UNIVERSITÉ DU QUÉBEC À MONTRÉAL

HIBER, UN LANGAGE POUR DÉCRIRE LES RÈGLES DE GÉNÉRATION DE CODE DANS SABLECC

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

DAVID CODERRE

JANVIER 2007

REMERCIEMENTS

En premier lieu, je me dois de remercier mon directeur de recherche *Étienne Gagnon*, professeur d'informatique à l'Université du Québec à Montréal, pour sa précieuse collaboration. Il a fait preuve d'une grande patience tout au court de la rédaction et de la correction de ce mémoire.

Je désire également souligner la précieuse collaboration de *Komivi Kevin Agbakpem*, étudiant et membre de la communauté de *SableCC*. J'ai apprécié son aide et la clarté de ses réponses.

Je tiens en plus à mentionner *Indrek Mandre* avec qui j'ai pu m'entretenir à propos de son outil *Alternative Output*. Il m'a aidé à éclaircir certaines ambiguïtés et je me suis inspiré de ses modèles de génération dans le langage *C++*. Dans le même ordre d'idée, j'aimerais citer *Fidel Veigas*. Il a écrit des modèles de génération dans le langage *Python* pour l'outil *Alternative Output*.

Je me dois de ne pas oublier *Richard Chabot* pour les vérifications techniques nécessaires à la mise à jour du vocabulaire illustré.

J'ajoute une mention honorable aux membres de ma famille qui m'ont encouragé et épaulé pendant ces longues heures d'élaboration du mémoire. Je termine avec des égards particuliers envers mes collègues de travail qui m'ont fait part de leurs suggestions et de leurs commentaires constructifs visant à achever ce travail.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS ET ACRONYMES

ANTLR *ANother Tool for Language Recognition*, Un autre outil pour la reconnaissance de langage.

ASCII *American Standard Code for Information Interchange*, Code standard américain pour l'échange d'information.

AST *Abstract Syntax Tree*, Arbre syntaxique abstrait.

BNF *Backus-Naur Form*, Forme Backus-Naur. Métalangage utilisé pour décrire et exprimer la syntaxe des langages de programmation.

CST *Concrete Syntax Tree*, Arbre Syntaxique Concret.

AFD Automate fini déterministe.

EBNF *Extended Backus-Naur Form*, Forme Backus-Naur étendue.

GNU *GNU's Not UNIX*, GNU n'est pas Unix.

HTML *HyperText Markup Language*, Langage HTML. Langage de balisage de texte qui permet la création de documents hypertextes affichables par un navigateur Web. (World Wide Web Consortium)

LALR *LookAhead LR*, Pré-vision basée sur LR. (Appel, 2002)

LL *Left to left scanning, leftmost derivation*, Parcours de l'entrée de la gauche vers la droite, dérivation gauche. (Appel, 2002)

LR *Left to right scanning of the input, rightmost derivation in reverse*, Parcours de l'entrée de la gauche vers la droite, en construisant une dérivation droite inverse. (Appel, 2002)

XML *Extensible markup language*, Langage XML. Évolution du langage SGML permettant aux concepteurs de documents HTML de définir leurs propres marqueurs, dans le but de personnaliser la structure des données qu'ils comptent présenter. (World Wide Web Consortium)

GLOSSAIRE

Les définitions sont extraites du grand dictionnaire terminologique de l'office québécois de la langue française. (Grand dictionnaire terminologique).

Arbre syntaxique Représentation arborescente qui définit la structure syntaxique d'un élément de langage en le décomposant en ses unités lexicales et en établissant les relations entre elles.

Casse Attribut d'un caractère typographique pouvant se présenter sous l'aspect d'un bas de casse (lettre minuscule), d'une capitale (lettre majuscule) ou d'une petite capitale (lettre qui a la forme d'une capitale, mais la taille d'un bas de casse).

Fonction Sous-programme constitué d'un ensemble d'instructions identifiées par un nom, dont l'exécution produit une valeur qui renvoie à l'endroit où le sous-programme a été appelé.

Interpréteur Programme qui traduit les instructions d'un langage évolué en langage machine et les exécute au fur et à mesure qu'elles se présentent.

Macro Instruction complexe écrite dans un langage d'assemblage ou créée par l'utilisateur, qui représente une suite d'instructions réelles en langage machine et qui est destinée à être remplacée par cette suite chaque fois qu'elle apparaît dans un programme.

Paquetage En programmation, conteneur qui, au moyen d'un mot-clé commun, regroupe des classes partageant des relations logiques.

Procédure	Séquence d'instructions constituant un sous-programme.
Script	Série d'instructions servant à accomplir une tâche particulière.
Sémantique	Ensemble des relations entre les caractères, ou groupes de caractères, et leur signification, indépendamment de la façon de les employer ou de les interpréter.
URL	Chaîne de caractères normalisés servant à identifier et à localiser des ressources consultables sur Internet et à y accéder à l'aide d'un navigateur.
Web	Système basé sur l'utilisation de l'hypertexte, qui permet la recherche d'information dans Internet, l'accès à cette information et sa visualisation.

RÉSUMÉ

SableCC est un générateur de compilateurs. Il permet de construire des analyseurs lexicaux, des analyseurs syntaxiques et des arbres syntaxiques. Il est toutefois limité à la génération de code en langage *Java*. Dans ce mémoire, nous présentons un outil qui s'intègre à *SableCC* et qui permet à celui-ci une génération de code dans d'autres langages orientés objets.

La méthode que nous proposons pour permettre la génération de code dans un autre langage consiste à construire des scripts et des patrons pour chaque nouveau langage. La rédaction des scripts se fait dans le langage *Hiber*, un langage que nous avons conçu pour faciliter la tâche de décrire les règles de génération de code. Les scripts sont interprétés par un interpréteur construit avec *SableCC*. Cet interpréteur reçoit de *SableCC* une structure de données qui est accessible à partir de scripts. Le principe original de *SableCC* pour la génération de code est donc conservé.

Pour vérifier le bon fonctionnement de l'interpréteur et de la structure de données, nous avons effectué la génération en *Java* avec notre méthode et nous avons comparé le code généré avec celui de la version originale de *SableCC*. Par la suite, nous avons ajouté la génération en *C++*. C'est un langage ayant des similitudes avec *Java*. Cet exercice nous a permis de raffiner l'interpréteur et la structure de données. Dans le but de démontrer une plus grande maturité de la génération, un troisième langage est ajouté : *Python*. Ce langage est très différent des deux derniers, ce qui démontre l'adéquation de notre méthode de génération pour les langages orientés objets.

La méthode de génération proposée permet de bien séparer les modèles de génération, les règles de génération et la structure de données. Elle rend possible l'ajout d'un nouveau langage destination en écrivant un script qui est peu volumineux, facile à lire et à maintenir.

Mots clés : Générateur de compilateur, interpréteur, structure de données, *Hiber*, script.

SUMMARY

SableCC is a compiler generator. It allows building lexers, parsers and syntax trees. However *SableCC* is limited to the generation of code in Java only. In this thesis, we present a tool being integrated into *SableCC* which allows the generation of code into other object oriented languages.

The method we propose, for the generation of code into another language, consists of building scripts and templates for each new language. The drafting of scripts is done in the *Hiber* language, a language conceived to facilitate the task of defining the rules for code generation. Scripts are interpreted by an interpreter built with *SableCC*. This interpreter receives from *SableCC* a data structure which is accessible from the script. The original principle of *SableCC* for the generation of code is preserved.

To verify that the interpreter is operating correctly and that the structure of data is correct, we carried out the generation into *Java* with our method and we compared the code generated with that of the original version of *SableCC*. Thereafter, we added the ability to generate code in *C++*. It is a language with similarities to *Java*. This exercise enabled us to refine the interpreter and the generated data structure. In an effort to test the flexibility of the code generator, we generated *Python* code. This language is very different from the previous languages, which shows the flexibility of our method for generating code in object oriented languages.

