est le suivant :

> le programme d'analyse lit les unités lexicales l'une après l'autre dans le tampon d'entrée. Il utilise une pile pour y ranger des chaînes de la forme $s_0X_1s_1X_2s_2 \ldots X_ms_m$, où s_m est au sommet. Chaque X_i correspond à un symbole

[2] Il existe cependant aussi des outils traitant les grammaires *LL*.

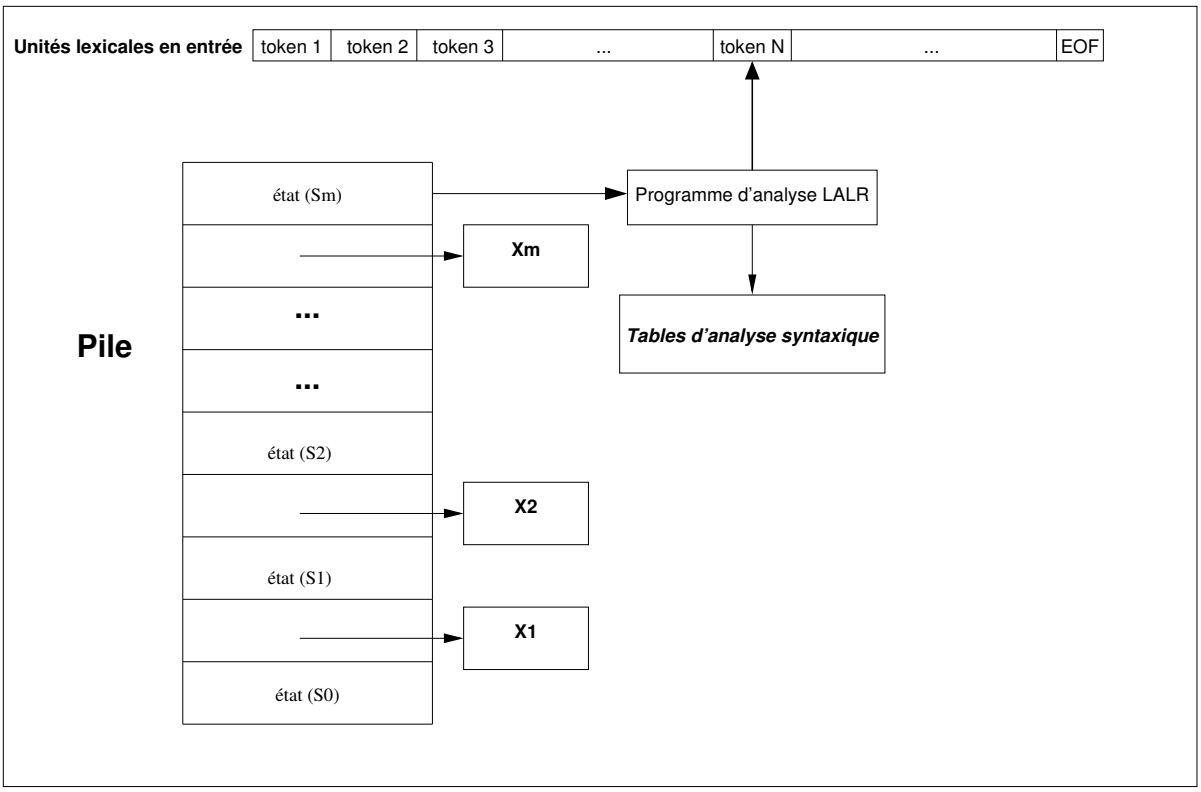

Figure 6.1 Constituants d'un analyseur syntaxique de SableCC **sans** intégration de transformations d'arbres syntaxiques

de la grammaire (terminal ou non-terminal) et chaque s_i est un symbole appelé *état*. Chaque état résume l'information contenue dans la pile en dessous de lui ; la combinaison du numéro de l'état en sommet de pile et du symbole d'entrée courant est utilisée pour indexer les tables d'analyse syntaxique et déterminer la prochaine action d'analyse à effectuer. Les tables d'analyse peuvent être vues comme un automate fini déterministe (voir chapitre 2). En effet, elles permettent à chaque étape du processus d'analyse de déterminer l'action à effectuer. Le programme dirigeant l'analyseur *LALR* (ou programme d'analyse) se comporte de la façon suivante. Il détermine s_m, l'état en sommet de pile, et a_i, le symbole terminal d'entrée courant qui provient de l'analyseur lexical. Il consulte alors l'automate (tables d'analyse) avec les

informations suivantes : l'état s_m et le terminal a_i. Les différentes transitions de l'automate peuvent être :

1. *décaler* s, où s est un état ;
2. *réduire* par une production de la grammaire ($A = \{alt_i\}\ elem_1\ elem_2\ ...\ elem_r$) ;
3. *accepter* ;
4. *signaler une erreur.*

Une configuration de l'analyseur syntaxique avant la lecture du prochain symbole d'entrée a_i est le couple : ($s_0\ X_1 s_1\ X_2 s_2\ ...\ X_m s_m,\ \ a_i\ a_{i+i}\ \ ...\ \ a_n$) où "$s_0\ X_1 s_1\ X_2 s_2\ ...\ X_m s_m$" représente le contenu de la pile et "$a_i\ a_{i+i}\ \ ...\ \ a_n$" la chaîne d'entrée restant à analyser. Les X_i correspondent à des noeuds créés à partir des alternatives de la grammaire ou à des jetons. Ces noeuds correspondent en fait à des sous arbres de l'arbre syntaxique final. Les différents cas possibles sont les suivants :

- *décaler s*, la configuration de l'analyseur syntaxique devient ($s_0\ X_1 s_1\ X_2 s_2\ ...\ X_m s_m\ \mathbf{a_i}$ s, $a_{i+i}\ ...\ a_n$). Ici, le symbole a_i et l'état s correspondant à `transition` ($s_m,\ a_i$) dans l'automate (tables d'analyse) sont mis sur la pile ;
- *réduire par une production de la grammaire* $A = \{alt_i\}\ elem_1\ elem_2\ ...\ elem_r$; Étant donné qu'une production peut avoir une ou plusieurs alternatives, la réduction vers une production de la grammaire correspondrait plutôt à une réduction vers une des alternatives de l'une des productions de la grammaire ($\{alti\}\ elem_1\ elem_2\ ...\ elem_r$;). L'action *réduire* revient à dépiler `r (r > 0)` symboles de la pile avec les états correspondants et à empiler le symbole A, plus précisement le symbole `AAltiA` dans le cas de SableCC. La configuration de l'analyseur syntaxique devient donc ($s_0\ X_1\ s_1\ X_2\ s_2\ ...\ X_{m-r}\ s_{m-r}$ `AAltiA s,` $a_i\ a_{i+i}\ ...\ \}a_n$ \$). Le nouveau symbole `AAltiA` empilé est en réalité un noeud de l'arbre syntaxique. Quand les r éléments sont dépilés, on construit, conformément à la nomenclature de SableCC le noeud `AAltiA = new AAltiA(`$elem_1$, $elem_2$, ..., $elem_r$`)` et c'est ce noeud qui est empilé. Donc à chaque fois qu'il y a une réduction, on construit un noeud de type X_i, c'est-à-dire un noeud correspondant à une des alternatives d'une des productions de la grammaire et

on empile ce noeud.

Remarque : le noeud construit étant en réalité un des noeuds de l'arbre syntaxique en construction, il s'agit donc d'un sous arbre avec une racine `AAltiA` et `r` fils (`r`, un entier naturel) qui sont : $\texttt{elem}_1$, $\texttt{elem}_2$, ..., $\texttt{elem}_\texttt{r}$;

– *accepter*. Dans ce cas, l'analyse est terminée et l'arbre syntaxique est complètement construit. Cela signifie qu'à l'étape précédente, on a fait une réduction par une des alternatives de la production dont le symbole gauche est l'axiome de la grammaire (`S = {alti}` $\texttt{elem}_1$ $\texttt{elem}_2$ ... $\texttt{elem}_\texttt{r}$) ;

– *signaler une erreur*. On arrête l'analyse et on signale à l'usager un message indiquant la source de l'erreur.

À la fin de l'analyse syntaxique, étant donné qu'une réduction est faite vers l'axiome de la grammaire, la construction de l'arbre syntaxique final est alors complétée.

6.1.2 Pseudo-code de l'algorithme d'analyse syntaxique

Le pseudo-code de la description de l'algorithme dans le paragraphe précédent sur le principe de fonctionnement de l'analyseur syntaxique est présenté à la figure 6.2. Initialement, l'analyseur a $\texttt{s}_0$ sur la pile, où $\texttt{s}_0$ est l'état initial, et une séquence d'unités lexicales (`W`) suivi du symbole de fin de fichier `$` dans le tampon d'entrée. L'analyseur exécute le programme de la figure 6.2 jusqu'à ce qu'il rencontre une action *accepter* ou encore une action *erreur* auquel cas il s'arrête.

Le pseudo-code de la figure 6.2 effectue l'analyse syntaxique mais permet aussi de construire l'arbre syntaxique des programmes en entrée. La construction de l'arbre syntaxique est effectuée par la procédure `construireNoeud` (ligne 10 du pseudo-code) qui prend des symboles de la grammaire et construit un noeud de l'arbre syntaxique avec ces symboles. Dans le cas de SableCC, la pile contient en réalité un type de données regroupant un symbole de grammaire et un état. Ce type est appelé *State*. Mais pour faciliter l'explication, nous avons préféré considérer le modèle standard de la pile des analyseurs LR. D'autre part, ce n'est pas vraiment un symbole de la grammaire qui est stocké sur la pile mais directement un noeud de l'arbre syntaxique. Cela implique

AnalyseurLALR(**Entrée** : une séquence d'unités lexicales ***W***, tables d'analyse ***Tables*** ; **Sortie** : arbre d'analyse ***AST***)

```
0 :   empiler s0 ;
1 :   initialiser le pointeur source ps sur le premier symbole de W$
2 :   répéter indéfiniment
3 :       soit s l'état en sommet de pile et a le symbole pointé par ps
4 :       si DéterminerProchaineAction(s, a) = decaler s' alors
5 :           empiler a puis s' ;
6 :           avancer ps sur le prochain symbole en entrée
7 :       fin
8 :       sinon si DéterminerProchaineAction(s, a) = reduire par A → β alors
9 :           dépiler (2 * |β| symboles) ;
10 :          NoeudA = construireNoeud(A, 2 * |β| symboles) ;
11 :          soit s' le nouvel état en sommet de pile
12 :          empiler NoeudA puis DéterminerSuccesseur(s, A) ;
13 :      fin
14 :      sinon si DéterminerProchaineAction(s, a) = accepter alors
15 :          retourner NoeudA et arrêter ;
16 :      sinon erreur
17 :  fin
```

Figure 6.2 Pseudo-code de l'algorithme d'analyse syntaxique

quelques modifications dans l'algorithme du pseudo-code de la figure 6.2. Notamment :

1. ligne 9 : on dépile $|\beta|$ symboles et non 2 * $|\beta|$ symboles ;
2. ligne 12 : on empile un type `State` regroupant le nouveau noeud construit `NoeudA` puis l'état successeur ;
3. ligne 5 : empiler `State = (a, s')` ;

Le symbole `a` empilé à la ligne 5 du pseudo-code est en réalité une unité lexicale car il s'agit d'un des symboles d'entrée de l'analyseur syntaxique. Jusque là, la pile contient donc des éléments de type `State`. Ces éléments sont des couples composés d'un état et d'un symbole qui peut être soit une unité lexicale, c'est-à-dire un jeton, soit un noeud créé par la procédure `construireNoeud(liste : Symboles)`. On suppose que `dépiler(N)` retourne une liste contenant des éléments suivant leur ordre de dépilement. Le pseudo-code de la procédure `construireNoeud` est présenté ci-dessous :

procédure **<u>construireNoeud</u>**(**Entrée** : `A` → [3]symboleGrammaire, `symboles` →liste ;
Sortie : Noeud→noeudDeLarbreSyntaxique)
Début
  `n` = |`symboles`| ;
  `TypeNoeud` = DéterminerTypeNoeud(`A`) ;
  pour `i` allant de `1` à `n` faire
    ($\texttt{symb}_\texttt{i}$, $\texttt{etat}_\texttt{i}$) = récupérerÉlémentALaPosition(`symboles`, `i`) ;
  fin pour
  `Noeud` = CréerNoeud `TypeNoeud`($\texttt{symb}_\texttt{n}$, $\texttt{symb}_{\texttt{n}-1}$, ..., $\texttt{symb}_1$, $\texttt{symb}_0$) ;
  retourner `Noeud` ;
fin
`symboles` correspond aux symboles de l'alternative de la production dont on fait l'analyse. Ces symboles proviennent de la pile d'analyse. Ils ont été dépilés juste avant (procédure *AnalyseurLALR* figure 6.2).

[3]Les flèches de l'entête de la procédure séparent le nom de ces paramètres situés à gauche de la flèche, de son type qui est situé à droite de cette même flèche.

Pile	**Entrée**	**Action**
(1) 0	`abc$`	*décaler*
(2) 0TA1	`bc$`	*décaler*
(3) 0TA1TB2	`c$`	*décaler*
(4) 0TA1TB2TC3	`$`	*réduire* par `B = abc`
(5) 0AB4	`$`	*réduire* par `S = {deuxieme} B`
(6) 0DeuxiemeS5	`$`	*accepter*

Tableau 6.1 Exemple d'une trace d'exécution de l'analyseur syntaxique de SableCC

6.1.3 Exemple de fonctionnement de l'analyseur syntaxique

Soit la grammaire SableCC suivante :

Spécification 6.1

```
Tokens
  a = 'a';
  b = 'b';
  c = 'c';
Productions
  S = {premier} A  |
      {deuxieme} B ;
  A = b c a;
  B = a b c;
```

Considérons la chaîne suivante "abc" donnée en entrée à l'analyseur syntaxique généré par SableCC pour cette grammaire. Observons le comportement de la pile d'analyse lors de l'analyse de cette chaîne. Notons `TA`, `TB`, `TC`, les unités lexicales retournées par l'analyseur lexical de SableCC quand les caractères `a`, `b` et `c` sont rencontrés et `TEOF` quand la fin de fichier est rencontrée. Comme mentionné dans le pseudo-code *AnalyseurLALR* de la figure 6.2, au début on empile l'état s_0. Ici, nous allons empiler l'état `0`. Ensuite

commence l'analyse proprement dite. L'entrée de l'analyseur syntaxique est uniquement composée d'unités lexicales, les mots apparaissant dans le texte ne sont en réalité lus que par l'analyseur lexical. La séquence donnée à analyser à l'analyseur syntaxique est donc "`TA TB TC TEOF`". Au début, la configuration de l'analyseur syntaxique est :

`(0, abc$)`

1. Après lecture de `TA` ("a") et consultation des tables d'analyse, l'action à effectuer est *décaler* et la configuration de l'analyseur devient :
   `(0TA1, bc$)`
   L'entrée restant à analyser devient "`bc$`" car lors d'une action *décaler*, le pointeur `ps` est avancé sur le prochain symbole d'entrée.
2. La même action *décaler* se répète quand les prochains mots `TB` ("`b`") et `TC` ("`c`") sont vus en entrée. La configuration de l'analyseur après le mot `TC` devient :
   `(0TA1TB2TC3, $)`
3. À partir de ce moment, à la vue du symbole "`$`", les tables d'analyse indiquent qu'il faut effectuer une action *réduire* par la production `B = abc`. Ainsi, on détermine `|B|` (le nombre de symboles de l'alternative) qui vaut 3 car elle est composée des 3 symboles `a`, `b` et `c`. Ensuite, on dépile 2 * 3 symboles c'est-à-dire "`3TC2TB1TA`" et, à partir de ce qui est dépilé, on construit un nouveau noeud en utilisant le type de la production "`B = abc`" c'est-à-dire `AB` conformément à la nomenclature SableCC. Une fois ce noeud construit, il est empilé à son tour. Le nouveau noeud construit est : `Noeud = CréerNoeud AB(TA, TB, TC)`. Ceci donne un noeud de l'arbre avec comme racine `AB` et trois fils qui sont `TA`, `TB` et `TC`. La configuration de l'analyseur syntaxique devient :
   `(0AB4, $)`
4. Ensuite, de l'état 4 et avec le symbole d'entrée "`$`", l'action *réduire* par la production "`S = {deuxieme} B`" est indiquée par les tables d'analyse. Le noeud `Noeud = CréerNoeud ADeuxiemeS(AB)` est construit et empilé, ce qui amène l'analyseur syntaxique dans la configuration suivante :
   `(0ADeuxiemeS5, $ )`

5. Finalement de l'état 5 et à la vue du symbole "$", les tables d'analyse indiquent l'action `accepter`. L'analyse s'arrête et le noeud `ADeuxiemeS` est retourné. Ce noeud constitue l'arbre finalement construit.

On peut assez facilement se rendre compte que le noeud `ADeuxiemeS` représente un arbre et aussi que cet arbre est conforme à la structure de la grammaire. Pour nous en convaincre, on peut remarquer qu'à l'étape 4, le noeud construit est de type `ADeuxiemeS`, c'est-à-dire une des alternatives de la production `S`, et que ce noeud a un seul fils qui est de type `AB` ce qui est conforme à la grammaire (`S = {deuxieme} B`). Ensuite, à l'étape 3, on se rend aisément compte du fait que le noeud `AB` a 3 fils qui sont dans l'ordre de type `TA`, `TB` et `TC`, ce qui est conforme à la production (`B = a b c`) de la grammaire. Ces fils correspondent à des feuilles dans l'arbre construit. Ainsi l'arbre dont la racine est le noeud `ADeuxiemeS` correspond bien à l'arbre syntaxique devant être construit pour la chaine "`abc`" et la grammaire de la spécification 6.1.

6.2 Intégration des transformations d'arbres syntaxiques dans l'analyseur syntaxique

Pour construire l'arbre syntaxique à partir d'une grammaire incluant une spécification de transformation d'arbres syntaxiques, il a fallu modifier la routine de construction de l'arbre intégrée à l'analyse syntaxique. L'analyse syntaxique du programme en elle même n'est pas affectée car la partie de la grammaire qui définit la structure du programme n'est pas modifiée par la spécification des transformations d'arbres syntaxiques. Dans cette nouvelle optique, deux changements majeurs ont été apportés à l'analyseur syntaxique, plus précisément au processus de construction de l'arbre syntaxique :

- la pile utilisée par l'analyseur pour stocker les états et les symboles de la grammaire ;
- la procédure de création des noeuds.

6.2.1 La pile de l'analyseur syntaxique

Comme nous l'avons vu précédemment, la pile contient deux types de symboles (les états et les noeuds ou feuilles de l'arbre syntaxique). Les noeuds de l'arbre syntaxique sont empilés lors de l'analyse syntaxique, plus précisément au moment d'une action *réduire*. Une action *réduire* correspondait jusque là (version antérieure de SableCC) à construire un noeud dont le type est celui de l'alternative courante. A présent, avec les transformations d'arbres syntaxiques, une alternative peut se transformer en un ou plusieurs autres symboles. Il faut donc pouvoir stocker sur la pile ce ou ces symboles. C'est la raison pour laquelle la pile a été modifiée pour contenir non plus un seul symbole mais une liste pouvant contenir un ou plusieurs symboles. Les états, quant à eux, y sont toujours stockés de la même manière.
Un diagramme de l'analyseur montrant le contenu de la pile dans ce nouveau contexte est présenté à la figure 6.3. Comme on peut le voir sur cette figure, ce ne sont plus des noeuds de l'arbre syntaxique qui sont directement stockés sur la pile mais une liste d'un ou plusieurs de ces noeuds. Ceci permet à une étape ultérieure du processus d'analyse syntaxique de pouvoir ne retenir que les éléments de cette liste qu'on veut inclure dans l'arbre. La liste empilée contient en fait les éléments spécifiés dans la transformation de l'alternative dont la réduction a occasionné cet empilement. Nous allons montrer cela par un exemple. Soit l'extrait suivant représentant une grammaire SableCC avec des transformations d'arbres syntaxiques :

Exemple 6.1

```
Productions
...
 decl_fonction                          {→ T.id params }
   = T.id l_par params ? r_par          {→ T.id params } ;

 def_fonction                           {→params }
   = T.id decl_fonction corps_fonction  {→ decl_fonction.params } ;  ...
```

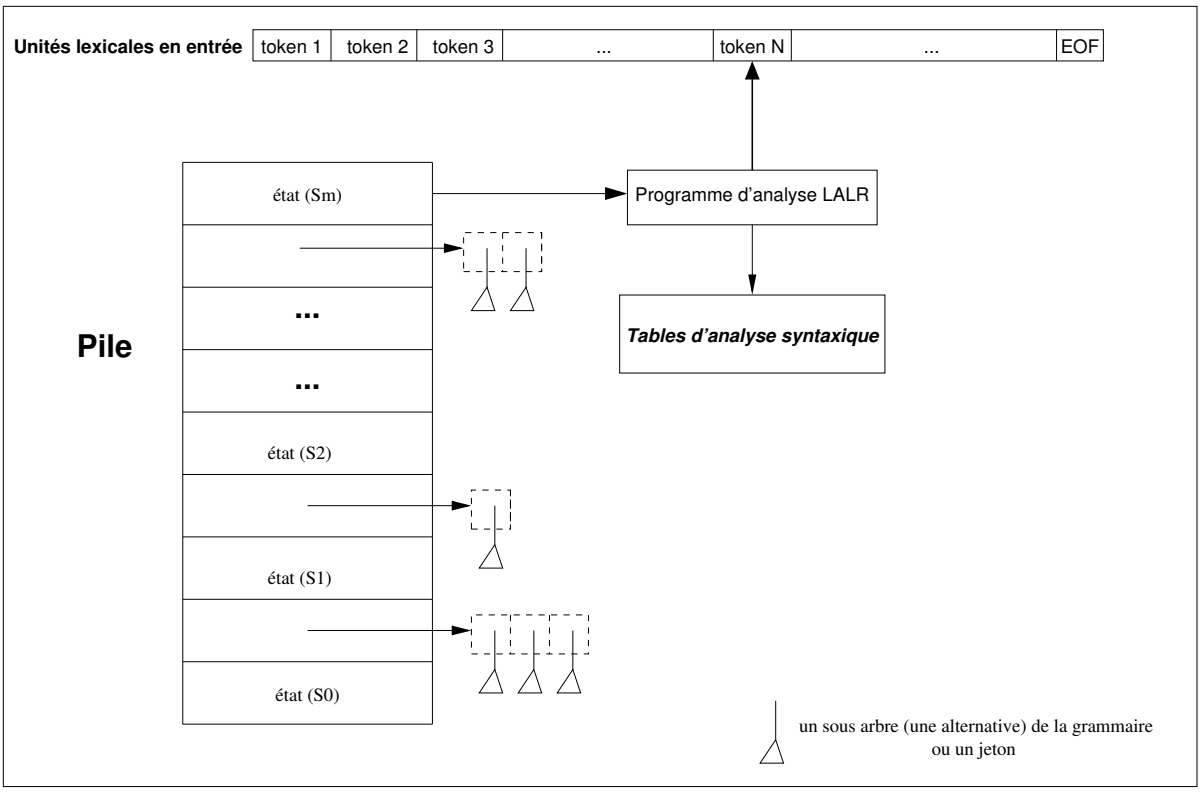

Figure 6.3 Constituants d'un analyseur syntaxique de SableCC **avec** intégration de transformations d'arbres syntaxiques

L'extrait de grammaire de l'exemple 6.1 montre deux productions avec des transformations d'arbres syntaxiques. Ceci implique donc qu'à chaque fois qu'une des alternatives de ces productions est reconnue, c'est une liste contenant les éléments de la transformation d'alternative appropriée qui est empilée plutôt qu'un simple noeud.

Concrètement, quand par exemple la phrase `methode(int un_argument)` est reconnue par l'analyseur syntaxique (phrase dérivée par la production `decl_fonction = T.id l_par params ? r_par`), seule une liste contenant les éléments `methode` et `int un_argument` est mise sur la pile. L'analyseur s'est donc débarassé des parenthèses. Mieux encore, "`methode`" et "`int un_argument`" ne sont plus emballés dans un noeud,

ils sont accessibles directement via des positions dans la liste.
Dans la production `decl_fonction`, cette flexibilité est utilisée. En effet, cette production se transforme un élément de type `params`. Et pour l'unique alternative de cette production qui est composée des trois symboles `T.id`, `decl_fonction` et `corps_fonction`, la transformation associée est `decl_fonction.params`. Cette transformation signifie que pour cette alternative, on ne conserve que le symbole `decl_fonction`. En réalité, seulement certains éléments de ce symbole sont conservés. En effet, on ne retient de ce symbole que l'élément `params` alors que la transformation de production correspondant à ce symbole possède deux éléments (`T.id` et `params`). Ce type de transformation permet à chaque étape du processus de construction de l'arbre syntaxique de ne garder que les éléments qui sont vraiment nécessaires dans l'arbre et ainsi de se débarasser de ceux qui s'avèrent inutiles.
Pour mettre en oeuvre ce mécanisme, il a fallu changer la procédure `construireNoeud` qui jusque là permettait de construire les noeuds de l'arbre syntaxique. La création d'un nouveau noeud doit à présent être basée sur les transformations de productions et d'alternatives spécifiées. La première chose qui change concerne le type du noeud créé après chaque action *réduire* lors de l'analyse. En effet une liste de noeuds est créée à la place d'un simple noeud. Pour cette liste, nous avons utilisé le type *ArrayList* des collections de Java pour représenter les listes à mettre sur la pile. Ce type de données a l'avantage d'être générique mais aussi dynamique. En d'autres termes, on peut y insérer n'importe quel type de noeud (*Object* en *Java*) et leur taille n'est pas fixée de façon statique. Un autre facteur qui nous a aussi motivé à utiliser cette structure est le fait qu'on peut accéder à n'importe lequel des éléments de la liste par son indice qui va de `0` à `N` $-$ `1`, `N` étant le nombre d'éléments dans la liste. Cette liste peut aussi contenir un objet dont la référence pointe vers `null`.

6.2.2 La procédure *ConstruireNoeud*

Voyons maintenant comment la procédure construireNoeud a été modifiée pour permettre la construction des différents types de noeuds. Cinq principaux types de trans-

formations d'arbres syntaxiques ont été introduits dans SableCC3, impliquant ainsi cinq différents types de noeuds à construire. Nous discuterons plus tard de certains détails d'implémentation concernant ces transformations d'alternative. Avant cela, rappelons la forme générale d'une production incluant des transformations d'arbre syntaxique :

Exemple 6.2

$$
\begin{array}{ll}
\texttt{prod} & \{\rightarrow \texttt{noeud\_ast}_1\ \texttt{noeud\_ast}_2 \ldots \texttt{noeud\_ast}_N\ \} \\
\quad = \{\texttt{alt}_1\}\ \texttt{elem}_1\ \texttt{elem}_2 \ldots \texttt{elem}_{L_1} & \\
\qquad\qquad & \{\rightarrow \texttt{alt}_1\texttt{\_transform}_1\ \texttt{alt}_1\texttt{\_transform}_2 \ldots \texttt{alt}_1\texttt{\_transform}_{N_1}\ \} \\
\quad | \{\texttt{alt}_2\}\ \texttt{elem}_1\ \texttt{elem}_2 \ldots \texttt{elem}_{L_2} & \\
\qquad\qquad & \{\rightarrow \texttt{alt}_2\texttt{\_transform}_1\ \texttt{alt}_2\texttt{\_transform}_2 \ldots \texttt{alt}_2\texttt{\_transform}_{N_2}\ \} \\
\quad | \ldots & \\
\quad | \{\texttt{alt}_k\}\ \texttt{elem}_1\ \texttt{elem}_2 \ldots \texttt{elem}_{L_k} & \\
\qquad\qquad & \{\rightarrow \texttt{alt}_k\texttt{\_transform}_1\ \texttt{alt}_k\texttt{\_transform}_2 \ldots \texttt{alt}_k\texttt{\_transform}_{N_k}\ \} \\
\quad ; &
\end{array}
$$

$\texttt{N}$, $\texttt{L}_1 \ldots \texttt{L}_k$ et $\texttt{N}_1 \ldots \texttt{N}_k$ sont tous des entiers naturels. $\texttt{N}$ représente le nombre de symboles dans la transformation de la production `prod`. $\texttt{L}_1 \ldots \texttt{L}_k$ et $\texttt{N}_1 \ldots \texttt{N}_k$ représentent respectivement le nombre de symboles dans les alternatives $\texttt{alt}_1$, $\texttt{alt}_2 \ldots \texttt{alt}_k$ de cette même production.

À chaque fois que la production `prod` ci-dessus est reconnue par l'analyseur syntaxique, plus précisément à chaque fois qu'une des alternatives de cette production est reconnue, une liste (`ArrayList`) contenant exactement `N` éléments est créée et est mise sur la pile. Ceci suppose donc que si `prod` apparaît comme symbole dans une alternative d'une autre production de la grammaire, pour accéder à chacun des éléments de sa transformation, il faut écrire $\texttt{prod.noeud\_ast}_\texttt{i}$ où `i` est un nombre compris entre `1` et `N` (le type de l'élément $\texttt{prod.noeud\_ast}_\texttt{i}$ est le même que l'élément $\texttt{noeud\_ast}_\texttt{i}$ spécifié dans la transformation de production de `prod`). Supposons maintenant qu'une alternative d'une autre production de la grammaire utilise le symbole `prod` et que dans la transformation de cette alternative, on ne retienne que certains éléments comme dans

l'exemple 6.3.

Exemple 6.3

```
prod2                          {→ noeud_ast1 noeud_astN }
      = {alt} elem prod elem2 {→ prod.noeud_ast1 prod.noeud_astN } ;
```

Il faut comprendre qu' étant donné que la production `prod` est transformée, on ne peut plus écrire `prod` dans une transformation d'alternative pour référencer le noeud correspondant au symbole `prod` car au niveau de l'arbre syntaxique, quand le symbole `prod` est reconnu par l'analyseur syntaxique, une liste de noeuds "$\texttt{noeud\_ast}_1$", "$\texttt{noeud\_ast}_2$", ..., "$\texttt{noeud\_ast}_N$" est construit en lieu et place d'un seul noeud de type `prod`. Pour construire la liste de noeud correspondant à la production `prod2`, il faut aller chercher les éléments correspondant à $\texttt{prod2.noeud\_ast}_1$ et $\texttt{prod2.noeud\_ast}_N$ dans la liste des éléments correspondant à `prod`. Ceci est effectué dans SableCC en allant tout simplement chercher dans cette liste les éléments aux positions *1* et *N*. Le nouveau pseudo-code de la procédure *construireNoeud* est :

procédure **<u>construireNoeud</u>**(**Entrée** : `AltTransformListe` → [4]liste, `symboles` →liste ;
Sortie : *Noeud*→liste)

1 : Début
2 : créer une nouvelle liste vide *nouvListe*
3 : pour chaque élément E de la liste des transformations d'alternative *AltTransformListe*
4 : // chercher l'élément concerné dans la liste symboles
5 : $\texttt{symb}_\texttt{i}$ = chercherElement(E, *symboles*) ;
6 : *nouvelElement* = **transformer**($\texttt{symb}_\texttt{i}$, E) ;
7 : ajouter(*nouvelElement*, *nouvListe*) ;
8 : fin pour
9 : *Noeud* = nouvListe ;
10 : retourner *Noeud*
11 : fin

AltTransformListe correspond à une liste contenant les différents éléments de la transformation d'alternative (`alt_transform`$_1$, `alt_transform`$_2$, etc.)
symboles correspond à une liste contenant les éléments de l'alternative qui ont été dépilés

Avant d'entrer dans les détails d'implantation et voir comment chacune des différentes transformations d'alternative est implantée, il convient de comprendre le fonctionnement général de la procédure `ConstruireNoeud`.
Comme précédemment mentionné, l'analyse syntaxique se fait en se basant sur des alternatives de la grammaire. Au fur et à mesure que sont reconnus des symboles correspondant à des éléments d'une alternative, une action "*décaler*" est effectuée entraînant l'ajout de ces symboles dans une liste nouvellement créée et l'empilement de cette liste. Quand l'analyseur syntaxique se rend à la fin de l'alternative, à ce moment là, une action "*réduire*" doit être effectuée. Cela implique de dépiler exactement un certain nombre `X` d'éléments de la pile. `X` correspond en fait au nombre d'éléments que comporte l'alternative qui est en train d'être utilisée pour l'analyse syntaxique. Ces différents `X` éléments dépilés sont en fait des listes car nous avons vu que la pile contient à présent des listes. Ces listes contiennent des noeuds qui vont être utilisés pour construire l'arbre syntaxique. Nous allons à travers quelques exemples illustrer le comportement de la pile d'analyse.