The proposed method of generation allows us to clearly separate the templates, the rules of generation, and the data structure. It allows for the generation of new languages by writing compact, easy to read and to maintain scripts.

Keywords: Compiler generator, interpreter, data structure, *Hiber*, script.

INTRODUCTION

SableCC est un générateur de compilateurs. Il permet de construire des analyseurs lexicaux, des analyseurs syntaxiques et des arbres syntaxiques en *Java* à partir d'une grammaire définie selon les normes de *SableCC*. Présentement, la génération de code se fait seulement en *Java*. Le but de ce mémoire est de permettre à *SableCC* de générer des analyseurs dans d'autres langages. *SableCC* deviendrait plus polyvalent et pourrait être utilisé par n'importe quel programmeur à l'aise dans un autre langage que *Java*.

Pour que *SableCC* puisse générer du code dans un autre langage, il importe de bien en comprendre le fonctionnement lors de la création des analyseurs. De plus, nous devons prévoir un processus afin d'ajouter les spécifications pour un nouveau langage destination. Il faut conserver l'ensemble le plus simple et le plus intuitif possible pour l'utilisateur.

SableCC prend en entrée un fichier de spécifications. Ce fichier représente une grammaire avec ses jetons et ses productions. À partir de ce fichier, *SableCC* produit un ensemble de classes qui représente cette grammaire. La version originale de *SableCC* construit des classes Java en mettant en œuvre des analyseurs lexicaux et syntaxiques. L'outil *SableCC* fonctionne avec des fichiers patrons. Ces fichiers contiennent les motifs en *Java* sur la structure des classes à générer. Il est intéressant de garder ce type de fichiers patrons car ils permettent de bien séparer la logique du code généré de la forme de celui-ci. Dans ce mémoire nous proposons donc une méthode où chaque nouveau langage doit avoir un ou plusieurs fichiers dans lesquels les motifs de la structure des classes sont insérés. Cependant, la façon de générer les classes d'un langage à un autre peut être très différente : le nombre de fichiers peut varier, chaque langage ayant sa propre syntaxe. C'est pourquoi, il importe de spécifier une méthode pour bien définir certaines règles lors de la génération du code.

La méthode proposée est d'écrire ces règles dans un script avec un langage simple. Cela a donné naissance à *Hiber*. *Hiber* est un langage très simple à utiliser. Il comprend très

peu de règles syntaxiques et de mots réservés. Il permet de déclarer des variables locales ou globales, des fonctions, des procédures et des notions arithmétiques et logiques. Avec ce langage, il est possible de construire des scripts qui seront exécutés par un interpréteur construit pour la grammaire *Hiber*. C'est dans ces scripts que sont indiqués les fichiers patrons à utiliser et les fichiers à créer pour la génération de la structure de classes. L'interpréteur reconnaît certaines fonctions et procédures prédéfinies qui peuvent être utilisées dans les scripts. Elles servent, entre autres, à faire certains traitements et à créer des fichiers.

Ce nouveau langage, avec lequel les scripts seront écrits, est nécessaire pour diverses raisons. *SableCC* est un outil portable d'un système d'exploitation à un autre. Les scripts doivent avoir la même flexibilité. Finalement, *Hiber* est simple et il travaille avec une structure de données complexe, ce qui permet un couplage entre le script et *SableCC*.

L'approche proposée dans ce mémoire pour permettre à *SableCC* de générer du code dans plusieurs langages se distingue des approches utilisées par d'autres générateurs de compilateurs car nous faisons une séparation entre les motifs, les données et les règles de génération. Les fichiers patrons contiennent les motifs et le script rassemble les règles de génération. De surcroît, nous utilisons une structure de données qui regroupe toutes les données requises pour la génération. Les données sont fournies à l'interpréteur par *SableCC*.

Sommairement, les contributions de ce mémoire sont :

- L'introduction d'un mécanisme permettant à un générateur de compilateur de générer du code dans plusieurs langages de programmation. Ce mécanisme permet de cibler un nouveau langage destination en fournissant une description séparée des règles de génération et des motifs de code généré. La séparation des motifs, des données et des règles apporte une meilleure lisibilité et facilite la maintenance.

- L'introduction d'une structure de données permettant d'accéder facilement aux informations tirées de la grammaire (jetons, productions, alternatives, ...).

- L'introduction d'un petit langage, *Hiber*, offrant la possibilité de décrire simplement les règles de génération de code.

- La réalisation du mécanisme proposé dans l'outil *SableCC*. Cela comprend l'écriture de la grammaire *Hiber*, l'écriture de l'interpréteur, l'adaptation à *SableCC* et la création d'une structure de données.

- L'écriture des scripts et des fichiers patrons dans trois langages orientés objets différents.

Ce mémoire est structuré de la façon suivante. Dans le chapitre I, nous commençons par rappeler les notions préliminaires sur *SableCC*. Le chapitre II introduit le langage *Hiber*. Il présente sa syntaxe ainsi qu'une vue d'ensemble de ce langage. Le chapitre III décrit en détails l'interpréteur du langage *Hiber*. Le chapitre IV introduit la structure de données utilisée lors de la génération du code. Ensuite, le chapitre V définit la construction d'un script. Le chapitre VI présente les expérimentations effectuées sur la génération de code. Le dernier chapitre compare l'approche proposée à celle qui est utilisée par d'autres générateurs de compilateurs. Enfin, nous présentons nos conclusions et offrons des orientations pour des travaux futurs.

CHAPITRE I

NOTIONS PRÉLIMINAIRES SUR *SABLECC*

Dans ce chapitre, nous abordons les notions préliminaires sur *SableCC*. Nous commençons par introduire l'outil *SableCC*. D'abord, nous expliquons les principales étapes pour la construction d'un compilateur ou d'un interpréteur. Par la suite, nous introduisons, l'analyseur lexical suivi de l'analyseur syntaxique inclus dans *SableCC*. Nous enchaînons avec une nouveauté de la version 3.0 de *SableCC* qui aboutit à un arbre syntaxique abstrait. Finalement, nous examinons la génération des classes bâties par *SableCC*. Une grande partie de ce chapitre résume la thèse de M. Étienne Gagnon sur *SableCC*, d'où beaucoup de similitudes peuvent apparaître. Pour approfondir le fonctionnement de cet outil, il est conseillé de se reporter aux mémoires (Gagnon, 1998), (Agbakpem, 2006) et au site Internet de *SableCC* (SableCC).

1.1 Introduction à *SableCC*

SableCC offre un environnement orienté objets dans le but de construire des compilateurs et des interpréteurs en *Java*. Il contient un analyseur syntaxique qui construit un arbre syntaxique abstrait (AST). Grâce au patron « visiteur », *SableCC* permet de naviguer sur un arbre syntaxique abstrait ou sur un arbre syntaxique concret (CST). La différence entre ces deux arbres est que les noeuds superficiels de l'arbre syntaxique concret (CST) n'apparaissent pas dans la structure de l'abre syntaxique abstrait (AST). Par l'environnement orienté objets, nous bénéficions de la génération d'un arbre où chaque noeud est strictement typé.

Pour utiliser *SableCC*, il faut écrire un fichier de spécifications. Ce fichier définit la grammaire avec laquelle *SableCC* génère un compilateur. Cette grammaire doit faire partie de l'ensemble des grammaires LALR(1) (Appel, 2002).

1.1.1 Les principales étapes pour construire un compilateur ou un interpréteur

Dans cette sous-section, nous définissons les principales étapes et nous donnons une brève description pour construire un compilateur ou un interpréteur en utilisant *SableCC* :

1. Créer un fichier des spécifications qui contient les définitions lexicales ainsi que la grammaire qui doit être compilée.
2. Démarrer *SableCC* avec le fichier de spécifications pour générer la structure.
3. Créer une ou plusieurs classes qui peuvent hériter des classes générées par *SableCC*.
4. Créer une classe avec la méthode `main` qui active l'analyseur lexical, l'analyseur syntaxique et l'interpréteur sur les classes. Il est possible de créer un compilateur au lieu d'un interpréteur.
5. Générer le compilateur ou l'interpréteur avec un compilateur *Java*.

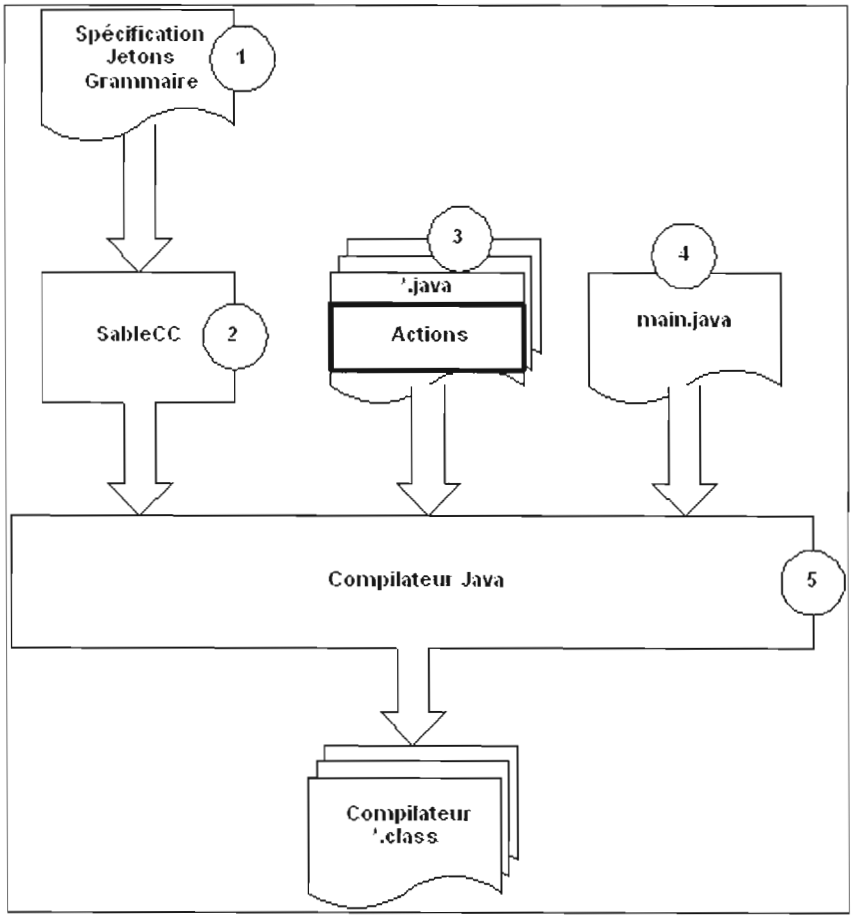

Figure 1.1 Les étapes pour créer un générateur en utilisant *SableCC* (Tirée de Gagnon 1998)

1.1.2 Le fichier de spécifications

Le fichier de spécifications est un fichier texte qui contient les définitions et les productions de la grammaire. Les définitions lexicales utilisent les expressions régulières et les définitions syntaxiques sont écrites sous la forme *Backus-Naur* (BNF) (Aho, Sethi et Ullman, 1991).

Ce fichier peut être divisé jusqu'en sept sections :

- `Package`
- `Helpers`

- `States`
- `Tokens`
- `Ignored Tokens`
- `Productions`
- `Abstract Syntaxe Tree`

Certaines de ces sections sont facultatives et d'autres sont obligatoires. Ces sections seront approfondies tout au long de ce chapitre; elles contiennent un grand nombre de propriétés importantes pour le bon fonctionnement de *SableCC*. Par exemple, la section `Package` définit le nom du paquetage où les classes générées sont référées. De plus, *SableCC* va se servir aussi de cette section pour la destination des fichiers.

1.1.3 La génération et le développement

Une fois que le fichier de spécifications est bien défini, il est possible de générer la structure de classes correspondant à la grammaire. Il suffit d'appeler la méthode principale de *SableCC* avec le fichier de spécifications en argument. *SableCC* génère quatre sous-paquetages au paquetage principal. Les paquetages sont: `lexer`, `parser`, `node` et `analysis`. Chacun contient des classes et des interfaces générées selon les spécifications.

- Le paquetage `lexer` contient les classes `Lexer` et `LexerException`. Celles-ci génèrent l'analyseur lexical et les exceptions qui sont lancées lorsque des erreurs sont identifiées dans la phase de l'analyse lexicale.
- Le paquetage `parser` contient les classes `Parser` et `ParserException`. Comme pour le paquetage `lexer`, ces classes génèrent l'analyseur syntaxique et les exceptions qui sont relevées durant l'analyse syntaxique.
- Le paquetage `node` contient toutes les classes définies dans l'arbre syntaxique abstrait. La nomenclature que *SableCC* utilise pour la génération des classes dans ce paquetage est:
  - La classe commence par un « `T` » lorsque la classe représente un jeton.
  - La classe commence par un « `P` » lorsque la classe représente une production.
  - La classe commence par un « `A` » lorsque la classe représente une alternative.

- Le paquetage `analysis` contient une interface et trois classes. Ces classes servent principalement à naviguer sur l'arbre syntaxique.

Par la suite, il est possible d'utiliser la structure de classes générée par *SableCC* pour construire un compilateur ou un interpréteur. Puisque les actions[1] sont dans des classes externes aux classes générées par *SableCC*, une modification à ces classes n'affecte pas les classes générées. Il est donc toujours possible d'ajouter et d'enlever des actions sans affecter la stabilité de la structure de classes générées.

1.2 L'analyseur lexical

Dans cette partie, nous examinons les sections importantes du fichier des spécifications pour l'analyseur lexical. Car quatre sections peuvent influencer l'analyseur lexical: les sections `Package`, `Helpers`, `States`, `Tokens`.

L'ensemble des caractères reconnus dans ce fichier est l'ensemble des caractères Unicode. Tous les caractères exprimés entre apostrophes se représentent eux-même. Les nombres décimaux représentent le code Unicode de ce caractère, puis les nombres hexadécimaux sont préfixés par `0X` ou `0x`. Si nous utilisons des ensembles de caractères, il est possible de faire des unions et des différences d'ensembles avec les opérateur + et −. De plus, il est commode de générer un ensemble qui constitue une suite ordonnée sur la base Unicode en utilisant le symbole « `..` ». Il est aussi possible d'utiliser des expressions régulières avec les opérateurs standards suivants: `[`, `]`, `+`, `−`, `*`, `?`.

La section `Package` est optionnelle. Si elle n'est pas spécifiée, la génération aura lieu dans le répertoire initial. La section `Helpers` est aussi optionnelle. Elle permet de définir des expressions régulières qui pourront être utilisées dans la section `Tokens`. Les sections `Tokens` et `Ignored Token` représentent l'énumération des jetons à considérer et des

---

[1] Le terme action représente le code associé à une production et écrit par le programmeur pour être exécuté à un moment spécifique par l'analyseur syntaxique.

jetons à ignorer par l'analyseur lexical. Dans la section `Tokens`, s'il existe une égalité sur deux jetons de la même longueur, la première déclaration a priorité.

SableCC accepte qu'un analyseur lexical puisse avoir plus d'un état. Ce qui signifie que lorsqu'il est dans un premier état, il reconnaît un certain nombre de jetons, puis lorsqu'il change d'état, il reconnaît un autre ensemble. Un exemple simple dans lequel il est possible d'appliquer ce principe est dans la grammaire pour *HyperText Markup Language* (HTML). Nous pouvons avoir un état pour le contenu des balises et un autre état pour le contenu entre les balises. La déclaration de ces états se fait dans la section `States`. La transition d'un état à un autre se réalise grâce au symbole `->`. La section `States` est utilisée seulement dans le cas où il y a plus d'un état.