Exemple 6.4

```
Tokens   a = 'a' ;
         b = 'b' ;
         c = 'c' ;
         d = 'd' ;
Productions
  ...
  prod           {→ b d } =
       a b c d   {→ b d} ;
```

Une phrase pouvant être dérivée à partir de la production `prod` de l'exemple ci-dessus, est "`abcd$`". Pour faire l'analyse syntaxique de cette phrase, l'analyseur syntaxique empile successivement les symboles correspondant aux caractères `a`, `b`, `c` et `d`. Ces symboles avant d'être empilés sont mis dans des listes. Ce qui donne la configuration suivante pour l'analyseur syntaxique :

```
-1- (0,                          TATBTCTD$)
-2- (0[TA]1,                       TBTCTD$)
-3- (0[TA]1[TB]2,                    TCTD$)
-4- (0[TA]1[TB]2[TC]3,                 TD$)
-5- (0[TA]1[TB]2[TC]3[TD]4,              $)
-6- (0[TB TD]5,                          $)
```

Les crochets (`[ ]`) dans la pile d'analyse ci-dessus indiquent que ce sont des listes qui sont empilées.

On peut en effet constater que l'analyseur syntaxique empile successivement les symboles correspondant aux caractères `a`, `b`, `c` et `b` pendant les étapes `2`, `3`, `4` et `5` et qu'une liste contenant les symboles associés aux caractères `b` et `d` est empilée à l'étape `6`. Cette étape correspond à l'action `réduire` qui consiste à dépiler `X` éléments (`X` étant le nombre d'éléments de l'alternative, il vaut 4). On dépile donc les quatre éléments au sommet de la pile, à savoir les listes (`[TD]`, `[TC]`, `[TB]` et `[TA]`) et ensuite, on réempile une liste contenant le symbole `TB` et le symbole `TD`. Nous ne cherchons pas à montrer pour l'instant comment la liste `[TB TD]` a été construite à partir des quatre listes précédentes. Seul le résultat compte à ce niveau.

Un deuxième exemple qui permet de bien comprendre le comportement de la pile est présenté (Exemple 6.5).

Exemple 6.5

```
Tokens   a = 'a' ;
         b = 'b' ;
         c = 'c' ;
         d = 'd' ;
```

```
        z = 'z';
        t = 't';
Productions
  ...
  prod2          {→ b t } =
       t prod z  {→ prod.b t};
  prod           {→ b d } =
       a b c d   {→ b d};
```

Pour la phrase "`tabcdz$`" correspondant à une dérivation de la production `prod2` de l'exemple ci-dessus, la configuration de l'analyseur syntaxique correspondant donne :

```
-1- (0,                        TTTATBTCTDTZ$)
-2- (0[TT]1,                       TBTCTDTZ$)
-3- (0[TT]1[TA]2,                    TCTDTZ$)
-4- (0[TT]1[TA]2[TB]3,                 TDTZ$)
-5- (0[TT]1[TA]2[TB]3[TC]4,              TZ$)
-6- (0[TT]1[TA]2[TB]3[TC]4[TD]5,           $)
-7- (0[TT]1[TB TD]6,                       $)
-8- (0[TT]1[TB TD]6[TZ]7,                  $)
-9- (0[TB TT]8,                            $)
```

L'alternative qui est analysée ici est l'unique alternative de la production `prod2`. Elle est composée du symbole terminal `t`, suivi d'un symbole non terminal `prod`, et enfin d'un symbole terminal `z`. Les étapes 1 à 6 ont permis d'empiler les listes contenant les noeuds associés aux symboles `t`, `a`, `b`, `c`, `d`. Cela correspond à l'analyse du symbole terminal `t` et du symbole non terminal `prod`. Étant donné que `prod` est un symbole non terminal, il faut en faire l'analyse complète avant de passer au prochain symbole. C'est pour cela qu'à l'étape 7, le nombre d'éléments (4) de l'alternative de la production `prod` est dépilé et ensuite une liste (`[TB TD]`) contenant les éléments de la transformation de production de `prod` est empilée. Ceci marque la fin de l'analyse du symbole non

terminal `prod`. Ensuite se poursuit l'analyse de l'alternative de la production `prod2` qui permet d'empiler la liste contenant le noeud associé au symbole `z`. Une fois cette liste empilée, on est à la fin de l'alternative `prod2` ce qui déclenche une action *réduire*. La réduction effectuée dépile donc les 3 listes (l'alternative de la production prod comporte 3 éléments) au sommet de la pile et réempile une liste contenant deux noeuds associés aux symboles `b` et `t`. Il s'agit des symboles de la transformation de l'alternative qui est analysée. Et l'analyse se termine.

Pour résumer, le mécanisme général lors de l'analyse syntaxique d'une phrase ou proto-phrase[5] basée sur une alternative d'une production de la grammaire consiste :

- dans un premier temps à mettre dans une liste tous les noeuds associés aux symboles de cette alternative et à les empiler au fur et à mesure ;
- ensuite à dépiler X éléments (X étant le nombre d'éléments composant l'alternative) ;
- et enfin à réempiler le résultat de la transformation de cette alternative s'il y a lieu[6].

6.2.3 Détails de mise en oeuvre

Nous allons aborder dans ce paragraphe les différents types de transformations d'alternatives. Au regard de ce qui a été précédemment vu, la plus importante des transformations est la rétention d'un élément déjà existant dans la forêt d'arbre en construction. Nous allons toutefois parler de toutes les sortes de transformations d'alternatives.

---

[5]Se référer à la définition 2.4 dans le chapitre 2 pour une définition du terme proto-phrase.

[6]Si elle n'est pas la transformation "*vide*".

La rétention d'un élément déjà existant dans la forêt d'arbres de l'arbre en construction

Considérons l'exemple 6.5. L'analyse syntaxique de la phrase "`tabcdz$`" a permis de construire comme noeud d'arbre syntaxique, une liste contenant un noeud associé au symbole `b` et un noeud associé au symbole `t`. Nous allons maintenant voir comment cette liste a été construite.
L'analyse syntaxique a été faite en se basant sur l'alternative de la production `prod2`. Cette alternative est composée de trois symboles. Le premier, `t`, est un symbole terminal, le second, `prod`, est un symbole non terminal et le troisième, `z`, un autre symbole terminal. La phrase "`tabcdz$`" est dérivée à partir de ces trois symboles. En effet, le caractère `t` correspond au jeton `t`, le groupe de caractère "`abcd`" est dérivé du symbole `prod` et le caractère `z` correspond au jeton `z`. L'analyse syntaxique commence par le premier symbole de l'alternative c'est à dire `t`. Pour ce symbole, une action *décaler* est effectuée, ce qui occasionne l'empilement de la liste correspondant à ce dernier. Ensuite, l'analyse se poursuit avec le second symbole qui est `prod`. Ce symbole est non terminal, ce qui déclenche une analyse de l'alternative de cette production. Nous ne rentrerons pas dans les détails de cette analyse. La chose à retenir cependant est que pour cette alternative, une liste contenant les deux symboles `b` et `d` est créée et empilée. Les éléments contenus dans cette liste sont ceux spécifiés dans la transformation de l'alternative de la production `prod`. Et pour finir l'analyse, le symbole `z` est analysé et permet d'empiler la liste contenant le noeud associé à `z`. Après le symbole `z`, la fin de l'entrée est vue par l'analyseur, ce qui occasionne une action *réduire*. L'action *réduire* comme précédemment mentionné occasionne le dépilement de 3 listes (car l'alternative contient 3 symboles), effectue les transformations relatives spécifiées dans la transformation d'alternative et empile le résultat. La transformation d'alternative est : `{→ prod.b t}.prod` fait référence à la liste contenant les noeuds `b` et `d` (`[b d]`). Les éléments de cette liste sont accédés par des indices. Il faut maintenant trouver l'indice de l'élément `b` dans cette liste car la transformation de production indique `prod.b`. Pour ce faire, il suffit de regarder la transformation de production de `prod`, c'est-à-dire `{→ b d }`. Ceci nous indique la

première position, donc pour trouver le noeud correspondant à la transformation d'alternative `prod.b`, il suffit d'aller chercher le premier élément de la liste `[b d]`, ce qui donne `b`. Pour le deuxième élément `t` de la transformation d'alternative, la liste empilée contenant le noeud `t` correspond au symbole `t`. Il suffit donc d'aller chercher le noeud `t` en question dans la liste.

La création d'un nouveau noeud

Ce type de transformation se combine en général avec un autre type de transformation, le plus souvent le type *simple* (`New ast_prod.ast_alt(param`$_1$`, ..., param`$_N$`))`. En effet, ce type de transformation permet de créer un nouveau noeud et la création d'un nouveau noeud requiert des éléments devant composer ce noeud en question. C'est la raison pour laquelle ce type de transformation est combiné avec d'autres. Les actions à effectuer pour implanter ce type de transformation ne sont pas nombreuses. Il faut d'une part créer le noeud dont le type est `AAstAltAstProd` conformément à la transformation spécifiée (`New ast_prod.ast_alt( ... )`). Pour les paramètres de ce noeud, étant donné qu'il s'agit de transformations d'alternatives, ils sont automatiquement traités selon les transformations appropriées.

La création d'une liste de noeuds

Ce type de transformation est semblable à la transformation précédente. L'action effectuée se limite à la création d'une liste dans laquelle sont ajoutés les éléments spécifiés dans la transformation d'alternative. Ces éléments sont à leur tour aussi des transformations d'alternatives, ce qui fait que l'ajout se fait tout seul.

La transformation *Null*

Un noeud avec une référence vers l'objet `null` est créé.

La transformation *vide*

Absolument rien n'est fait dans le cas de ce type de transformation.

6.3 Résumé

Nous avons décrit dans ce chapitre le moteur de l'analyseur syntaxique de SableCC. Nous nous sommes basés sur ce moteur pour décrire comment étaient prises en charge les différentes sortes de transformations d'arbres syntaxiques supportées par SableCC3 à travers des exemples basés sur des extraits de grammaire. Nous avons évité de trop entrer dans les détails du code pour ne pas perdre le lecteur, mais pour ceux qui sont intéressés à adapter le code de génération de l'arbre syntaxique, ce chapitre pourrait constituer un bon départ pour la compréhension de la structure interne.

Chapitre VII

RÉSOLUTION AUTOMATIQUE DE CERTAINS CONFLITS LALR(1) PAR UNE MÉTHODE BASÉE SUR L'INCLUSION DE PRODUCTIONS

7.1 Introduction

Une utilisation immédiate et pratique des transformations d'arbres syntaxiques a été la résolution automatique de conflits par la méthode d'inclusion de productions. Nous allons maintenant, dans les quelques paragraphes qui suivent, décrire ce qu'est un conflit dans un analyseur LR (Aho, Sethi et Ullman, 2000), quels types de conflits existent, quel est le principe de fonctionnement de l'inclusion de productions, comment il fonctionne et finalement, nous allons aussi montrer des exemples de sa mise en oeuvre dans SableCC.

7.2 Définition d'un conflit et types de conflits

Comme nous l'avons déjà vu, le principe de fonctionnement d'un analyseur par décalage-réduction (incluant les analyseurs LALR) repose sur quatre types d'action (Aho, Sethi et Ullman, 2000) :

- *décaler* le symbole en entrée sur la pile d'analyse ;
- *réduire* par une production (plutôt une alternative) de la grammaire ;
- *accepter* et arrêter l'analyse ;
- *signaler une erreur* et arrêter l'analyse dans le cas de SableCC.

Les actions *accepter* et *erreur* quand elles surviennent, occasionnent l'arrêt de l'analyse du moins dans le cas de SableCC. Les actions *décaler* et *réduire* quant à elles, constituent les actions pouvant être effectuées plusieurs fois par l'analyseur lors d'un cycle d'analyse pouvant conduire à un succès ou un échec. Il faut aussi savoir qu'à chaque étape de l'analyse syntaxique, une seule action peut être effectuée à la fois. Un conflit survient donc si, à un moment donné, l'analyseur a le choix entre plusieurs actions, plus précisément entre une action *décaler* et une ou plusieurs actions *réduire* ou encore entre des actions *réduire*. On parle alors de conflit *décaler/réduire* dans le premier cas et de conflit *réduire/réduire* dans le second cas. Les conflits, s'ils existent, proviennent de la grammaire et sont détectés au moment de la génération de l'analyseur syntaxique. Nous allons montrer un exemple de chaque type de conflit et ensuite nous montrerons aussi comment nous parvenons à les résoudre.

7.2.1 Conflit décaler/réduire

Soit le fichier de spécification de SableCC suivant :

Spécification 7.1

```
Tokens
  a = 'a' ;
  b = 'b' ;
  c = 'c' ;
  d = 'd' ;
Productions
  prod =
         {premier}  a prod1 prod2 c  |
         {second}   a prod2 prod1 d  ;
  prod1 = b ;
  prod2 = b b ;
```

La grammaire (section `Productions`) de la spécification 7.1 contient un conflit car pour la chaîne d'entrée `abbbc$`[1], on ne peut pas déterminer comment analyser cette phrase, bien qu'il fasse partie du *langage engendré* (ensemble des phrases valides) par la grammaire (section `Productions`) de la spécification 7.1, c'est-à-dire `{abbbc, abbbd}`. La phrase "`abbbc`" est dérivable avec l'alternative `premier` de la production `prod` et "`abbbd`" est dérivable avec l'alternative `second` de cette même production. Regardons le comportement de l'analyseur pour comprendre pourquoi donc le conflit survient même si le mot fait partie du langage engendré par la grammaire. L'analyseur empile successivement les jetons `a` et `b` sans aucun problème. Après avoir empilé ces deux symboles, il se trouve dans la configuration suivante :

Pile d'analyse	**Entrée**
`0`	`abbc$`
`0a1`	`bbc$`
`0a1b2`	`bc$`

Au niveau de la grammaire, l'état d'avancement de l'analyseur syntaxique après la lecture du premier symbole `a` serait (*état 1*) :

```
prod =
        {premier}  a (*)[2] prod1 prod2 c |
        {second}   a (*) prod2 prod1 d ;
prod1 =  (*) b ;
prod2 =  (*) b b ;
```

Après le symbole `a`, le symbole de positionnement de l'analyseur syntaxique "`(*)`" se trouve aussi au début des alternatives des productions `prod1` et `prod2`. Ceci est dû au fait que le symbole de positionnement de l'analyseur syntaxique dans les alternatives

[1] Le caractère $ représente la fin de la phrase à analyser.

[2] Le symbole "`(*)`" représente l'endroit où l'analyseur syntaxique est rendu. Les symboles à droite qui suivent immédiatement `(*)` sont ceux qu'il est susceptible de voir en entrée.

`premier` et `second` de la production `prod` apparaît devant les productions `prod1` et `prod2`. En effet, étant donné que ce symbole détermine ce qui a déjà été analysé (les éléments à gauche du symbole) et ce qui reste à analyser (les éléments à droite du symbole) par l'analyseur syntaxique, la présence de ce symbole devant une production implique que l'analyseur s'attend à analyser cette production. Et étant donné qu'une production est constituée d'alternatives, cela revient donc à analyser les alternatives de la production en question. C'est la raison pour laquelle la présence du symbole de positionnement de l'analyseur syntaxique devant un symbole correspondant à une production dans une alternative quelconque occasionne automatiquement la présence de ce symbole au début des alternatives de cette production. Ce processus est répété jusqu'à ce que le symbole de positionnement se retrouve uniquement au début de symboles terminaux. Le prochain symbole pouvant être rencontré est le symbole `b`. Et après ce symbole, l'état d'avancement de l'analyseur syntaxique dans la grammaire est (*état 2*) :

```
prod =
        {premier} a (*) prod1 prod2 c |
        {second}  a (*) prod2 prod1 d ;
prod1 =  b (*) ; lookahead[3] b
prod2 =  b (*) b ;
```

Le prochain symbole en entrée est donc `b` et l'analyseur se trouve dans la configuration de l'état 2. En consultant les tables d'analyse en vue de trouver l'action d'analyse à effectuer, deux actions sont possibles. Soit réduire par l'alternative de la production `prod1 = b` ou encore décaler le `b` sur la pile (`prod2`). Dès lors que ces deux actions sont possibles, un conflit est généré.

Le problème dans ce cas-ci vient du fait que l'analyseur est *LALR(1)*, c'est-à-dire qu'il détermine la prochaine action à effectuer en regardant **un seul symbole de prévision**. Si on avait utilisé un analyseur *LALR(3)*, c'est à dire avec trois symboles de prévision, on se serait rendu compte que seule l'alternative *premier* aurait pu correspondre et on

[3]Lookahead : représente les jetons pouvant suivre une production dans un contexte (état) donné.

aurait ainsi pu éviter le conflit. Malheureusement, le problème avec cette approche réside dans le fait que les analyseurs *LALR(k)* (k>2) nécessitent un espace mémoire considérablement élevé et le temps d'analyse est exponentiel, ce qui les rend inutilisables dans les systèmes réels. La solution qui est introduite dans SableCC est celle de l'inclusion de productions. Elle est abordée à la section 7.3.

7.2.2 Conflit réduire/réduire

Nous allons illustrer ce conflit à l'aide un exemple. Soit le fichier de spécification de SableCC suivant :

Spécification 7.2

```
Tokens
  x = 'b';
  y = 'c';
  z = 'd';
  t = 't';
Productions
  p =
      {pa} a b z |
      {pb} b a t ;
  a = x y;
  b = x y;
```

Le conflit dans ce cas-ci est dû au fait que deux actions différentes peuvent être effectuées pour la chaîne d'entrée "`bcbcd$`". En effet, pour les deux premiers caractères `b` et `c`, l'analyseur syntaxique effectue des actions "*décaler*" mettant ainsi les symboles correspondant à ces caractères sur la pile d'analyse. L'état d'avancement de l'analyseur syntaxique au niveau de la grammaire donne :

```
p =
    {pa} a (*) b z |
```

```
    {pb} b (*) a t ;
a = x y (*) ; lookahead : x
b = x y (*) ; lookahead : x
```

Ensuite, en voyant le prochain caractère qui est *b*, deux actions peuvent être faites par l'analyseur syntaxique : soit réduire en utilisant la production `a = x y`, soit réduire en utilisant la production `b = x y`. Dès lors que plus d'une action est possible, il y a un conflit. Et il s'agit dans ce cas d'un conflit de type *réduire/réduire*.

7.3 Inclusion de productions

7.3.1 Principe de fonctionnement

Tel que mentionné précédemment, l'inclusion de productions a pour but la résolution de certains conflits dans la grammaire. Le processus d'inclusion de productions est itératif. Il fonctionne de la manière suivante :

1. tout d'abord, il faut trouver l'ensemble des productions impliquées dans le conflit. Il y a au moins une, autrement il n'y aurait pas de conflit. Soit *EnsProdConflits* cet ensemble ;
2. ensuite, il faut éliminer de *EnsProdConflits*, les productions récursives. La raison est simple : étant donné qu'on inclut ces productions dans toute la grammaire, une production récursive occasionnerait une inclusion à l'infini, ce qu'on veut évidemment éviter ;
3. si *EnsProdConflits* n'est pas vide, on inclut dans toute la grammaire les productions de cet ensemble et on supprime ces productions de la grammaire ;
4. on reprend ensuite la tentative de génération de l'analyseur syntaxique. Deux cas de figure peuvent se présenter à partir de ce moment :
   (a) soit le conflit est résolu et l'analyseur syntaxique est généré ;
   (b) soit un autre conflit apparaît et on répète le même processus (retourner à 1).

Preuve de fin d'itération

Dans le cas où l'inclusion des productions dans la grammaire ne permet pas de résoudre le conflit, le processus d'inclusion est automatiquement repris tel mentionné précédemment. Dans un tel cas, nous garantissons quand même qu'il s'arrête. Comment ?

- Soit toutes les productions impliquées dans le conflit sont récursives et ne peuvent donc être inclues dans la grammaire, auquel cas on arrête et on reporte le conflit à l'utilisateur.
- Soit on arrive dans le pire cas qui consiste à inclure toutes les productions sauf la première (car correspondant à l'axiome). Dans un tel cas de figure, on arrête la tentative de génération peu importe l'issue.

7.3.2 Exemple d'application

En appliquant le principe d'inclusion de productions aux spécifications 7.1 et 7.2, ceci implique dans le cas de la spécification 7.1 (conflit "*décaler/réduire*") d'inclure les productions `prod1` et `prod2` dans la production `prod` et dans le cas de la spécification 7.2 (conflit "*réduire/réduire*") d'inclure les productions `a` et `b` dans la production `p`.
Le résultat de ces inclusions nous donne les nouvelles productions `prod` et `p` suivantes :

Avant Inclusion	**Après Inclusion**

`prod = {premier} a prod1 prod2 c |`
`{second} a prod2 prod1 d ;`
`prod1 = b ;`
`prod2 = b b ;`
	`prod = {premier} a` $\underbrace{\texttt{b}}$ $\underbrace{\texttt{bb}}$ `c |`
	`{second} a` $\underbrace{\texttt{bb}}$ $\underbrace{\texttt{b}}$ `d ;`

`p = {pa} a b z |`
`{pb} b a t ;`
`a = xy ;`
`b = xy ;`
	`p = {pa}` $\underbrace{\texttt{xy}}$ $\underbrace{\texttt{xy}}$ `z |`
	`{pb}` $\underbrace{\texttt{xy}}$ $\underbrace{\texttt{xy}}$ `t ;`

L'avantage de cette nouvelle composition tient au fait que les productions sont à présent constituées en majorité de jetons et, étant donné que la seule action à effectuer en présence de jetons est une action *décaler*, les conflits éventuels sont de ce fait éliminés.

7.3.3 Incidence sur les transformations d'arbres syntaxiques

La résolution automatique de conflits par la méthode d'inclusion des productions a pu être réalisée de manière transparente pour l'usager grâce aux transformations d'arbres syntaxiques. En effet, étant donné que les arbres syntaxiques sont à présent construits en se basant sur les transformations spécifiées à l'intérieur de la section `Productions` du fichier de spécification, des ajustements peuvent être faits au niveau de ces transformations pour que l'arbre originellement spécifié par l'usager soit construit même s'il y avait inclusion de productions dans la grammaire. Prenons un exemple pour illustrer la chose.

Avant Inclusion
Productions
`prod =  {→ ast_prod }`
`  {premier} a prod1 prod2 c  {→ New ast_prod([prod1.b prod2.b])}  |`
`  {second} a prod2 prod1 d  {→ New ast_prod([prod2.b prod1.b])}  ;`
`prod1  {→ b }`
`  = b  {→ b };`
`prod2  {→ b* }`
`  = b b  {→ [b b]};`
`Abstract Syntax Tree`
`ast_prod = b+ ;`
Après Inclusion
`Productions`
`prod =  {→ ast_prod }`
`  {premier} a b b b c  {→ New ast_prod([b b b])}  |`
`  {second} a b b b d  {→ New ast_prod([b b b])}  ;`
`Abstract Syntax Tree`
`ast_prod = b+ ;`

L'arbre syntaxique qui résultera de l'inclusion de l'analyseur syntaxique généré pour la grammaire après inclusion est le même que celui de la grammaire avant inclusion des productions. Pour preuve, la section `Abstract Syntax Tree` qui sert à la construction de cet arbre n'a pas changé. Ce qui a changé, ce sont les transformations insérées dans la section `Productions` pour faire en sorte que l'arbre syntaxique résultant soit le même malgré l'inclusion des productions.

7.4 Résumé

Nous avons présenté dans ce chapitre le mécanisme de résolution automatique de conflits de SableCC basé sur l'inclusion de productions. Nous avons d'abord présenté les

différents types de conflits existant à l'aide d'exemples et ensuite, nous avons expliqué le principe de fonctionnement de ce mécanisme à l'aide de fichier de spécification de SableCC (plus précisément la section `Productions`) incluant des transformations d'arbres syntaxiques.

Chapitre VIII

TESTS DE BON FONCTIONNEMENT ET DE PERFORMANCE

8.1 Introduction

Dans ce chapitre, nous parlerons des tests que nous avons effectués dans l'environnement SableCC. Ces tests ont été de deux ordres. Nous avons testé la validité de transformations effectuées dans le logiciel mais aussi la pertinence de ces transformations en ce qui concerne les outils et logiciels développés avec SableCC. Les premiers tests ont permis de vérifier la robustesse des transformations d'arbres syntaxiques et nous ont permis de découvrir certains bogues qui ont été par la suite corrigés. La deuxième série de tests a consisté à tester la performance des logiciels développés avec SableCC. En effet, c'est avec ces tests que nous avons pu vérifier le gain en espace mémoire et en temps qu'ont permis de réaliser les logiciels développés avec SableCC et les transformations d'arbres syntaxiques.

8.2 Tests systèmes

Le souci, en développant ces tests, a été de couvrir de la manière la plus exhaustive possible toutes les transformations pouvant être réalisées. Dans cette optique, nous avons recensé les types de transformations pouvant être effectuées par SableCC. Il y en a cinq en tout. Notre première série de tests a donc tourné autour de ces cinq types de transformations. Nous avons procédé en écrivant des tests pour chacun des types de transformations. Ainsi nous avons produit quatre tests pour chacune des transforma-

tions `New`[1], `Null`[2], `id`[3] et `list`[4] et un test pour la transformation `vide`[5]. En prenant comme type de transformation le *New*, les quatre tests par transformation sont constitués comme suit :

1. `prod {→ ast_node1} =`
   `elem1 ... elemN`
   `{→ New ast_node1() };`
2. `prod {→ ast_node1 ast_node2} =`
   `elem1 ... elemN`
   `{→ New ast_node1() New ast_node2() };`
3. `prod {→ ast_node1 ast_node2 ast_node3} =`
   `elem1 ... elemN`
   `{→ New ast_node1() New ast_node2() New ast_node3() };`
4. `prod {→ ast_node1 ast_node2 ast_node3 ast_node4} =`
   `elem1 ... elemN`
   `{→ New ast_node1() New ast_node2()`
   `New ast_node3() New ast_node4()};`

Étant donné le caractère particulier de la transformation *vide* (`{-> }`), il n'est pas possible d'effectuer les même transformations pour ce type de transformation que les quatre autres. En effet, une transformation vide indique qu'aucun noeud ne doit être construit pour l'arbre syntaxique pour ce non terminal, il n'est donc pas utile et même possible de l'indiquer plusieurs fois pour la transformation de production et d'alternative. Le seul test effectué dans ce cas est donc :

[1]Création d'un nouveau noeud de l'arbre syntaxique.

[2]Joker de remplacement d'un noeud en cas de pénurie.

[3]Rétention d'un noeud existant dans la forêt d'arbres de l'arbre en construction.

[4]Création d'une liste homogène de noeuds ou de feuilles.

[5]Élimination d'un sous arbre complet

```
prod {→ } = elem1 ... elemN {→ }
```

Pour les autres types de transformation, nous avons choisi de nous arrêter à quatre répétitions car nous estimons qu'il s'agit d'une limite raisonnable. Nous ne pourrons jamais tout tester dans tous les cas.

En plus de tests individuels pour les différents types de transformation, nous avons aussi fait des tests en les combinant. Nous avons effectué deux types de combinaisons, les combinaisons par composition et les combinaisons simples à savoir :

- la combinaison par composition est intrinsèque au type de transformation. En effet, si on prend par exemple la transformation de type `New`, elle se combine presque toujours avec une autre transformation (`id`, `liste`, `Null` ou encore `New`) car la création d'un nouveau noeud nécessite toujours des enfants qui correspondent à d'autres transformations.

  Exemple 8.1

  ```
  prod    {→ ast_node1} =
      elem1 elem2 elem3 elem4 elem5
          {→ New ast_node1(elem1, [elem2 elem3], Null,
                                New ast_node2(elem5) )
          } ;
  ```

- la combinaison simple consiste tout simplement à mettre côte à côte différentes sortes de transformations.

  Exemple 8.2

  ```
  prod    {→ ast_node2 token1 } =
      elem1 elem2 elem3 elem4 elem5
          {→ New ast_node2(elem5) elem2 } ;
  ```

  On peut aussi ajouter qu'une combinaison simple requiert que la transformation de production correspondante soit composée de plusieurs éléments alors que pour une combinaison par composition, il y a un seul élément.

Pour la réalisation des tests de combinaison, nous avons procédé en les mixant deux par deux, ensuite trois par trois et enfin, nous avons élaboré des tests qui combinent les quatre sortes de transformations lorsque cela était possible. Pour le nombre de tests de combinaison, nous nous sommes encore une fois limités à quatre pour les raisons énoncées précédemment.

Une autre partie des tests réalisés concerne la compatibilité avec les versions antérieures de SableCC, c'est-à-dire les versions dont le fichier de spécification ne supporte pas les transformations d'arbres syntaxiques. Pour les supporter, le nouveau logiciel SableCC insère des spécifications de transformation d'arbres syntaxiques au niveau de la grammaire. Les spécifications de transformations d'arbres syntaxiques insérées par SableCC en présence des opérateurs (`?`, `*` ou `+`) exigent un travail supplémentaire au niveau de la grammaire. Nous avons donc développé des tests pour nous assurer que ces opérateurs sont bien pris en compte. Ces tests ont été élaborés en suivant le même principe que les tests précédents. Nous avons donc développé un certain nombre de tests pour chacun des opérateurs `?`, `*` et `+` individuellement et ensuite, pour des combinaisons de ces opérateurs.

8.3 Tests de performance

Dans cette section, nous allons décrire trois outils qui ont initialement été développés avec une version antérieure de SableCC n'incluant pas les transformations d'arbres syntaxiques. Ces outils ont par la suite été réécrits avec la nouvelle version incluant le module de transformation des arbres syntaxiques que nous avons développé.

8.3.1 SableCC3.0

Certaines des composantes du logiciel SableCC3-beta1 ont été générées en utilisant SableCC2. Ainsi, l'analyseur lexical, l'analyseur syntaxique, les classes de l'arbre syntaxique et son générateur sont générés par SableCC lui-même à partir du fichier de spécification. Une fois cette première version fonctionnelle obtenue, nous avons décidé

de produire une nouvelle version du logiciel en profitant des transformations d'arbres syntaxiques. Nous avons, pour ce faire, fait des changements au fichier de spécification de SableCC pour y insérer des transformations d'arbres syntaxiques que nous avons ensuite donné à SableCC3-beta2 pour générer un nouveau cadre de travail. Nous avons ensuite réécrit les parties du code basées sur les classes de l'ancien cadre de travail. Nous nous sommes pour cela beaucoup servis du compilateur Java pour détecter les incompatibilités entre le nouveau cadre de travail et l'ancien code qui y faisait appel. Après détection de ces incompatibilités, il a fallu modifier ce code pour le rendre compatible au nouveau cadre de travail. La version de SableCC résultant, nommée SableCC3-beta3 fonctionne et rend le code plus simple à certains endroits.

L'objectif visé lors de la réécriture du logiciel SableCC en y intégrant les transformations d'arbres syntaxiques était de montrer, comme pour toutes les autres versions de SableCC, que le logiciel pouvait s'autogénérer mais aussi fournir une version de ce logiciel qui tire profit des fonctionnalités proposées. Dans cette optique, des gains de performance n'ont pas nécessairement été observés.

8.3.2 Expressions arithmétiques

Nous avons réécrit un calculateur d'expressions arithmétiques simples qui utilise les opérateurs arithmétiques `+`, `-`,`*` et `/` en insérant dans la grammaire de ces expressions des transformations d'arbres syntaxiques. Le résultat nous a permis de passer d'une grammaire avec trois productions et huit alternatives à une grammaire avec une unique production et cinq alternatives. De plus, les arbres syntaxiques construits pour les expressions arithmétiques du calculateur résultant sont beaucoup plus petits en termes d'espace mémoire car moins de noeuds sont nécessaires pour représenter les mêmes expressions. Ainsi, la représentation de l'arbre syntaxique pour l'expression "45 + 189 - 9 * 3 + 67 -102" par exemple, nécessite 9 noeuds avec la première version du calculateur et 5 noeuds avec la seconde version. Cette différence est considérable si on tient compte du fait qu'il s'agit d'un simple calculateur. Dans le cas de calculateurs plus complexes ou des expressions dans de langages de programmation d'envergure comme Java, la diffé-

rence pour la représentation de l'arbre syntaxique peut se compter en terme de dizaines voire même de centaines de noeuds. Le fichier de spécification complet des expressions arithmétiques se trouve en annexe B.

8.3.3 JJOOS

JJOOS est un compilateur développé comme projet de session pour un cours avancé sur les techniques de compilation par Othman Alaoui à l'université *McGill* (Alaoui, 2000). Ce compilateur utilise un langage qui est un sous-ensemble du langage de programmation Java. La version du compilateur développée par Othman utilise une version de SableCC qui n'inclut pas les transformations d'arbres syntaxiques car la version intégrant cette fonctionnalité n'était pas encore développée. Toutefois, des transformations ont été faites sur l'arbre syntaxique à l'intérieur du compilateur grâce à du code écrit manuellement. Nous avons modifié le compilateur JJOOS en nous servant de SableCC3-beta3. Le travail effectué pour cette réécriture a consisté dans un premier temps à insérer des transformations d'arbres syntaxiques dans la grammaire du langage supporté par JJOOS et, dans un second temps, à éliminer l'ancien code des transformations écrit à la main. L'objectif a été atteint à 90%. En effet, presque toutes les transformations d'arbres syntaxiques effectuées manuellement par du code ont pu être remplacées par les transformations automatiques de SableCC. Cela nous a permis de supprimer près de 400 lignes de code dans deux classes différentes qui étaient initialement destinées à effectuer les transformations au niveau de l'arbre syntaxique. Nous avons effectué des tests de performance pour deux versions de l'analyseur syntaxique du compilateur JJOOS (avec et sans simplification de l'arbre syntaxique). D'une part, en utilisant le même fichier d'entrée d'une taille de 6648126 octets, nous avons obtenu dans le premier cas un temps d'analyse syntaxique de 127 minutes et 27 secondes alors que dans le second cas, le temps est passé à 116 minutes et 41 secondes. Il s'agit là d'une amélioration du temps d'exécution d'environ 10%. D'autre part, des informations de sortie obtenues avec l'option `-verbose :gc` passée à la machine virtuelle *HotSpot* de *Sun Microsystems* nous ont permis de constater, dans le cas de l'analyseur syntaxique

sans simplification de l'arbre syntaxique, que la sollicitation du ramasse miettes (chargée de la libération d'espace mémoire lorsque nécessaire) est beaucoup plus importante que dans le cas de l'analyseur syntaxique avec simplification de l'arbre.