Dans le but de donner une meilleur vue d'ensemble de chacune des sections, la figure suivante affiche un extrait de la grammaire de SableCC (version 2) où les sections `Package`, `Helpers`, `States`, `Tokens` et `Ignored Token` sont définies.

```
Package org.sablecc.sablecc2x;

Helpers
    lowercase = ['a' .. 'z'];
    uppercase = ['A' .. 'Z'];
    digit = ['0' .. '9'];
    tab = 9;
    cr = 13;
    lf = 10;
    eol = cr lf | cr | lf;
    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 | '_' | '$';

States
    normal,
    package;

Tokens
{package}
    pkg_id = letter (letter | digit)*;

{normal->package}
    package = 'Package';
    states = 'States';
    helpers = 'Helpers';
    tokens = 'Tokens';
    ignored = 'Ignored';
    productions = 'Productions';

{normal, package->normal}
    equal = '=';
    l_bkt = '[';
    r_bkt = ']';
    dec_char = digit+;
    blank = blank;
    comment = comment;

Ignored Tokens
    blank,
    comment;
```

Figure 1.2 Extrait de la grammaire de SableCC (version 2)

La génération des classes génère la classe `Lexer` qui se trouve dans le paquetage `lexer`. Celle-ci fournit trois méthodes publiques :

- `public Lexer(PushbackReader in)`
- `public Token peek() throws LexerException, IOException`
- `public Token next() throws LexerException, IOException`

La méthode `Lexer` représente le constructeur de la classe. Elle accepte en paramètre un argument de type `PushbackReader` du paquetage `java.io`, puis les deux dernières méthodes retournent le prochain caractère disponible sur le flot de données. Cependant, la méthode `next` avance au prochain jeton tandis que la méthode `peek` retourne le même jeton en restant à la même position.

1.3 L'analyseur syntaxique

Dans cette partie, nous examinons les sections importantes du fichier des spécifications pour l'analyseur syntaxique. Quatre sections peuvent influencer l'analyseur syntaxique: les sections `Package`, `Tokens`, `Ignored Tokens` et `Productions`.

SableCC accepte en partie la notation *Extended Backus-Naur Form* « EBNF », ce qui peut simplifier la syntaxe de la section `Productions`. Il est primordial de bien comprendre la notation que *SableCC* utilise pour identifier correctement les classes qui sont générées. Chaque production contient au moins une alternative. Celle-ci est identifiable par les possibilités que chaque production offre. Chaque nom de production et d'alternative doit être unique. Une alternative est composée de zéro ou plus d'éléments. Un élément représente, soit le nom d'un jeton « ce qui représente un symbole terminal », soit le nom d'une production « ce qui représente un symbole non terminal ». La génération des classes est effectuée dans le paquetage `node`. Le nom de la classe représentant la production est désigné par un « P » suivi de son nom « PNom », puis chaque alternative hérite de cette production. Le nom de la classe est présenté par un « A » suivi du nom de l'alternative « ANom ». Il faut noter que la classe production est une classe abstraite.

La génération des classes produit aussi la classe `Parser` qui se trouve dans le paquetage `parser`. Celle-ci fournit deux méthodes publiques :

- `public Parser(Lexer lexer)`
- `public Start parse() throws ParserException,`
  `LexerException,`
  `IOException`

La méthode `parser` représente le constructeur de cette classe. Elle reçoit en paramètre un `lexer`. Pour référer vers un arbre syntaxique abstrait, il suffit d'appeler la méthode `parse` qui retourne toujours un objet de type `Start`. Cet objet réfère au premier nœud de l'arbre syntaxique, celui-ci représente la première production de la grammaire.

Lorsque l'analyseur navigue sur l'arbre syntaxique, il est possible d'exécuter une série d'instructions, soit que nous entrons dans un noeud, soit que nous passons sur le nœud, soit que nous sortons du noeud. La nomenclature pour l'appel de ces noeuds se fait ou bien par `inANoeud`, ou bien par `caseANoeud`, ou bien par `outANoeud` dans lequel Noeud représente le type des noeuds où nous voulons exécuter les instructions. Pour exposer la syntaxe de cette nomenclature, la figure suivante montre une petite classe qui vérifie l'unicité des noms des procédures et des fonctions en passant par les nœuds `outAProcedure` et `outAFunction`.

```
package org.sablecc.sablecc.script;

import org.sablecc.sablecc.script.node.*;
import org.sablecc.sablecc.script.analysis.*;
import java.util.*;

public class VerifDeclarations extends DepthFirstAdapter {
  final HashMap functions;
  final HashMap procedures;

  public VerifDeclarations(HashMap functions,
                           HashMap procedures) {
    this.functions = functions;
    this.procedures = procedures;
  }

  public void outAProcedure(AProcedure node) {
    String procName = node.getName().getText();
    if (procedures.containsKey(procName)) {
      System.out.println("error procedure " + procName + " is
       defined ", node);
    }
    else {
      procedures.put(procName, node);
    }
  }

  public void outAFunction(AFunction node) {
    String funcName = node.getName().getText();
    if (functions.containsKey(funcName)) {
      System.out.println("error function " + funcName + " is
       defined ", node);
    }
    else {
      functions.put(funcName, node);
    }
  }
}
```

Figure 1.3 Exemple d'une classe ayant le type de noeud `outANoeud`

1.4 Arbre syntaxique abstrait

La version 3 de *SableCC* (Agbakpem, 2006) apporte une nouvelle fonctionnalité très intéressante. Il est maintenant envisageable de profiter d'un arbre syntaxique abstrait. Avec les versions précédentes, il était seulement possible de travailler avec un arbre syntaxique concret. Pour en bénéficier, il faut ajouter une nouvelle section dans le fichier de

spécifications. Celle-ci est insérée après la section `Production` et elle est nommée `Abstract Syntax Tree`. Il faut noter que cette nouvelle fonctionnalité est optionnelle. Les fichiers de spécifications de la version 2 sont toujours acceptés.

Pour utiliser un arbre syntaxique abstrait, il sera nécessaire de modifier la partie `production`. La transformation d'une production devra correspondre à une production de la section `Abstract Syntax Tree` ou à un jeton. L'ajout du symbole étoile * après un terme désigne une liste. Une transformation ressemblera à l'exemple suivant :

```
production  {-> prod_transform1 prod_transform2* ...
             prod_transformN} = element1 element2 ... elementN
            {-> alt_transform1 alt_transform2 ... alt_transformN}
```

Figure 1.4 Transformation d'une production

Sommairement, la transformation d'une alternative se fait selon les changements de la production. Le nombre d'éléments doit rester le même et les éléments doivent être du même type. Si nous examinons l'exemple de la transformation, le nombre d'éléments est resté le même et le type de chaque élément coïncide. Il existe cinq types de conversions d'une alternative que nous décrivons brièvement ci-dessous.

Le premier type est un changement vers un élément existant. Ce type sert normalement à garder les termes utiles. Par exemple, lors d'une énumération d'expressions séparées par des virgules, la virgule n'est pas très essentielle à garder car elle ne fait que grossir l'arbre.

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

Figure 1.5 Transformation d'une alternative vers un élément existant

Le second type est la création d'une nouvelle alternative. Ce type de transformation crée un nouveau nœud dans l'arbre syntaxique abstrait. Pour générer ce nouveau nœud, il faut employer le mot `New`. La figure suivante montre la création d'un nœud dans l'arbre syntaxique ayant deux enfants (`exp` et `factor.exp`).

```
exp = {plus} exp plus factor{-> New exp.plus(exp, factor.exp)};
```

Figure 1.6 Transformation d'une alternative, création d'une nouvelle alternative

Le troisième type est la création d'une liste de termes homogènes. Pour créer une telle liste, il faut mettre les éléments entre crochets.