Banc d'essai	**Sans transformations d'arbres syntaxiques**	**Avec transformations d'arbres syntaxiques**
Expression arithmétique	127mn27s	116m41s
JJOOS	6mn56s	6mn48s

Tableau 8.1 Comparaison des temps d'exécution entre bancs d'essai avec et sans transformation d'arbres syntaxiques

8.4 Résumé

Nous avons présenté dans ce chapitre les tests, une des composantes importantes du processus de validation d'un logiciel. Nos tests ont consisté à vérifier la fonctionnalité des transformations d'arbres syntaxiques mises en place, mais aussi la pertinence de ces transformations à travers des mesures de performance effectuées sur deux versions (sans et avec transformations d'arbres syntaxiques) d'outils développés avec SableCC.

Chapitre IX

TRAVAUX RELIÉS

9.1 Introduction

Parmi la grande variété d'environnements de développements de compilateurs existants, un certain nombre tentent de s'attaquer au problème de transformation de l'arbre syntaxique. Diverses approches sont proposées par les outils existants. Nous allons dans ce chapitre présenter certains de ces outils et le type de support proposé pour les transformations d'arbres syntaxiques. Nous mettrons en évidence le contraste entre l'approche adoptée par SableCC et celle utilisée par ces outils le cas échéant. Nous conclurons ensuite en faisant ressortir les différences fondamentales entre ces outils.

9.2 Txl (Turing eXtender Language)

Txl est à la fois un langage de programmation et un outil qui sert à décrire les constructions de langages de programmation. Il s'agit d'un langage complet qui permet de développer les applications de type traduction source à source. Il a été initialement conçu comme étant un langage capable de prototyper rapidement de nouveaux types de langages de programmation, soit en concevant de nouveaux, soit par l'extension de langages déjà existants (Cordy, Halpem et Promislow, 1988). Txl se base sur la détection de patrons (*pattern-matching*) pour effectuer les transformations à l'intérieur des programmes. Un programme Txl est constitué de deux parties. La première correspond à une description de la structure des programmes à transformer. Il s'agit en fait d'une

grammaire non contextuelle exprimée avec la notation *BNF*. La deuxième partie est constituée des règles qui vont servir à effectuer les transformations dans les textes dont la structure a été spécifiée par la grammaire contenue dans la première partie. L'application de ces règles est faite en se basant sur le *pattern-matching* qui consiste à rechercher dans les textes conformes à la grammaire une séquence de mots correspondant à un patron et à appliquer les règles de transformations pour ce patron. Le résultat est une nouvelle structure manipulable uniquement par un programme Txl autrement dit par d'autres règles écrites avec le langage Txl. En utilisant Txl, il est possible d'arriver au même résultat que les transformations d'arbres syntaxiques offertes par SableCC. Cependant les deux approches diffèrent complètement. Txl se base sur le pattern matching qui suppose que les transformations soient appliquées après que l'analyse syntaxique ait été faite alors que SableCC utilise une approche déterministe qui intervient au moment de l'analyse syntaxique et permet ainsi d'appliquer exactement la transformation souhaitée par l'utilisateur. Un autre facteur de divergence entre ces deux outils réside dans le fait que les utilisateurs de Txl doivent se conformer uniquement à sa syntaxe pour tout le processus de développement du compilateur alors qu'avec SableCC, seule la transformation de l'arbre syntaxique utilise la syntaxe particulière de SableCC qui, au demeurant, est très simple. Le reste du processus peut être traité par des langages de programmation conventionnels supportés par SableCC comme *Java*, *C++* ou *C#*. Aussi, le fait de faire reposer les transformations sur le *pattern-matching* oblige l'utilisateur (Cordy, 2004) à contrôler les conditions d'arrêt de l'application des règles sur les patrons. L'ordre d'application des patrons peut être contrôlé par l'utilisateur mais le résultat d'application d'un patron peut invalider l'application d'un autre, ce qui peut être problématique. Voici maintenant certains des avantages et inconvénients de l'utilisation de l'outil Txl.

Avantages

- Prototypage rapide de langages de programmation.
- Possibilité de redéfinir les symboles non terminaux de la grammaire grâce à un

mécanisme d'héritage.

- Vérification sémantique du type des transformations effectuées.
- Exécution des opérations subséquentes aux transformations comme le calcul d'opérations arithmétiques par exemple. Le fait que Txl soit un langage de programmation complet facilite grandement cet aspect.
- Facilité pour les traductions entre langages.

Inconvénients

- Apprentissage d'un nouveau langage de la part de l'utilisateur.
- Intervention de l'utilisateur requise pour la gestion de la condition d'arrêt d'application d'une règle sur un patron. Ceci peut s'avérer dangereux dans le cas d'utilisateurs non avertis, par exemple si des patrons se chevauchent. Aussi, la mauvaise spécification de patrons peut entraîner l'outil dans une boucle infinie. En effet, supposons qu'une transformation demande le remplacement d'un patron "`a`" par "`aa`" et une autre qui spécifie le remplacement de "`aa`" par "`a`". Dans de telles conditions, l'outil peut effectuer des remplacements successifs qui n'aboutiront jamais.
- Syntaxe abstraite cachée par la spécification des transformations. Il faut, en effet, la déduire à partir de la spécification des transformations faites sur la syntaxe concrète.
- Obligation de travailler avec Txl de bout en bout. Il n'est pas possible d'obtenir l'arbre syntaxique transformé et d'y effectuer le reste des opérations dans un langage de programmation conventionnel avec toute la puissance que ce langage peut offrir (structure de données disponibles, algorithmes pré-implantés ou pouvant être implantés).

9.3 ANTLR (ANother Tool for Language Recognition)

ANTLR est un environnement de développement de compilateurs et de traducteurs qui génère des analyseurs lexicaux et des analyseurs syntaxiques descendants.

ANTLR a été principalement développé par Terrence Parr. ANTLR peut être classé dans la catégorie des environnements bout en bout, c'est-à-dire un environnement qui permet de développer le compilateur au complet en se servant uniquement de l'outil. Toutefois, il est possible avec ANTLR de laisser à l'utilisateur la possibilité de compléter le code généré grâce à du code que ce dernier a écrit. Ce code peut être écrit avec les langages de programmation *C++*, *Java*, *C#* et *Python*. ANTLR supporte la construction d'arbres syntaxiques abstraits. Pour ce faire, l'outil construit un arbre syntaxique abstrait homogène en indiquant pour chacune des productions de la grammaire lequel des symboles la constituant sera la racine du sous-arbre à construire pour cette production. Les autres symboles sont automatiquement considérés comme des fils de la racine. Aussi, les noeuds de l'arbre syntaxique qui sont des parents (racine de sous-arbres ou racine de l'arbre) doivent absolument correspondre à un symbole terminal. Il s'agit là d'une contrainte que ANTLR impose car les arbres syntaxiques qu'il génère ne sont pas typés. En effet, tous les noeuds des arbres syntaxiques générés par ANTLR ont un seul et même type. Ceci rend le parcours de l'arbre avec un visiteur assez pénible et enclin à toute forme d'erreurs. Toutefois, ANTLR offre la possibilité d'utiliser une grammaire d'arbres. Il s'agit d'une grammaire dont le but est d'offrir un moyen de parcours de l'arbre syntaxique abstrait précédemment construit afin d'y effectuer des opérations. Ainsi, à l'image de la grammaire du langage qui permet de parcourir les programmes, il est possible d'enchâsser du code à l'intérieur de cette grammaire. Ce code sera exécuté lors du parcours de l'arbre quand le sous arbre correspondant sera reconnu.

La construction de l'arbre syntaxique abstrait doit se faire par l'intermédiaire d'annotations dans la grammaire. Grâce à ces annotations, l'utilisateur indique à ANTLR les jetons à inclure ou non dans l'arbre, lesquels de ces jetons correspondent à des racines de sous arbres et ceux qui seront fils de ces racines. En fait, l'utilisateur spécifie de cette manière la structure de l'arbre. L'exemple suivant est un extrait d'une grammaire ANTLR qui intègre des annotations de constructions d'arbres syntaxiques.

```
expr
        : mexpr (PLUS^  mexpr)* SEMI!   ;
```

```
mexpr
        : atom (STAR^  atom)*  ;
atom
        : INT  ;
```

Les symboles en majuscules sont terminaux alors que les autres symboles sont non terminaux. Le caractère "^" sert à indiquer à ANTLR que le symbole qui précède doit être considéré comme la racine du sous arbre construit pour cette alternative. Le caractère " !" permet d'exclure un symbole de l'arbre. Ainsi, la transformation ci-dessus construit l'arbre de la figure 9.1 pour la phrase d'entrée "34+25*12". La structure de l'arbre résultant de la transformation ci-dessus n'est pas très évidente à voir malgré le fait qu'il s'agit d'une transformation relativement simple. Comme on peut l'imaginer dans le cas de transformations complexes pouvant intervenir dans le cas de grammaires de langages de programmation comme *Java* ou *C++*, la structure de l'arbre syntaxique correspondant aux programmes de ces langages d'envergure serait très difficile à visualiser dans un cas de maintenance voire même pendant le développement. Aussi, il est facile de changer la structure d'un tel arbre étant donné qu'il n'est pas strictement typé. L'approche adoptée par ANTLR pour les transformations d'arbres syntaxiques s'inscrit cependant assez bien dans la philosophie de l'outil qui propose aussi de travailler avec ses arbres par l'intermédiaire de grammaires d'arbre (*tree grammar*). En effet, avec les grammaires d'arbres, il est facile de travailler avec les arbres syntaxiques construits car ces grammaires permettent de spécifier directement à l'intérieur de la grammaire les actions sémantiques subséquentes à effectuer. Par contre, nous n'adhérons pas tout à fait au principe des actions (code à exécuter en des noeuds donnés de l'arbre syntaxique) à l'intérieur de la grammaire car ce principe implique l'invocation de l'outil de génération du compilateur (ANTLR) à chaque fois que la moindre action devra être changée ou ajoutée pour effectuer une tâche quelconque. D'autres inconvénients existent quant à l'utilisation des transformations d'arbres syntaxiques de ANTLR :

– non représentation de la syntaxe abstraite. En effet, la structure de l'arbre

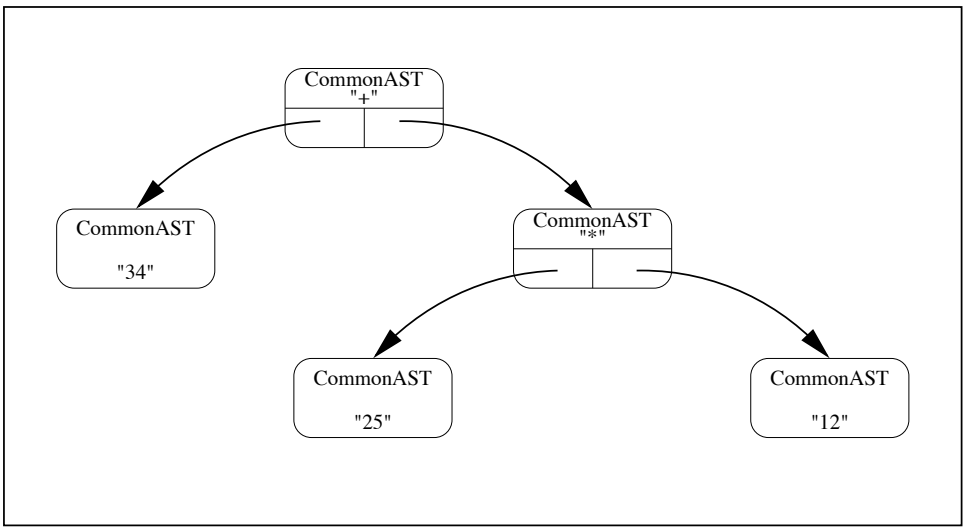

Figure 9.1 Arbre syntaxique abstrait de l'expression "34 + 25 * 12"

syntaxique abstrait est induite par les transformations insérées dans la syntaxe concrète. Rien ne nous permet de connaître explicitement la structure de l'arbre abstrait ;

- obligation suivant laquelle la racine de l'arbre ou des sous arbres de l'arbre doit absolument être un jeton (symbole terminal)
- identité de type de tous les noeuds de l'arbre syntaxique. Ceci n'est pas très adapté pour le parcours de l'arbre en dehors de la grammaire ;
- corruption facile de l'arbre du fait du typage non strict.

9.4 Gentle sur Lex et Yacc

Gentle est un "*term rewriting system*" qui a été initialement développé en 1989 dans les laboratoires de GMD Karlsruhe. À ce jour, c'est devenu un produit commercial. Gentle utilise les outils *Lex* et *Yacc* respectivement comme analyseur lexical et analyseur syntaxique. Avec cette configuration, Gentle peut donc être considéré comme un environnement de développement de compilateur complet. Dans ce sens, il est possible de définir une spécification lexicale et grammaticale pour un langage de programmation et ce dans le même fichier. Cette spécification correspond à la syntaxe concrète

du langage. À partir de cette spécification, Gentle crée une spécification lexicale et une spécification grammaticale qui sont traitées par *Lex* et *Yacc*. C'est donc en quelque sorte un préprocesseur pour ces outils. Cependant, Gentle offre la possibilité de définir une syntaxe abstraite de manière assez élégante. Cette syntaxe abstraite a pour but de simplifier la syntaxe concrète et de permettre de définir des actions pouvant y être effectuées de manière plus élégante que l'équivalent du code C qui aurait été nécessaire pour effectuer la même chose avec une grammaire Yacc. Un exemple d'action pouvant être définie dans la syntaxe abstraite est l'évaluation d'une expression arithmétique. Les possibilités offertes par Gentle en ce qui concerne la transformation de la syntaxe concrète vers la syntaxe abstraite sont comparables aux transformations d'arbres syntaxiques de SableCC. Cependant des différences existent entre les deux outils. D'une part, Gentle supporte uniquement deux types de transformation (création d'un nouveau noeud et rétention d'un noeud déjà existant) alors que SableCC en supporte trois de plus (la création de liste, la transformation *vide* et la transformation *Null*). D'autre part, avec SableCC, une production (les symboles non terminaux) peut être transformée en plusieurs autres symboles, ce qui permet à une étape ultérieure de la transformation de ne retenir que les symboles qui sont pertinents pour l'arbre syntaxique abstrait ; cela évite par ailleurs de créer un niveau de profondeur de plus dans l'arbre syntaxique. Pour finir, avec SableCC, le résultat de la transformation de la syntaxe concrète vers la syntaxe abstraite est un arbre syntaxique qui peut être manipulé dans un langage de programmation conventionnel (*Java*, *C++*, *C#* et d'autres à venir) alors qu'avec Gentle, la syntaxe abstraite n'est qu'un raccourci pour alléger le code encombrant qui serait enchâssé dans une grammaire *YaCC*.

9.5 JJTree et JavaCC

JavaCC est un générateur d'analyseurs syntaxiques descendants développé et maintenu dans les laboratoires *Sun MicroSystems*. *JavaCC* supporte les grammaires attribuées, autrement dit, du code à exécuter par l'analyseur syntaxique qui doit être généré pour cette grammaire peut y être enchâssé pour effectuer diverses tâches dont

la création d'arbres syntaxiques par exemple. *JavaCC* par défaut ne supporte pas la création d'arbres syntaxiques. L'outil *JJTree* s'occupe de la génération automatique d'arbres syntaxiques avec *JavaCC*. C'est un préprocesseur pour *JavaCC*. Il fonctionne de la manière suivante : *JJTree* prend en entrée une grammaire *JavaCC* augmentée avec une syntaxe particulière qui sert principalement à créer l'arbre syntaxique. Cette grammaire décorée est ensuite transformée par *JJTree* en une grammaire incluant des actions de constructions d'arbres syntaxiques pour *JavaCC*. Nous allons maintenant présenter les différents avantages et inconvénients de *JJTree* pour la transformation d'arbres syntaxiques.

Avantages

- Possibilité que l'arbre soit partiellement typé. En effet, un mode de transformation de l'arbre permet de créer un type différent pour toutes les productions de la grammaire.
- Possibilité d'éliminer un niveau de profondeur dans l'arbre à chaque production.
- Mode de transformation de l'arbre assez développé pour des utilisateurs avertis. En effet, on peut décider à un certain niveau de la transformation de l'arbre d'ignorer la construction d'un noeud parent et laisser ainsi les noeuds fils sur la pile pour qu'ils soient récupérés plus tard pour construire un autre noeud parent. Prenons un exemple pour clarifier les choses.

```
Start = Exp ;
Exp = AddExp
AddExp = MultExp ('+' | '-') MultExp ;
MultExp = UnaryExp ('*' | '/') UnaryExp ;
UnaryExp = INTEGER |
           IDENTIFIER ;
```

  On peut par exemple au niveau des productions `Exp`, `MultExp` et `UnaryExp` décider de ne pas construire de noeuds pour l'arbre syntaxique et d'en construire seulement pour les productions `Start` et `UnaryExp`. Ceci fait que pour la simple

expression arithmétique “`5+6`”, au lieu d’avoir un arbre syntaxique aussi profond que celui de la colonne gauche de la figure 9.2, on aurait l’arbre réduit de la colonne droite de la figure 9.2.

– Génération d’un visiteur minimaliste mais tout de même présent et donc disponible pour effectuer des premiers tests.

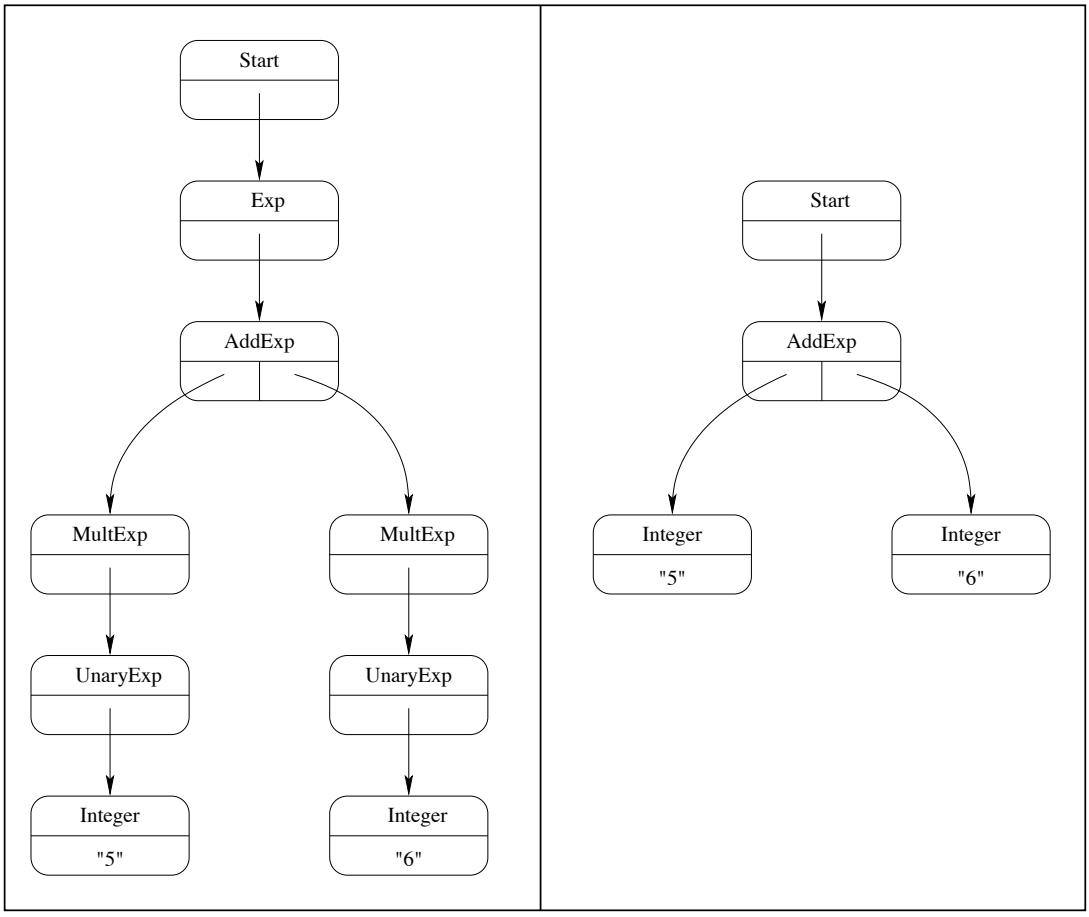

Figure 9.2 Arbre syntaxique concret et abstrait de l’expression “`5 + 6`” par *JJTree*

Inconvénients

– L’arbre syntaxique généré n’est pas strictement typé, ce qui implique une corruption facile de l’arbre. En effet, *JJTree* offre la possibilité de construire pour chacune des productions de la grammaire une classe différente pour représenter

les noeuds de l'arbre syntaxique qui seront construits à partir de cette production. Cependant, la structure de cette classe ne permet pas de contrôler les différents noeuds fils que pourrait avoir ce noeud (un parent peut avoir n'importe quel noeud comme fils).

- Il n'est pas possible de créer des listes de noeuds comme avec SableCC. En effet, SableCC permet de prendre des éléments (symboles) différents d'une même alternative et de les combiner ensemble dans une même liste homogène. La seule contrainte à respecter concerne le type des éléments de la liste qui doivent tous être identiques.
- On ne peut pas ne réutiliser que des parties de la transformation d'une production comme avec SableCC. Avec *JJTree*, le fait pour certaines productions de ne pas construire le noeud de l'arbre syntaxique correspondant est à peu près équivalent au mode de transformation multiple de SableCC. Cependant, avec SableCC, les symboles conservés pour la production peuvent être choisis par l'utilisateur alors qu'avec *JJTree*, ce sont automatiquement tous les symboles d'une alternative de la production.

9.6 JTB et JavaCC

Comme *JJTree*, *JTB* (*Java Tree Builder*) est un préprocesseur pour *JavaCC*. Il permet d'y inclure des actions destinées à la création d'un arbre syntaxique concret dans une grammaire *JavaCC* pour les programmes du langage décrit par cette grammaire. Il n'y a aucune notion de simplification ou transformation. Il s'agit tout simplement de la construction d'un arbre syntaxique concret.

9.7 Résumé

Ce chapitre a servi à faire une revue des différents outils et techniques disponibles pour les transformations d'arbres syntaxiques dans les environnements de développement de compilateurs. Nous avons identifié deux grandes techniques, à savoir les transformations basées sur la reconnaissance de patrons et celles basées sur la transformation

déterministe de l'arbre qui est, en général, faite pendant l'analyse syntaxique. Les différents outils utilisant l'une ou l'autre de ces techniques tentent à leur manière d'apporter des solutions au problème des transformations d'arbres syntaxiques en s'attaquant à un aspect particulier du problème ce qui laisse souvent d'autres failles ailleurs. Avec les transformations d'arbres syntaxiques de SableCC, nous avons essayé de proposer une approche qui se veut complète en terme des transformations qui sont faites sur l'arbre syntaxique, qui est simple d'utilisation et qui permet de garantir la robustesse de l'arbre syntaxique construit.

Chapitre X

CONCLUSION

10.1 Synthèse

L'utilisation des arbres syntaxiques est fréquent et est devenue presque incontournable lors des différentes phases de la compilation et de l'interprétation. Cependant, l'arbre syntaxique *concret* tel que construit au terme de l'étape d'analyse syntaxique ne correspond pas nécessairement aux besoins des étapes ultérieures de la compilation ou de l'interprétation qui nécessite un arbre syntaxique abstrait.

Dans ce mémoire, nous avons présenté une nouvelle approche de transformation des arbres syntaxiques concrets en arbres syntaxiques abstraits. Notre approche se base sur une spécification des transformations à l'intérieur de la grammaire à l'aide de cinq primitives de transformations que nous avons introduites. Cette approche a aussi le mérite de séparer la grammaire d'analyse syntaxique de la grammaire d'arbres syntaxiques permettant ainsi pour les étapes ultérieures du développement du compilateur ou de l'interpréteur de se focaliser uniquement sur la grammaire d'arbres syntaxiques.

Nous avons montré dans ce mémoire comment faire pour utiliser les transformations d'arbre syntaxiques. Nous avons notamment présenté la syntaxe de spécification de la grammaire d'arbres syntaxiques ainsi que celle de l'analyse syntaxique. Nous avons ensuite présenté comment se fait l'intégration de la spécification des transformations d'arbres syntaxiques à l'intérieur de la grammaire d'analyse syntaxique. Par la suite, nous avons présenté les vérifications sémantiques effectuées comme la vérification de la

concordance de types sur les transformations spécifiées et montré comment ces vérifications sémantiques permettaient de garantir l'exactitude des transformations spécifiées par l'usager en leur conférant un caractère déterministe. Nous avons ensuite montré comment le logiciel SableCC transforme la spécification *EBNF* fournie par l'usager en spécification *BNF* en vue de la faire accepter par l'algorithme du moteur d'analyse syntaxique qui ne supporte que la syntaxe *BNF*. Cette transformation automatique permet ainsi aux usagers d'écrire des grammaires plus concises. Par la suite, nous avons présenté les détails d'implantation de l'algorithme de construction de l'arbre syntaxique simplifié fournissant ainsi aux usagers désireux de transformer l'outil une documentation de base. Nous avons aussi montré comment, une fois le nouveau cadre de travail mis en place, nous avons pu facilement y intégrer un mécanisme de résolution automatique de conflits basé sur l'inclusion de productions. Ce mécanisme a l'avantage d'être transparent pour l'usager et permet également de ne pas changer le langage engendré par la grammaire, contrairement à d'autres mécanismes de résolution de conflits comme la résolution du conflit *décaler/réduire* au profit de l'action *décaler* de l'outil `YACC` (Levine, Mason et Brown, 1992). Pour finir, nous avons présenté les quelques tests effectués à la fois sur l'outil lui-même afin de tester sa robustesse et détecter d'éventuels bogues mais aussi pour mesurer l'apport de cette nouvelle fonctionnalité à travers des mesures de performance sur les outils développés avec SableCC.

10.2 Perspectives futures

L'objectif de ce travail a été atteint d'une manière générale. Plusieurs tests ont été effectués et ces tests montrent que le résultat obtenu est à la hauteur des attentes. Toutefois dans le souci d'améliorer encore plus l'utilisabilité du logiciel, nous pensons qu'il serait judicieux de faire un certain nombre de modifications à savoir :

- redéfinir la syntaxe du fichier de spécification des transformations dans le souci de dissocier visuellement la grammaire d'analyse syntaxique des transformations d'arbres syntaxiques ;
- fournir une documentation détaillée avec plusieurs exemples d'utilisation dans

des contextes de développement de vrais outils dérivés (les expressions arithmétiques dans les langages de programmation habituels par exemple, etc.) ;

– intégrer un outil de déboguage muni d'une interface graphique au logiciel afin de mieux aider les usagers à cerner les problèmes surtout en présence de conflits.

Aussi, nous recevons régulièrement des commentaires d'usagers de SableCC à propos de diverses fontionnalités qu'ils auraient aimé voir disponible dans le logiciel. Nous nous effor cons dans la mesure du possible de satisfaire à leur demande à partir du moment où ces demandes respectent la philosophie du logiciel.

Le logiciel SableCC ainsi qu'un ensemble de fichiers de spécification sont disponibles gratuitement en ligne à l'adresse suivante : `http://www.sablecc.org`.

Annexe A

FICHIER DE SPÉCIFICATION DE SABLECC3.0

Nous présentons dans cet annexe le fichier de spécification de SableCC3.0. Tel que mentionné précédemment dans ce mémoire, il s'agit de la version de SableCC intégrant les transformations d'arbres syntaxiques. Nous avons créé ce fichier de spécification en utilisant comme base le fichier de spécification des versions antérieures de SableCC qui n'intègrent pas les transformations d'arbres syntaxiques. Notre travail ici, a principalement consisté, dans un premier temps, à définir la syntaxe du langage reconnu par le logiciel SableCC3 en écrivant des exemples concrets et, dans un second temps, à partir de ces exemples déduire le fichier de spécification qui couvre ce langage. Cela nous a permis d'obtenir jusque là, une première version du fichier de spécification du logiciel SableCC3.0 à partir de laquelle nous avons développé le logiciel SableCC3.0. Ce fichier contenait uniquement les sections `Package`, `Helpers`, `States`, `Tokens`, `Ignored Tokens` et `Productions`.

Une fois que nous avons obtenu une première version fonctionnelle du logiciel SableCC3, nous avons décidé de mettre à profit du logiciel lui-même les transformations qui y ont été implantées. Nous avons ainsi ajouté au fichier de spécification obtenu jusque là des transformations d'arbres syntaxiques. Concrètement, nous y avons ajouté une section `Abstract Syntax Tree` et nous avons modifié la section `Productions` existante pour y intégrer des transformations d'arbres syntaxiques. Le résultat obtenu est donc un logiciel qui peut complètement s'auto-générer. La suite de ce document contient le code complet du fichier de spécification finalement obtenu.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * This file is part of SableCC.                              *
 * See the file "LICENSE" for copyright information and the  *
 * terms and conditions for copying, distribution and        *
 * modification of SableCC.                                   *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */


/* This grammar defines the SableCC 3.x input language. */

Package org.sablecc.sablecc; // Root Java package for generated files.

Helpers

/* These are character sets and regular expressions used in the
   definition of tokens. */

    all = [0 .. 0xFFFF];
    lowercase = ['a' .. 'z'];
    uppercase = ['A' .. 'Z'];
    digit = ['0' .. '9'];
    hex_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];

    tab = 9;
    cr = 13;
    lf = 10;
    eol = cr lf | cr | lf; // This takes care of different platforms

    not_cr_lf = [all - [cr + lf]];
    not_star = [all - '*'];
```

```
    not_star_slash = [not_star - '/'];

    blank = (' ' | tab | eol)+;

    short_comment = '//' not_cr_lf* eol;
    long_comment =
        '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
    comment = short_comment | long_comment;

    letter = lowercase | uppercase | '_' | '$';
    id_part = lowercase (lowercase | digit)*;

States
    normal, /* The first state is the initial state. */
    package;





Tokens

/* These are token definitions. It is allowed to use helper regular *
 * expressions in the body of a token definition.                   *
 * On a given input, the longest valid definition is chosen, In     *
 * case of a match, the definition that appears first is chosen.    *
 * Example: on input -> 's' <- "char" will have precedence on       *
 * "string", because it appears first.                              */

{package}
```

```
    pkg_id = letter (letter | digit)*;

{normal->package}
    package = 'Package';

    states = 'States';
    helpers = 'Helpers';
    tokens = 'Tokens';
    ignored = 'Ignored';
    productions = 'Productions';

    abstract = 'Abstract';
    syntax = 'Syntax';
    tree = 'Tree';
    new = 'New';
    null = 'Null';

    token_specifier = 'T';
    production_specifier = 'P';

    dot = '.';
    d_dot = '..';

{normal, package->normal}
    semicolon = ';';

    equal = '=';
    l_bkt = '[';
    r_bkt = ']';
    l_par = '(';
```

```
    r_par = ')';
    l_brace =  '{';
    r_brace =  '}';
    plus = '+';
    minus = '-';
    q_mark = '?';
    star = '*';
    bar = '|';
    comma = ',';
    slash = '/';
    arrow = '->';
    colon = ':';

    id = id_part ('_' id_part)*;

    char = ''' not_cr_lf ''';
    dec_char = digit+;
    hex_char = '0' ('x' | 'X') hex_digit+;

    string = ''' [not_cr_lf - ''']+ ''';

    blank = blank;
    comment = comment;

Ignored Tokens

/* These tokens are simply ignored by the parser. */

    blank,
    comment;
```

Productions

```
/* These are the productions of the grammar. The first production is *
 * used by the implicit start production:                            *
 *   start = (first production) EOF;                                 *
 * ?, * and + have the same meaning as in a regular expression.      *
 * In case a token and a production share the same name, the use of  *
 * P. (for production) or T. (for token) is required.                *
 * Each alternative can be explicitely named by preceding it with a  *
 * name enclosed in braces.                                          *
 * Each alternative element can be explicitely named by preceding it *
 * with a name enclosed in brackets and followed by a colon.         */


    grammar =
        P.package? P.helpers? P.states? P.tokens? ign_tokens?
        P.productions? P.ast?
       {-> New grammar([P.package.list_pkg_id], P.helpers, P.states,
                       P.tokens, P.ign_tokens, P.productions, P.ast)
       };

    package
       {-> [list_pkg_id]:pkg_id*} =
        T.package pkg_name
       {-> [pkg_name.pkg_id] };
```