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

Figure 1.7 Transformation d'une alternative, création d'une liste de termes homogènes

Le quatrième type est l'utilisation du terme `Null` pour éliminer un élément. Les deux règles que nous énumérons consistent à obtenir le même nombre d'éléments et à avoir des éléments de même type lors de la transformation. Le terme `Null` correspond à tous les types d'éléments et est en conformité avec un terme pour garder le même nombre d'éléments. Si nous voulons utiliser une liste vide, il faut seulement utiliser les parenthèses ouvertes et fermées.

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

Figure 1.8 Transformation d'une alternative, utilisation du terme `Null`

Le cinquième type est l'utilisation d'une transformation vide qui sert normalement à naviguer dans un sous arbre. Il existe une différence entre une transformation vide et une transformation avec le terme `Null`. Cette dernière permet de consulter le noeud même s'il contient la référence nulle. Dans le cas d'une transformation vide, il n'est pas possible de consulter le nœud.

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

Figure 1.9 Transformation d'une alternative, utilisation d'une transformation vide

La figure suivante montre un petit exemple de transformation entre la section `Production` et la section `Abstract Syntax Tree`.

```
Productions
  Script = global_declarations?
         {-> New script ([global_declarations.ident])};

  global_declarations {-> ident* } = global ident_list ?
         semi_colon {-> [ident_list.ident] };

  ident_list {-> ident* } = ident ident_list_tail*
         {-> [ident, ident_list_tail.ident] };

  ident_list_tail {-> ident} = comma ident {-> ident};

Abstract Syntax Tree
  script =
    [globals]:ident*;
```

Figure 1.10 Exemple de transformation entre la section Production et la section `Abstract Syntax Tree`

La description exhaustive de la syntaxe et des mécanismes de transformation d'arbre syntaxique est fournie dans (Agbakpem, 2006).

1.5 Génération des classes avec `MacroExpander`

Nous avons vu jusqu'à présent dans ce chapitre comment *SableCC* génère les classes en *Java*. Maintenant, nous allons examiner comment *SableCC* conçoit chaque fichier. Cet outil se sert de certains fichiers gabarits pour générer les classes. Ces fichiers sont en format texte et ils définissent le contenu des classes, des attributs et des méthodes. Ces définitions sont regroupées sous une ou plusieurs macros. Le contenu d'une macro est délimité par un en-tête sous le format de la figure suivante et par le symbole `$` en début de ligne qui identifie la fin d'une macro.

```
Macro:Nom de la macro
```

Figure 1.11 Définition d'un en-tête d'une macro

Le contenu d'une macro peut contenir des balises qui sont remplacées lors de l'appel. Ces balises sont représentées par `$X$` où `X` représente une séquence numérique commençant

par 0. Les balises peuvent se répéter. Lors de l'appel d'une macro avec la classe `MacroExpander`, il faut passer le bon nombre de balises. La génération d'une classe se fait par la concaténation du résultat d'une ou de plusieurs macros.

```
Macro:FixedTextToken

package $0$;

import $1$.*;

public final class $2$ extends Token
{
    public $2$()
    {
        super.setText("$3$");
    }

    public $2$(int line, int pos)
    {
        super.setText("$3$");
        setLine(line);
        setPos(pos);
    }

    public Object clone()
    {
      return new $2$(getLine(), getPos());
    }

    public void apply(Switch sw)
    {
        ((Analysis) sw).case$2$(this);
    }

    public void setText(String text)
    {
        throw new RuntimeException("Cannot change $2$ text.");
    }
}
$
```

Figure 1.12 Exemple de la macro `FixedTextToken` de *SableCC*

1.6 Conclusion

Finalement, dans ce chapitre, nous avons introduit l'outil *SableCC* et les principales étapes pour construire un compilateur ou un interpréteur. Nous avons ensuite fait un survol

des sections du fichier de spécifications, de la génération des classes et des étapes de développement. De surcroît, nous avons décrit la nomenclature que *SableCC* utilise et nous avons approfondi certaines classes et leurs méthodes. Nous avons survolé également le fonctionnement des analyseurs lexical et syntaxique, puis nous avons examiné la section d'arbre syntaxique abstrait ainsi que de la génération des classes.

CHAPITRE II

LANGAGE

Dans ce chapitre, nous décrivons les motivations pour la création d'un nouveau langage et nous explorons les spécifications de celui-ci. Nous expliquons ensuite les avantages d'utiliser ce nouveau langage et nous examinons les règles syntaxiques qui le gouvernent.

2.1 Grammaire *Hiber*[2]

La raison pour laquelle nous avons opté pour un nouveau langage est de permettre à tous les usagers de *SableCC* d'élaborer des spécifications en vue de générer des classes dans un autre langage que celui de Java. La grammaire *Hiber* facilite le travail de l'utilisateur par sa simplicité. Celui-ci a seulement à écrire un ou plusieurs fichiers de spécifications en tenant compte des règles syntaxiques de *Hiber* et à exécuter ce fichier dans l'interpréteur accessible par *SableCC*.

Hiber, ce nom purement inventé, est un langage de scripts qui utilise une structure de données qui sera décrite dans le chapitre « Structure de données ». Sans cette structure, *Hiber* est un langage qui n'a pas vraiment d'utilité. Cette structure représente toutes les élements de la grammaire (les jetons, les productions, ...).

---

[2] La grammaire complète de *Hiber* est fournie à l'appendice A.

Dans *Hiber*, il n'est pas nécessaire d'écrire des lignes de code spécifiques pour définir le type des variables avant de pouvoir les utiliser. Il suffit d'assigner une valeur à un nom de variable pour que celle-ci soit automatiquement créée avec le type qui correspond le mieux à la valeur fournie. Cette démarche constitue une particularité intéressante qui se retrouve aussi dans d'autres langages comme *Python* et *Lisp*. Le typage statique est préférable dans le cas des langages compilés, parce qu'il permet d'optimiser l'opération de compilation. Puisque *Hiber* n'est pas compilé, mais seulement interprété, le typage n'est pas une caractéristique primordiale. Cependant, on doit s'assurer que les actions sont valides et qu'elles s'exécutent de la façon voulue. Par exemple, le résultat de l'addition d'une valeur littérale avec un nombre est la concaténation de ces deux valeurs. En revanche, lors d'une soustraction impliquant une valeur littérale et un nombre, une erreur est levée durant l'exécution. Dans *Hiber*, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres `a-z`, `A-Z` et de chiffres `0-9`. Ce nom doit toujours commencer par une lettre minuscule.

- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$`, `#`, `@`, etc. sont interdits.

- La casse est significative.

Même si la grammaire est fondée sur la simplicité, elle permet également certaines fonctionnalités. Il est possible de créer diverses procédures et fonctions. Il est possible d'utiliser des conditions pour les opérations logiques et pour certains formats de boucle. Toutes ces caractéristiques sont abordées dans la section syntaxe de ce chapitre.

Cependant, il est important de noter que pour des raisons de simplicité et de clarté grammaticales, certains mots réservés sont gérés par l'interpréteur de la grammaire. Ces mots peuvent être utiles dans plusieurs situations. Par exemple lorsque nous voulons parcourir des listes, il est possible d'aller chercher seulement le premier élément. Toutes ces fonctionnalités sont décrites dans le chapitre de l'interpréteur.

2.2 Syntaxe

Nous allons définir la syntaxe de *Hiber*. C'est une syntaxe qui est fondée sur les usages les plus communs des langages de programmation, ce qui rend la courbe d'apprentissage de ce langage d'un niveau très facile. *Hiber* est simple grâce à son petit nombre de mots réservés. Il en comporte très peu, seulement une douzaine. Ils sont énumérés dans le tableau 2.1.