```
pkg_name
   {-> pkg_id*} =
    pkg_id [pkg_ids]:pkg_name_tail* semicolon
   {-> [pkg_id, pkg_ids.pkg_id] };

pkg_name_tail
   {-> pkg_id } =
    dot pkg_id
   {-> pkg_id };

helpers =
    T.helpers [helper_defs]:helper_def+
   {-> New helpers([helper_defs]) };

helper_def =
    id equal reg_exp semicolon
   {-> New helper_def(id, reg_exp) };

states =
    T.states id_list semicolon
   {-> New states([id_list.id]) };

id_list
   {-> id*} =
    id [ids]:id_list_tail*
   {-> [id, ids.id]};

id_list_tail
   {-> id } =
    comma id
```

```
   {-> id};

tokens =
    T.tokens [token_defs]:token_def+
   {-> New tokens([token_defs]) };

token_def =
    state_list? id equal reg_exp look_ahead? semicolon
   {-> New token_def(state_list, id, reg_exp, look_ahead.slash,
                     look_ahead.reg_exp) };

state_list =
    l_brace id transition? [state_lists]:state_list_tail* r_brace
   {-> New state_list(id, transition, [state_lists])};

state_list_tail =
    comma id transition?
   {-> New state_list_tail(id, transition) };

transition =
    arrow id
   {-> New transition(id)};

ign_tokens =
    ignored T.tokens id_list? semicolon
   {-> New ign_tokens([id_list.id]) };

look_ahead
   {-> slash reg_exp} =
    slash reg_exp
```

```
   {-> slash reg_exp};

reg_exp =
    concat [concats]:reg_exp_tail*
   {-> New reg_exp([concat, concats.concat])};



reg_exp_tail
   {-> concat } =
    bar concat
   {-> concat};

concat =
    [un_exps]:un_exp*
   {-> New concat([un_exps])};

un_exp =
    basic un_op?;

basic =
    {char}    P.char
   {-> New basic.char(P.char)}      |
    {set}     set
   {-> New basic.set(set)}      |
    {string}  string
   {-> New basic.string(string)}    |
    {id}      id
   {-> New basic.id(id)}        |
    {reg_exp} l_par reg_exp r_par
   {-> New basic.reg_exp(reg_exp)}  ;
```

```
char =
    {char} T.char |
    {dec}  dec_char |
    {hex}  hex_char;

set =
    {operation} l_bkt [left]:basic  bin_op [right]:basic  r_bkt
   {-> New set.operation(left, bin_op, right) } |
    {interval}  l_bkt [left]:P.char d_dot  [right]:P.char r_bkt
   {-> New set.interval(left, right) };

un_op =
    {star}   star
   {-> New un_op.star(star)}   |
    {q_mark} q_mark
   {-> New un_op.q_mark(q_mark)} |
    {plus}   plus
   {-> New un_op.plus(plus)}   ;

bin_op =
    {plus}  plus
   {-> New bin_op.plus()}  |
    {minus} minus
   {-> New bin_op.minus()} ;

productions =
    T.productions [prods]:prod+
   {-> New productions([prods]) };
```

```
prod =
    id prod_transform? equal alts semicolon
   {-> New prod(id, prod_transform.arrow, [prod_transform.elem],
                [alts.list_alt])};

prod_transform
   {-> arrow elem*} =
    l_brace arrow [elems]:elem* r_brace
   {-> arrow [elems]};

alts
   {-> [list_alt]:alt*} =
    alt [alts]:alts_tail*
   {-> [alt, alts.alt]};

alts_tail
   {-> alt} =
    bar alt
   {-> alt};

alt =
    alt_name? [elems]:elem* alt_transform?
   {-> New alt(alt_name.id, [elems], alt_transform)};

alt_transform =
    l_brace arrow [terms]: term* r_brace
   {-> New alt_transform(l_brace, [terms], r_brace)};

term =
    {new} new prod_name l_par params? r_par
```

```
   {-> New term.new(prod_name, l_par, [params.list_term]) }  |

    {list} l_bkt list_of_list_term? r_bkt
   {-> New term.list(l_bkt, [list_of_list_term.list_terms])} |

    {simple} specifier? id simple_term_tail?
   {-> New term.simple(specifier, id, simple_term_tail.id)}  |

    {null} null
   {-> New term.null()}                    ;

list_of_list_term
   {-> [list_terms]:list_term* } =
    list_term [list_terms]:list_term_tail*
   {-> [list_term, list_terms.list_term] }    ;

list_term =
    {new} new prod_name l_par params? r_par
   {-> New list_term.new(prod_name, l_par, [params.list_term])}  |
    {simple} specifier? id simple_term_tail?
   {-> New list_term.simple(specifier, id, simple_term_tail.id)};

list_term_tail
   {-> list_term} =
comma list_term
   {-> list_term} ;

simple_term_tail
   {-> id} =
    dot id
```

```
   {-> id};

prod_name =
    id prod_name_tail?
   {-> New prod_name(id, prod_name_tail.id)};

prod_name_tail
   {-> id} =
    dot id
   {-> id};

params
   {-> [list_term]:term*} =
   term [params]:params_tail*
   {-> [term, params.term]};

params_tail
   {-> term} =
   comma term
   {-> term};

alt_name
   {-> id} =
    l_brace id r_brace
   {-> id};

elem =
    elem_name? specifier? id un_op?
   {-> New elem(elem_name.id, specifier, id, un_op) };
```

```
elem_name
   {-> id} =
    l_bkt id r_bkt colon
   {-> id};

specifier =
    {token}      token_specifier dot
   {-> New specifier.token()}           |
    {production} production_specifier dot
   {-> New specifier.production()}      ;

ast =
    abstract syntax tree [prods]:ast_prod+
   {-> New ast([prods]) };

ast_prod =
    id equal [alts]:ast_alts semicolon
   {-> New ast_prod(id, [alts.list_ast_alt])};

ast_alts
   {-> [list_ast_alt]:ast_alt*} =
    ast_alt [ast_alts]:ast_alts_tail*
   {-> [ast_alt, ast_alts.ast_alt]};

ast_alts_tail
   {-> ast_alt} =
    bar ast_alt
   {-> ast_alt};

ast_alt =
```

```
        alt_name? [elems]:elem*
       {-> New ast_alt(alt_name.id, [elems])};


/*********************************************************************
 *********************************************************************
 *********************************************************************
 *********************************************************************
 *********************************************************************/

Abstract Syntax Tree

    grammar =
        [package]:pkg_id* P.helpers? P.states? P.tokens? P.ign_tokens?
        P.productions? P.ast?;

    helpers =
        [helper_defs]:helper_def*;

    helper_def =
        id reg_exp;

    states =
    [list_id]:id*;

    tokens =
        [token_defs]:token_def*;

    token_def =
        state_list? id reg_exp slash? [look_ahead]:reg_exp?;
```

```
state_list =
    id transition? [state_lists]:state_list_tail*;

state_list_tail =
    id transition?;

transition =
    id;

ign_tokens =
    [list_id]:id*;

reg_exp =
    [concats]:concat*;

concat =
[un_exps]: un_exp*;

un_exp =
    basic un_op?;

basic =
    {char}    P.char |
    {set}     set |
    {string}  string |
    {id}      id |
    {reg_exp} reg_exp;

char =
```

```
    {char} T.char |
    {dec}  dec_char |
    {hex}  hex_char;

set =
    {operation} [left]:basic bin_op [right]:basic |
    {interval}  [left]:P.char [right]:P.char ;

un_op =
    {star}   star   |
    {q_mark} q_mark |
    {plus}   plus   ;

bin_op =
    {plus} |
    {minus};

productions =
    [prods]:prod*;

prod =
    id arrow? [prod_transform]:elem* [alts]:alt*;

alt =
    [alt_name]:id? [elems]:elem* alt_transform?;

alt_transform =
l_brace [terms]:term* r_brace;

term =
```

```
    {new} prod_name l_par [params]:term* |
    {list} l_bkt [list_terms]:list_term* |
    {simple} specifier? id [simple_term_tail]:id? |
    {null} ;

list_term =
    {new} prod_name l_par [params]:term* |
    {simple} specifier? id [simple_term_tail]:id? ;

prod_name =
    id [prod_name_tail]:id? ;

elem =
    [elem_name]:id? specifier? id un_op?;

specifier =
    {token}     |
    {production}    ;

ast =
    [prods]:ast_prod*;

ast_prod =
    id [alts]:ast_alt*;

ast_alt =
    [alt_name]:id? [elems]:elem*;
```

Annexe B

FICHIER DE SPÉCIFICATION DES EXPRESSIONS ARITHMÉTIQUES

Ce document contient le fichier de spécification d'expressions arithmétiques extrêmement simples. En effet, les seuls opérateurs acceptés sont : "`+`", "`-`", "`*`" et "`/`". La section `Productions` intègre des transformations d'arbres syntaxiques qui permettent de simplifier l'arbre syntaxique. La section `Abstract Syntax Tree` contient les différents noeuds de l'arbre syntaxique final qui sera construit par SableCC.

```
Package expressionlist;

Helpers

    digit = ['0' .. '9'];

    tab = 9;
    cr = 13;
    lf = 10;
    eol = cr lf | cr | lf; // This takes care of different platforms
    tous = [0 .. 0xffff];
    blank = (' ' | tab | eol)+;

Tokens
```

```
    l_par = '(';
    r_par = ')';
    plus = '+';
    minus = '-';
    mult = '*';
    div = '/';
    comma = ',';

    blank = blank;
    number = digit+;
    comment = '/*' tous* '*/';

Ignored Tokens

    blank,
    comment;

Productions

    exp_grammar
           { -> ast_exp_grammar} =
     exp_list
           { -> New ast_exp_grammar([exp_list.exp])}
     ;

    exp_list
           {-> exp+ } =
     exp  exp_list_tail
           {-> [exp, exp_list_tail.exp] }
     ;
```

```
exp_list_tail
    {-> exp } =
 comma exp
    {-> exp}
 ;

exp
    {-> exp? } =
 {plus}   exp plus factor
    {-> New exp.plus(exp, factor.exp)   } |
 {minus}  exp minus factor
    {-> New exp.minus(exp, factor.exp)  } |
 {factor} factor
            {-> factor.exp}
 ;


factor
    {-> exp? } =
 {mult}      factor mult term
    {-> New exp.mult(factor.exp, term.exp ) } |
     {div}       factor div term
    {-> New exp.div(factor.exp, term.exp )  } |
     {term}      term
            {-> term.exp}
 ;

term
    {-> exp? } =
```

```
 {number}   number
            {-> New exp.number(number) } |
     {exp}      l_par exp r_par
    {->  exp};
```

Abstract Syntax Tree

```
ast_exp_grammar = exp*;

exp = {plus}   [l]:exp [r]:exp? |
  {minus}  [l]:exp [r]:exp? |
  {div}    [l]:exp? [r]:exp? |
  {mult}   [l]:exp? [r]:exp? |
  {number} number;

expression =
  {list} expression* |
  {single} exp;
```

Annexe C

FICHIER DE SPÉCIFICATION DU LANGAGE SUPPORTÉE PAR JJOOS

Ce document contient le fichier de spécification du langage JJOOS. JJOOS est un compilateur développé comme projet de session pour un cours avancé sur les techniques de compilation par Othman Alaoui à l'université *McGill* (Alaoui, 2000). Ce compilateur utilise un langage qui est un sous ensemble du langage de programmation *Java*. La version du compilateur développée par Othman utilise une version de SableCC qui n'inclut pas les transformations d'arbres syntaxiques car la version intégrant cette fonctionnalité n'était pas encore développée. Toutefois, des transformations ont été faites sur l'arbre syntaxique à l'intérieur du compilateur grâce à du code écrit manuellement. Nous avons modifié le compilateur JJOOS en nous servant de SableCC3-beta3. Le travail effectué pour cette réécriture a consisté dans un premier temps à insérer des transformations d'arbres syntaxiques dans la grammaire du langage supporté par JJOOS et, dans un second temps, à éliminer l'ancien code des transformations écrit à la main. Nous présentons dans la suite de ce document la spécification complète du langage JJOOS avec les transformations d'arbres syntaxiques.

```
/*
 * JOOSJ is Copyright (C) 2000 Othman Alaoui
 *
 * Reproduction of all or part of this software is permitted for
 * educational or research use on condition that this copyright notice is
 * included in any copy. This software comes with no warranty of any
 * kind. In no event will the author be liable for any damages resulting from
 * use of this software.
 *
 * email: oalaou@po-box.mcgill.ca
 */




/*
 * Glossary:
 *  us = underscore
 *  stm = statement
 *  exp = expression
 */

Package sablecc3.joosc;

/*******************************************************************
 * Helpers                                                         *
 *******************************************************************/
Helpers

    all =                   [0..0xffff];
    letter =                [['a'..'z'] + ['A'..'Z']];
```

```
    digit =                 ['0'..'9'];
    letter_or_digit =       letter | digit;
    letter_or_digit_or_us = letter_or_digit | '_';
    letter_or_us =           letter | '_';
    nonzero_digit =         ['1'..'9'];
    octal_digit =           ['0'..'7'];
    lf =                    10;
    sp =                    32;
    ht =                    9;
    line_terminator =       lf;
    octal_escape =          '\' octal_digit octal_digit octal_digit;
    esc_sequence =          '\b' | '\t' | '\n' | '\f' | '\r' |
                            '\"' | '\' ''' | '\\' | octal_escape;
    /*@A+begin*/
    star =                  '*';
    slash =                 '/';
    not_star =              [all - star];
    not_star_slash =        [not_star - slash];
    /*@A+end*/

/*******************************************************************
 * Tokens                                                          *
 *******************************************************************/
Tokens

    blanks =                (sp|ht|line_terminator)+;
    eol_comment =           '//' [all - line_terminator]* line_terminator;
    /*@A+begin*/
    // alternatively, could have used lexer states
    ext_comment =           '/*' not_star* star+ (not_star_slash not_star* star+)* '/
```

```
/*A+end*/

/*********************
 Keywords
**********************/
abstract =              'abstract';
boolean =               'boolean';
break =                 'break';
byte =                  'byte';
case =                  'case';
catch =                 'catch';
char =                  'char';
class =                 'class';
const =                 'const';
continue =              'continue';
default =               'default';
do =                    'do';
double =                'double';
else =                  'else';
extends =               'extends';
extern =                'extern';
final =                 'final';
finally =               'finally';
float =                 'float';
for =                   'for';
goto =                  'goto';
if =                    'if';
implements =            'implements';
import =                'import';
in =                    'in';
```

```
instanceof =            'instanceof';
int =                   'int';
interface =             'interface';
long =                  'long';
main =                  'main';
native =                'native';
new =                   'new';
package =               'package';
private =               'private';
protected =             'protected';
public =                'public';
return =                'return';
short =                 'short';
static =                'static';
super =                 'super';
switch =                'switch';
synchronized =          'synchronized';
this =                  'this';
throw =                 'throw';
throws =                'throws';
transient =             'transient';
try =                   'try';
void =                  'void';
volatile =              'volatile';
while =                 'while';

/*********************
 Operators
**********************/
assign =                '=';
```

```
gt =                    '>';
lt =                    '<';
not =                   '!';
eq =                    '==';
leq =                   '<=';
geq =                   '>=';
neq =                   '!=';
and =                   '&&';
or =                    '||';
plus =                  '+';
minus =                 '-';
mult =                  '*';
div =                   '/';
mod =                   '%';
l_brace =               '{';
r_brace =               '}';
semicolon =             ';';
l_par =                 '(';
r_par =                 ')';
l_bracket =             '[';
r_bracket =             ']';
comma =                 ',';
dot =                   '.';
inc =                   '++';

/*********************
 Literals
**********************/
null =                  'null';
true =                  'true';
```

```
    false =                 'false';
    charconst =             ''' ([[[all - line_terminator] - '''] - '\'] |
                                 esc_sequence) ''';
    intconst =              '0' | (nonzero_digit digit*);
    stringconst =           '"' [all - '"']* '"';

    identifier =            letter_or_us letter_or_digit_or_us*;

    importpath =            'import '
                            (letter_or_us letter_or_digit_or_us* '.')*
                            ('*' | letter_or_us letter_or_digit_or_us*)
                            ';';

/*****************************************************************
 * Ignored Tokens                                                *
 *****************************************************************/
Ignored Tokens

    blanks,
    eol_comment/*A+begin*/,
    ext_comment/*A+end*/;

/*****************************************************************
 * Productions                                                   *
 *****************************************************************/
Productions

classfile =
  {default} importpath* [p_class]:P.class
            {-> New classfile.default(p_class)} |
```