Tableau 2.1 ***Mots réservés***

`case`	`function`	`procedure`
`default`	`global`	`return`
`else`	`if`	`reversed`
`foreach`	`in`	`switch`

Les instructions doivent se terminer par un point-virgule; les blocs d'instructions sont tous délimités par des accolades ouvertes et fermées. Un bloc d'instructions comporte une ou plusieurs instructions. Qu'il s'agisse d'un élément de tabulation, d'un espace entre deux caractères ou d'un changement de ligne, les espaces sont ignorés, mais permettent de rendre le code plus lisible.

2.2.1 Commentaires

Les commentaires permettent de bien documenter le code et de rendre certaines sections plus lisibles. C'est du texte qui n'est pas traité par l'interpréteur. Il est possible d'utiliser deux types de commentaires. Le premier est symbolisé par deux barres obliques `//`. Les caractères compris entre ce signe et le changement de ligne ne sont pas interprétés. Le deuxième est représenté par le symbole barre oblique étoile `/*` et il indique à l'interpréteur qu'il ne doit pas traiter les caractères qui suivent jusqu'au symbole de fin de commentaire étoile barre oblique `*/`. Ce dernier peut s'étendre sur plusieurs lignes.

2.2.2 Variables globales

Il est possible de déclarer des variables globales. Cependant elles doivent être inscrites en début du script. Ce type de variable peut être appelée à tout moment dans le script. La syntaxe utilisée pour déclarer une variable globale est constituée du mot `global` suivi du nom de la variable. Il est possible de réserver plusieurs variables globales sur plusieurs lignes ou sur la même ligne. Dans ce dernier cas, il faut séparer les variables d'une virgule. Toutes déclarations de variables doivent se terminer par un point-virgule. La déclaration implique seulement la réservation du nom de la variable. L'affectation à cette variable est entreprise plus tard.

```
global nom1, nom2, ... ;
    ou
global nom1;
global nom2;
```

Figure 2.1 Syntaxe d'une déclaration d'une variable globale

2.2.3 Procédures et fonctions

Une procédure permet de diviser le script en blocs sans qu'elle retourne une valeur. Le script doit comporter au moins une procédure ou une fonction. Il est possible d'appeler une procédure à n'importe quel endroit dans le script. La syntaxe de la déclaration d'une procédure se fait par le mot `procedure` suivi du nom de cette procédure puis, entre parenthèses, par une série d'arguments séparés de virgules. À noter que ces arguments sont facultatifs. Le contenu de la procédure est délimité par des accolades ouvertes et fermées. L'appel d'une procédure peut être fait n'importe où dans une procédure ou dans une fonction. La syntaxe est simple : il faut appeler seulement la procédure en lui faisant suivre ses paramètres entre parenthèses, en tenant compte que les paramètres sont séparés par des virgules.

```
procedure nom([arg1, arg2, ...]) {
    // une série d'instructions
}
```

Figure 2.2 Syntaxe d'une déclaration de procédure

```
nom([arg1, arg2, ...]) ;
```

Figure 2.3 Syntaxe de l'appel d'une procédure

Une fonction permet de regrouper une série d'instructions dans un bloc; cette fonction retourne alors une valeur. La déclaration est très similaire à la procédure; la seule différence est l'utilisation de mot `function` au lieu de `procedure`. De plus, comme une fonction doit retourner une valeur, il faut, à la fin de la fonction, que l'instruction `return` soit appelée. Cette instruction est suivie de l'expression dont la valeur doit être retournée. En ce qui a trait à l'appel d'une fonction, la syntaxe est la même que celle d'une procédure. La syntaxe commence avec le nom de la fonction appelée avec ses paramètres entre parenthèses, dans lesquelles les paramètres sont toujours séparés par des virgules.

```
function nom([arg1, arg2, ...]) {
   // une série d'instructions
   return expression;
}
```

Figure 2.4 Syntaxe d'une déclaration de fonction

```
variable = nom([arg1, arg2, ...]) ;
```

Figure 2.5 Syntaxe de l'appel d'une fonction

2.2.4 L'assignation

L'instruction d'assignation peut être utilisée dans n'importe quel bloc. Cet opérateur affecte l'opérande à gauche de la valeur de celle de droite. La syntaxe de cette instruction commence par la variable où est insérée la valeur; vient ensuite le symbole d'assignation « = », puis une expression. Une expression peut être un appel de fonction, une constante, une variable, une expression arithmétique ou une expression logique. Pour assigner une valeur dans une variable globale, il faut ajouter devant la variable le mot `global` pour désigner que celle-ci est affectée. Il est important de noter qu'une variable locale peut avoir le même nom qu'une variable globale. Cependant, la variable locale aura priorité lors de l'utilisation.

```
variable = expression;
```

Figure 2.6 Syntaxe d'affectation d'une variable locale

```
global variable = expression;
```

Figure 2.7 Syntaxe d'affectation d'une variable globale

2.2.5 Opérateurs mathématiques et logiques

Il existe un certain nombre d'opérations arithmétiques et logiques. Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend des règles de priorités. La priorité usuelle des opérations arithmétiques et logiques est respectée. Il est toujours possible d'utiliser les parenthèses ouvertes et fermées pour prioriser certaines opérations.

Hiber distingue cinq opérateurs mathématiques : l'addition « + », la soustraction « - », la multiplication « * », la division « / » et le modulo « % ».

Et il existe neuf opérateurs logiques : le et « && », le ou « || », plus grand que « > », plus petit que « < », plus petit que ou égal à « <= », plus grand que ou égal à « >= », égal « == », n'égale pas « != » et la négation « ! ».

2.2.6 Boucle `foreach`

Il est possible de boucler sur une liste déjà définie. Grâce à l'instruction `foreach`, il est possible de boucler dans l'ordre de la liste ou dans l'ordre inverse. La syntaxe de cette instruction commence par le mot `foreach` suivi d'une variable qui est locale à cette boucle, et du mot réservé `in`. Par la suite, on peut optionnellement ajouter le mot `reversed` pour boucler dans l'ordre inverse de la liste. Enfin il faut désigner la liste qui sera utilisée. Les instructions sur cette boucle sont délimitées par des accolades ouvertes et fermées.

```
foreach variable in [reversed] Liste{
    // une série d'instructions
}
```

Figure 2.8 Syntaxe de la boucle `foreach`

2.2.7 Instruction Conditionnelle `if / else`

Il est possible d'utiliser l'instruction conditionnelle `if`. Cette instruction permet d'évaluer une expression booléenne. Si l'instruction est vraie, le bloc suivant est exécuté sinon le second bloc est exécuté si présent, car le second bloc est facultatif. La syntaxe de cette instruction commence par le mot `if`, puis une expression booléenne est insérée entre parenthèses. Les expressions booléennes possibles sont, soit une variable qui contient une valeur booléenne, soit une fonction qui retourne une valeur booléenne, soit une comparaison logique, puis le premier bloc est délimité par des accolades ouvertes et fermées. Si le deuxième bloc est désiré le mot `else` est utilisé suivi du deuxième bloc délimité par des accolades.

```
if(expression) {
  // une série d'instructions
}
[else {
  // une série d'instructions
} ]
```