```
  {extern}  extern_class+;



class =
  public classmods? [t_class]:T.class identifier extension? l_brace
  field* constructor+ method* r_brace
  {-> New class(classmods, identifier, extension, [field],
                [constructor], [method])};

classmods =
  {final}    final |
  {abstract} abstract;

extern_class =
  extern public classmods? [t_class]:T.class identifier extension?
  in stringconst l_brace extern_constructor+ extern_method* r_brace
  {-> New extern_class(classmods, identifier, extension, stringconst,
                       [extern_constructor], [extern_method])};

extension =
  extends identifier
  {-> New extension(identifier)};

type =
  {reference} identifier |
  {char}      char |
  {boolean}   boolean |
  {int}       int;

field =
```

```
  protected type identifier_list semicolon
  {-> New field.first(type, [identifier_list.identifier])};

identifier_list {-> identifier*} =
  identifier identifier_list_tail*
  {-> [identifier, identifier_list_tail.identifier]};

identifier_list_tail {-> identifier} =
  comma identifier
  {-> identifier};

constructor =
  public identifier l_par formal_list? r_par l_brace super
  [super_l_par]:l_par argument_list? [super_r_par]:r_par semicolon
  stm* r_brace
  {-> New constructor(identifier, [formal_list.formal],
                      [New stm.supercons([argument_list.exp]), stm])};

extern_constructor =
  public identifier l_par formal_list? r_par semicolon
  {-> New extern_constructor(identifier, [formal_list.formal])};

formal_list {-> formal*} =
  formal formal_list_tail*
  {-> [formal, formal_list_tail.formal]};

formal_list_tail {-> formal } =
  comma formal
  {-> formal};
```

```
formal =
  type identifier;

method =
  {mod}      public methodmods returntype identifier l_par
             formal_list? r_par l_brace stm* r_brace
             {-> New method.mod(methodmods, returntype, identifier,
                                [formal_list.formal], [stm])} |
  {nonmod}   public returntype identifier l_par formal_list? r_par
             l_brace stm* r_brace
             {-> New method.nonmod(returntype, identifier,
                                   [formal_list.formal], [stm])} |
  {abstract} public abstract returntype identifier l_par formal_list?
             r_par semicolon
             {-> New method.abstract(returntype, identifier,
                                     [formal_list.formal])} |
  {main}     public static void main l_par mainargv r_par l_brace
             stm* r_brace
             {-> New method.main(mainargv, [stm])};

methodmods =
  {final}        final |
  {synchronized} synchronized;

mainargv =
  {first}  [type]:identifier [name]:identifier l_bracket r_bracket
           {-> New mainargv(type, name)} |
  {second} [type]:identifier l_bracket r_bracket [name]:identifier
           {-> New mainargv(type, name)};
```

```
extern_method =
  {mod}    public extern_methodmods returntype identifier l_par
           formal_list? r_par semicolon
           {-> New extern_method.mod(extern_methodmods, returntype,
                                     identifier,
                                     [formal_list.formal])} |
  {nonmod} public returntype identifier l_par formal_list? r_par
           semicolon
           {-> New extern_method.nonmod(returntype, identifier,
                                        [formal_list.formal])};

extern_methodmods =
  {final}        final |
  {abstract}     abstract |
  {synchronized} synchronized;

returntype =
  {void}    void
            {-> New returntype.void()} |
  {nonvoid} type;

stm =
  {simple} simplestm
           {-> simplestm.stm} |
  {decl}   type identifier_list semicolon
           {-> New stm.decl_first(type,
                                  [identifier_list.identifier]) } |
  {if}     if l_par exp r_par stm
           {-> New stm.if(exp, stm)} |
  {ifelse} if l_par exp r_par stm_no_short_if else stm
```

```
      {-> New stm.ifelse(exp, stm_no_short_if.stm, stm) } |
  /*@A+begin*/
  {for}        for l_par [initializer]:stm_exp [semicolon1]:semicolon exp
                 [semicolon2]:semicolon [updater]:stm_exp r_par stm
      {-> New stm.for(initializer.exp, exp, updater.exp, stm)}|
  /*@A+end*/
  {while}  while l_par exp r_par stm
           {-> New stm.while(exp, stm)};

simplestm {-> stm} =
  {skip}   semicolon
           {-> New stm.skip()} |
  {block}  l_brace stm* r_brace
           {-> New stm.block([stm])} |
  {exp}    stm_exp semicolon
           {-> New stm.exp(stm_exp.exp)} |
  {return} return exp? semicolon
           {-> New stm.return(exp)} ;

stm_no_short_if {-> stm} =
  {simple}     simplestm
               {-> simplestm.stm} |
  {tmp_ifelse} if l_par exp r_par
               [then_stm_no_short_if]:stm_no_short_if else
               [else_stm_no_short_if]:stm_no_short_if
               {-> New stm.ifelse(exp, then_stm_no_short_if.stm,
                                  else_stm_no_short_if.stm)} |
  /*@A+begin*/
  {tmp_for}    for l_par [initializer]:stm_exp [semicolon1]:semicolon exp
                 [semicolon2]:semicolon [updater]:stm_exp r_par
```

```
                stm_no_short_if
        {-> New stm.for(initializer.exp, exp, updater.exp, stm_no_short_if.stm)}|
  // intermediate AST transformation for replacement of noshortif stms
  /*@A+end*/
  {tmp_while}  while l_par exp r_par stm_no_short_if
               {-> New stm.while(exp, stm_no_short_if.stm)};

stm_exp {-> exp} =
  {assign} assignment
           {-> assignment.exp} |
  {call}   methodinvocation
           {-> methodinvocation.exp} |
  /*@A+begin*/
  {inc}       identifier inc
       {-> New exp.inc(identifier)}|
  /*@A+end*/
  {new}    classinstancecreation
           {-> classinstancecreation.exp};

assignment {-> exp} =
  identifier assign exp
  {-> New exp.assign(identifier, exp)};

// transformation: collapse precedence cascade below into one production
exp =
  // 'default' nodes not present in "fixed" AST
  {default} or_exp
            {-> or_exp.exp } |
  {assign}  assignment
            {-> assignment.exp};
```

```
// all *_exp productions alternatives below deleted from AST
// (replaced by hidden exp alternatives)

or_exp {-> exp } =
  {default} and_exp
            {-> and_exp.exp } |
  {or}      [left]:or_exp or [right]:and_exp
            {-> New exp.or(left.exp, right.exp) };

and_exp {-> exp } =
  {default} eq_exp
            {-> eq_exp.exp } |
  {and}     [left]:and_exp and [right]:eq_exp
            {-> New exp.and(left.exp, right.exp) };

eq_exp {-> exp } =
  {default} rel_exp
            {-> rel_exp.exp } |
  {eq}      [left]:eq_exp eq [right]:rel_exp
            {-> New exp.eq(left.exp, right.exp) } |
  {neq}     [left]:eq_exp neq [right]:rel_exp
            {-> New exp.neq(left.exp, right.exp)};

rel_exp {-> exp } =
  {default}    add_exp
               {-> add_exp.exp } |
  {lt}         [left]:rel_exp lt [right]:add_exp
               {-> New exp.lt(left.exp, right.exp) } |
  {gt}         [left]:rel_exp gt [right]:add_exp
```

```
                {-> New exp.gt(left.exp, right.exp) } |
  {leq}         [left]:rel_exp leq [right]:add_exp
                {-> New exp.leq(left.exp, right.exp) } |
  {geq}         [left]:rel_exp geq [right]:add_exp
                {-> New exp.geq(left.exp, right.exp) } |
  {instanceof} rel_exp instanceof identifier
                {-> New exp.instanceof(rel_exp.exp, identifier)};

add_exp {-> exp } =
  {default} mult_exp
            {-> mult_exp.exp } |
  {plus}    [left]:add_exp plus [right]:mult_exp
            {-> New exp.plus(left.exp, right.exp) } |
  {minus}   [left]:add_exp minus [right]:mult_exp
            {-> New exp.minus(left.exp, right.exp)};

mult_exp {-> exp } =
  {default} unary_exp
              {-> unary_exp.exp } |
  {mult}      [left]:mult_exp mult [right]:unary_exp
              {-> New exp.mult(left.exp, right.exp) } |
  {div}       [left]:mult_exp div [right]:unary_exp
              {-> New exp.div(left.exp, right.exp) } |
  {mod}       [left]:mult_exp mod [right]:unary_exp
              {-> New exp.mod(left.exp, right.exp)};

unary_exp {-> exp } =
  {default} unary_exp_not_minus
            {-> unary_exp_not_minus.exp } |
  {minus}   minus unary_exp
```

```
            {-> New exp.uminus(unary_exp.exp)};

unary_exp_not_minus {-> exp } =
  {default} postfix_exp
            {-> postfix_exp.exp } |
  {not}     not unary_exp
            {-> New exp.not(unary_exp.exp) } |
  {cast}    cast_exp
            {-> cast_exp.exp };

cast_exp {-> exp } =
  {nonchar} l_par exp r_par unary_exp_not_minus
            {-> New exp.tmpcast(exp, unary_exp_not_minus.exp) } |
  {char}    l_par char r_par unary_exp
            {-> New exp.casttochar(unary_exp.exp)};

postfix_exp {-> exp } =
  {id}      identifier
            {-> New exp.id(identifier) } |
  {primary} primary_exp
            {-> primary_exp.exp};

primary_exp {-> exp } =
  {literal} literal
            {-> literal.exp } |
  {this}    this
            {-> New exp.this() } |
  {paren}   l_par exp r_par
            {-> exp.exp } |
  {new}     classinstancecreation
```

```
            {-> classinstancecreation.exp } |
  {call}    methodinvocation
            {-> methodinvocation.exp };

classinstancecreation {-> exp}=
  new identifier l_par argument_list? r_par
  {-> New exp.new(identifier, [argument_list.exp])};

methodinvocation {-> exp}=
  receiver dot identifier l_par argument_list? r_par
  {-> New exp.call(receiver, identifier, [argument_list.exp])};

receiver =
  {tmpobject} postfix_exp
              {-> New receiver.object(postfix_exp.exp) } |
  // transformation necessary as a result of precedence cascade collapse
  {super}     super
              {-> New receiver.super()};

argument_list{-> exp*} =
  exp argument_list_tail*
  {-> [exp, argument_list_tail.exp]};

argument_list_tail {-> exp } =
  comma exp
  {-> exp };

// literal production alternatives deleted from AST
// (replaced by hidden exp alternatives)
literal {-> exp } =
```

```
  {int}    intconst
           {-> New exp.intconst(intconst) } |
  {true}   true
           {-> New exp.true() } |
  {false}  false
           {-> New exp.false() } |
  {char}   charconst
           {-> New exp.charconst(charconst) } |
  {string} stringconst
           {-> New exp.stringconst(stringconst) } |
  {null}   null
           {-> New exp.null()};


/*******************************************************************
 * Abstract Syntax  Tree                                           *
 *******************************************************************/
Abstract Syntax Tree

classfile =
  {default} [p_class]:P.class |
  {extern}  [classes]:extern_class*;

class =
  classmods? identifier extension? [fields]:field*
  [constructors]:constructor* [methods]:method* ;

classmods =
  {final}    final |
  {abstract} abstract;
```

```
extern_class =
  classmods? identifier extension? stringconst
  [constructors]:extern_constructor* [methods]:extern_method* ;

extension =
  identifier;

type =
  {reference} identifier |
  {char}      char |
  {boolean}   boolean |
  {int}       int |
  // hidden alternatives: generated type classes used for type-checking
  {polynull}  |
  {void}      ;

field =
  {first} type [identifiers]:identifier* |
  [fields]:onefield*;

// used by hidden alternative
onefield =
  type identifier ;

constructor =
  identifier [formals]:formal* [stmts]:stm* ;

extern_constructor =
  identifier [formals]:formal* ;
```

```
formal =
  type identifier;

method =
  {mod}      methodmods returntype identifier [formals]:formal* [stmts]:stm* |
  {nonmod}   returntype identifier [formals]:formal* [stmts]:stm* |
  {abstract} returntype identifier [formals]:formal* |
  {main}     mainargv [stmts]:stm* ;

methodmods =
  {final}        final |
  {synchronized} synchronized;

mainargv =
  [type]:identifier [name]:identifier;

extern_method =
  {mod}    extern_methodmods returntype identifier [formals]:formal* |
  {nonmod} returntype identifier [formals]:formal* ;

extern_methodmods =
  {final}        final |
  {abstract}     abstract |
  {synchronized} synchronized;

returntype =
  {void}    |
  {nonvoid} type;
```

```
stm =
  {skip}       |
  {block}      [stmts]:stm* |
  {exp}        exp |
  {return}     exp? |
  {decl_first} type [identifiers]:identifier*|
  {decl}       [locals]:onelocal* |
  {supercons}  [args]:exp* |
  {if}         exp stm |
  {ifelse}     exp [then_stm]:stm [else_stm]:stm |
  {for}        [initializer]:exp exp [updater]:exp stm |
  {while}      exp stm;

// used by hidden decl alternative of stm production
onelocal =
  type identifier ;

exp =
  {assign}      identifier exp|
  {or}          [left]:exp [right]:exp |
  {and}         [left]:exp [right]:exp |
  {eq}          [left]:exp [right]:exp |
  {neq}         [left]:exp [right]:exp |
  {lt}          [left]:exp [right]:exp |
  {gt}          [left]:exp [right]:exp |
  {leq}         [left]:exp [right]:exp |
  {geq}         [left]:exp [right]:exp |
  {instanceof}  exp identifier |
  {plus}        [left]:exp [right]:exp |
  {minus}       [left]:exp [right]:exp |
```

```
  {mult}        [left]:exp [right]:exp |
  {div}         [left]:exp [right]:exp |
  {mod}         [left]:exp [right]:exp |
  {uminus}      exp |
  {not}         exp |
  {tmpcast}     [caster]:exp [castee]:exp |
  {cast}        identifier exp |
  {casttochar}  exp |
  {id}          identifier |
  {this}        |
  {new}         identifier [args]:exp* |
  {call}        receiver identifier [args]:exp*|
  {intconst}    intconst |
  {true}        |
  {false}       |
  {charconst}   charconst |
  {stringconst} stringconst |
  {inc}     identifier |
  {null}        ;

receiver =
  {object} exp |
  {super}  ;
```

Bibliographie

(Aho, Sethi et Ullman, 2000) Aho, A., Sethi, R., et Ullman, J. 2000. *Compilateurs Principes, techniques et outils*. Dunod.

(Alaoui, 2000) Alaoui, O. 2000. Developping a JJOOS compiler using SableCC framework. `http://www.cs.mcgill.ca/~hendren/SableCC/JJOOS/report.ps`.

(Appel, 1998) Appel, A. 1998. *Modern compiler implementation in Java*. Cambridge University Press.

(Cordy, 2004) Cordy, J. R. 2004. « TXL - a language for programming language tools and applications ». In *International Workshop on Language Descriptions, Tools and Applications*, p. 1–27. ACM.

(Cordy, Halpem et Promislow, 1988) Cordy, J. R., Halpem, C. D., et Promislow, E. 1988. « TXL : A rapid prototyping system for programming language dialects ». In *International Conference on Computer Languages*, p. 280–285. IEEE Computer Society.

(Gagnon, 1998) Gagnon, E. 1998. « SableCC, an object-oriented compiler framework ». Mémoire de maîtrise, Computer Science, McGill. `http://sablecc.org/thesis/thesis.html`.

(Gagnon et Hendren, 1998) Gagnon, E. et Hendren, L. 1998. « SableCC, an object-oriented compiler framework ». In *TOOLS '98 : Proceedings of the Technology of Object-Oriented Languages and Systems*, p. 140–154. IEEE Computer Society.

(Gamma et al., 1995) Gamma, E., Helm, R., Johnson, R., et Vlissides, J. 1995. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley Prefessional Computing series.

(Levine, Mason et Brown, 1992) Levine, J., Mason, T., et Brown, D. 1992. *Lex & Yacc*. O'reilly.

(MicroSystem, 2000) MicroSystem, S. 2000. JJTree home page. `https://javacc.dev.java.net/doc/JJTree.html`.

(Parr, 2005) Parr, T. J. 2005. ANTLR reference manual. `http://www.antlr.org/doc/index.html`.

(Parr et Quong, 1995) Parr, T. J. et Quong, R. W. 1995. « ANTLR : A predicated-LL(k) parser generator », *Software Practice and Experience*, vol. 25, no. 7, p. 789–810.

(Schroer, 1997) Schroer, F. W. 1997. The Gentle compiler construction system. `http://gentle.compilertools.net/book/index.html`.

(Tao, Wang et Palsberg, 1997) Tao, K., Wang, W., et Palsberg, J. 1997. Documentation homepage of java tree builder web page. http://compilers.cs.ucla.edu/jtb/jtb-2003/.