Figure 2.9 Syntaxe de la condition `if/else`

2.2.8 Condition `switch`

Il est aussi possible d'exécuter certains blocs d'instructions particuliers selon certaines valeurs en utilisant l'instruction `switch`. La syntaxe de cette instruction commence avec le mot `switch`, puis ensuite apparaît une expression qui est évaluée entre parenthèses. Un ou plusieurs blocs `case` sont insérés entre des accolades ouvertes et fermées. Si l'expression est liée à un bloc `case`, alors l'exécution du bloc est faite. La syntaxe d'un `case` commence avec le mot `case` suivi d'une ou de plusieurs valeurs littérales. Lorsqu'il y a plus d'un littéral, une virgule est nécessaire pour les séparer, puis le

tout est suivi de deux points et d'un bloc délimité d'accolades ouvertes et fermées. Il est possible d'insérer un bloc qui est toujours exécuté dans la mesure où aucun bloc n'est inscrit. C'est la même syntaxe qu'un `case`, mais le mot `case` sera remplacé par `default`. Contrairement à *C*, *C++* et *Java*, *Hiber* ne requiert pas l'utilisation de l'instruction `break` pour freiner l'exécution du code à la fin d'un `case`.

```
switch(expression) {
     case littéral1[, littéral2,...]: {
          // une série d'instructions
     }
      [default: {
          // une série d'instructions
     }]
}
```

Figure 2.10 Syntaxe de la condition `switch`

2.3 Une vue d'ensemble globale

Pour bien saisir la facilité d'utiliser ce langage, nous examinons un exemple où il est possible de voir presque toutes les fonctionnalités offertes par *Hiber*. Puisque la boucle `foreach` prend en paramètre une liste, nous supposons dans cet exemple que la liste « `Liste` » existe. Nous pouvons constater que l'utilisation est simple et qu'il est possible de faire quelques opérations sur des variables, de faire des comparaisons et de boucler sur des listes.

```
procedure token(){

  macroFile = loadmacro(folder + "tokens.txt");

  foreach token in Tokens{
        if (PackageName == ""){
             nodePackageName = "node";
             import = "analysis";
        }
        else{
             nodePackageName = PackageName + ".node";
             import = PackageName + ".analysis";
        }

        dir = DirName + "/node";

        file = createfile(dir, token.name + ".java");

        if(token.isFixed){
             macro(file, macroFile, "FixedTextToken",
                   nodePackageName, import, token.name,
                   token.text);
        }
        else{
             macro(file, macroFile, "VariableTextToken",
                   nodePackageName, import, token.name);
        }

        close(file);
  }
}
```

Figure 2.11 Exemple de la procédure `Token` pour la génération en *Java* avec *Hiber*

2.4 Conclusion

Dans ce chapitre, nous avons vu que *Hiber* est très simple et qu'il permet d'écrire facilement des scripts. Nous avons énuméré les mots réservés et nous avons défini la syntaxe de chaque fonctionnalité de la grammaire lors de l'utilisation de ce langage.

CHAPITRE III

INTERPRÉTEUR

Dans ce chapitre, nous introduisons un interpréteur qui est construit pour la grammaire *Hiber*. Cet interpréteur permet de lire des scripts écrits en *Hiber* pour générer des compilateurs dans un langage orienté objets. Nous commençons par mettre l'accent sur les particularités de l'interpréteur en examinant les sections du générateur, des vérifications sémantiques et de l'interpréteur. Nous enchaînons avec la gestion des déclarations des variables, des procédures et des fonctions, puis nous énumérons les définitions des procédures, des fonctions fournies par l'interpréteur et des fonctionnalités sur les listes qui sont prédéfinies. Également, nous expliquons comment il est possible d'ajouter de nouvelles fonctionnalités prédéfinies dans l'interpréteur autant pour les procédures que pour les fonctions sur les listes. Finalement, nous examinons un petit exemple d'application valide pour l'interpréteur.

3.1 Les particularités de l'interpréteur

Avant d'utiliser l'interpréteur, il faut générer la structure de fichiers à l'aide de *SableCC* à partir de la grammaire *Hiber*. Ces fichiers sont des classes en *Java* qui représentent la grammaire *Hiber*. Cette série de classes est donc créée et regroupée dans des répertoires communs (`analysis`, `lexer`, `node`, `parser`). Par la suite, il est possible d'utiliser l'interpréteur. Puisque l'interpréteur est construit sous *SableCC*, il possède les mêmes normes pour les paquetages et pour les classes. Ce dernier possède ses propres classes et ses fichiers binaires `parser.dat` et `lexer.dat`.

Dans le but de permettre une plus grande robustesse, une vérification sémantique est nécessaire. Cette phase implique une vérification des déclarations des variables globales, des fonctions et des procédures. Si les noms par exemple ne sont pas bien écrits ou s'ils sont répétés, une erreur apparaîtra avant que l'interprétation ne commence. Une particularité profitable de l'interpréteur est l'ordre des déclarations des procédures et des fonctions. Cet ordre n'est pas important, ce qui implique qu'une procédure ou une fonction peut être déclarée après qu'elle soit utilisée.

Il est possible d'utiliser l'interpréteur à partir de l'outil *SableCC* puisque cet outil est généré à partir de *SableCC*. Pour faire appel à celui-ci, il faut ajouter quelques paramètres facultatifs à la ligne de commande lors de l'appel de *SableCC*. Les arguments qu'il faut joindre à la commande sont `-o` suivi du nom du fichier contenant le script. Le script est une combinaison d'instructions construit à partir du langage *Hiber* faisant appel à une structure de données fondée sur la grammaire.

Évidemment, dans l'écriture d'un script, des erreurs peuvent se glisser. S'il y a des problèmes sémantiques ou bien syntaxiques, l'interpréteur est suffisamment précis pour faire ressortir avec exactitude le problème. Il commence par donner un message d'erreur puis la ligne et la position de l'erreur dans le script.

3.1.1 Générateur

La classe `Generator` est le point d'entrée pour l'interpréteur. Cette classe est appelée lorsque l'argument `-o` est reçu lors de l'appel de *SableCC*. Elle est composée d'une seule méthode.

```
public void scriptEngine( String scriptName,
                          File pkgDir,
                          String pkgName,
                          LinkedList tokenList,
                          LinkedList productionList,
                          LinkedList stateList,
                          LinkedList reduceList,
                          HashMap constantHash,
                          HashMap parsingTable,
                          HashMap lexerTable)
```

Cette méthode reçoit dix paramètres. Tout d'abord elle reçoit le nom du script. Ce nom est l'argument qui suit le `-o` sur la ligne de commande. Par la suite, tous les autres paramètres sont obtenus à partir de la grammaire. Elle obtient le nom du répertoire où les fichiers sont créés et le nom du paquetage auquel les fichiers appartiennent. Ensuite, elle accepte quatre listes. La première liste possède tous les jetons et leurs particularités. La seconde englobe les productions avec leurs propriétés, y compris les alternatives, puis la troisième et la quatrième contiennent des états et des réductions pour la construction des classes pour l'analyseur lexical et l'analyseur syntaxique. Toutes ces listes seront approfondies dans le chapitre sur la structure de données. Ensuite, elle reçoit aussi une table de hachage qui comporte quelques constantes. Finalement, elle reçoit deux listes qui seront utiles pour construire les tables utiles à l'analyse syntaxique et lexicale.

Cette méthode accomplit deux tâches. En premier lieu, elle démarre une vérification sémantique sur le script. Celle-ci construit trois tables de hachage : une table qui contient les noms des variables globales, une autre avec les noms des fonctions, une dernière avec les noms des procédures. En second lieu, elle appelle l'interpréteur pour l'exécution du script.

3.1.2 Vérification sémantique du script

La vérification sémantique s'exécute dans la classe `VerifDeclaration`. Cette vérification navigue sur un arbre syntaxique. Si deux procédures ou deux fonctions ont le même nom, une erreur sera relevée. Le message identifie le nom de la procédure ou de la fonction avec la position dans le script, où la procédure ou la fonction est déclarée une

seconde fois. Le constructeur de cette classe est constitué de trois paramètres qui réfèrent aux trois tables de hachage pour les noms des variables globales, les noms des fonctions et les noms des procédures.

```
public VerifDeclaration(  HashMap globalVariables,
                          HashMap functions,
                          HashMap procedure)
```

3.1.3 Interpréteur

Tout comme la vérification sémantique, l'interpréteur navigue sur un arbre syntaxique. Il exécute les instructions définies dans le script dans le langage *Hiber*. Il utilise aussi une structure de données. Celle-ci est décrite dans le chapitre structure de données. Le constructeur de cette classe est constitué de douze paramètres.

```
public Interpreter(  HashMap globalVariables,
                     File pkgdir,
                     String pkgName,
                     LinkedList tokens,
                     LinkedList productions,
                     LinkedList stateLists,
                     LinkedList reduces,
                     HashMap constants,
                     HashMap functions,
                     HashMap procedures,
                     HashMap parsingTable,
                     HashMap lexerTable)
```

Ces paramètres sont en partie les mêmes que ceux que le constructeur de la classe `Generator` reçoit; il faut aussi ajouter les trois tables de hachage (`globalVariables`, `functions` et `procedures`) déclarées pour la vérification sémantique de la classe `VerifDeclaration`. Ces trois tables de hachages contiennent les noms des variables globales, les noms des fonctions et les noms des procédures venant du script.

L'interpréteur est construit en *Java*, en conséquence les données sont stockées dans des objets *Java*. L'assignation des données se fait donc dans certains objets spécifiques. Lorsque l'interpréteur aura une expression de type littéral à mettre en mémoire, il utilisera alors un objet `String`.

Toutes les opérations arithmétiques, de comparaisons et de logique telles que l'addition « + », la soustraction « – », la multiplication « * », la division « / », le modulo « % », le et « && », le ou « || », le plus petit que « < », le plus petit que ou égal « <= », le plus grand que « > », le plus grand que ou égal « >= », l'assignation « = », l'égal « == », la négation « ! » ainsi que le n'égale pas « != » sont implémentées dans l'interpréteur puisque la grammaire les définit. Pour permettre une plus grande utilisation, l'objet `BigInteger` de *Java* est utilisé lors de l'exécution des opérations arithmétiques. L'objet `Boolean` est utilisé lors des opérations de comparaisons et de logique, puis l'opérateur d'addition peut servir comme opérateur de concaténation entre deux littéraux ou bien entre un littéral et un nombre. Si le but est de faire une concaténation de deux nombres, alors il faut convertir ces deux nombres en littéraux, puis ensuite faire l'enchaînement.

3.2 Gestion des déclarations

Nous approfondissons dans cette section la méthode utilisée par l'interpréteur de *Hiber* pour conserver chaque variable globale ou locale, chaque fonction et chaque procédure. Avant que l'interpréteur ne soit appelé, une vérification sémantique est faite sur le script. Cette étape remplit une table de hachage. Par la suite, cette table est passée en paramètre à l'interpréteur, celui-ci conséquemment peut s'en servir tout au long de son traitement. Cependant, la gestion des variables locales se fait seulement dans l'interpréteur. Nous observons cette gestion à la prochaine sous-section. Par la suite, nous examinons la gestion des procédures et des fonctions dans les sous-sections suivantes.

3.2.1 Gestion des variables

Les variables globales et locales sont insérées respectivement dans leur propre table de hachage. Les variables locales sont les seules qui ne sont pas insérées dans la table de hachage lors de la vérification sémantique. Elles sont insérées durant l'exécution de l'interpréteur. Grâce à ces tables, il est possible d'accéder rapidement aux données. Les

validations pour vérifier si les données existent ou bien pour insérer ces mêmes données se font donc trivialement.

Il n'est toutefois pas possible de détruire une variable locale; elle est retirée lorsqu'elle ne fait plus partie de la procédure ou de la fonction où elle est déclarée. En résumé, le contenu de la variable est accessible durant l'exécution de la procédure ou de la fonction à laquelle elle appartient. Si elle est déclarée dans une boucle `foreach` ou une condition `if`, il est possible d'aller chercher sa valeur tant que le bloc d'exécution de la procédure ou de la fonction n'est pas terminé. Les variables globales restent actives durant tout le script. Cependant, il est possible de remplacer le contenu de la variable autant de fois qu'on désire. Toutefois, lorsqu'une variable n'est plus active, il est possible de déclarer une variable ayant le même nom. De surcroît, il n'est pas envisageable d'avoir une variable locale et globale disposant du même nom puisque l'interpréteur parcourt la table de hachage des variables locales en premier. Subséquemment, il parcourt la table des variables globales. Aussi la casse du nom de la variable est très importante. Par exemple, la variable nommée `nomVariable` n'est pas la même que `nomvariable`.

La déclaration d'une variable globale permet seulement d'introduire son nom comme clé dans la table; par la suite, la valeur attribuée à cette clé sera nulle. Ensuite, lorsqu'une affectation est appelée pour cette clé, la nouvelle valeur prendra place. L'insertion d'une variable locale dans sa table de hachage se fait en une étape. La clé est ajoutée en même temps que sa valeur. La clé correspond toujours au nom de la variable.

3.2.2 Gestion des procédures et des fonctions

Les procédures et les fonctions sont insérées, tout comme les variables, dans leur propre table de hachage. La case de leur nom est tout aussi importante. Il ne peut y avoir deux procédures ou deux fonctions avec le même nom. Cependant, il pourrait y avoir une procédure et une fonction avec le même nom. Cette insertion se fait comme les variables globales, dans la vérification sémantique.

3.3 Les fonctions prédéfinies

Dans cette section, nous examinons les fonctionnalités de l'interpréteur. Après cela, dans ce chapitre, nous montrons l'endroit, dans l'interpréteur, où il est possible d'ajouter de nouvelles fonctions. L'ajout d'une nouvelle fonction se fait de façon pratiquement élémentaire. Puisque nous voulons avoir une grammaire abrégée et commode à utiliser, nous regardons dans cette section plusieurs fonctionnalités et certains mots qui sont réservés en ce qui concerne l'interpréteur.

L'interpréteur possède un certain nombre de fonctions et de procédures prédéfinies. Celles-ci ont l'avantage de faciliter la conception du script. De plus, l'interpréteur doit recevoir une collection de jetons, de productions, de transitions, d'états et de réductions pour rendre certaines fonctionnalités opérationnelles. Dans les sous-sections suivantes, nous énumérons les fonctions définies par l'interpréteur.

3.3.1 `createfile`

Cette fonction permet de créer virtuellement un fichier en écriture. C'est un objet `BufferedWriter` qui est créé dans l'interpréteur. Cet objet est assigné dans la variable définie pour cette fonction. Il faut noter que cette fonction reçoit deux paramètres: le répertoire où le fichier doit être créé et le nom de celui-ci. De plus, pour rendre cette création de façon concrète, la procédure `close` doit être appelée pour ajouter ce fichier physiquement.

```
fileParser = createfile(dir, "Parser.java");
```

Figure 3.1 Exemple d'appel de la fonction `createfile`

3.3.2 `loadmacro`

Cette fonction permet de définir un objet `MacroExpander`, appartenant déjà à *SableCC*, sur un fichier qui est ouvert en lecture. Cet objet permet de lire un bloc de caractères et de remplacer du contenu dans celles-ci. Celui-ci est passé en paramètre, soit par un littéral, soit par une variable. Cet objet est déterminé dans la variable assignée par cette fonction.

```
macroFile = loadmacro("parser.txt");
```

Figure 3.2 Exemple d'appel de la fonction `loadmacro`

3.3.3 `substring`

Cette fonction permet de retourner une chaîne de caractères identifiée dans une variable. De ce fait, cette fonction doit recevoir trois paramètres : une variable, la position où la chaîne doit commencer et la longueur de la chaîne résultante. Il est important de noter que si la longueur définie est supérieure à la longueur de la chaîne initiale, il y aura une erreur à l'exécution.

```
resultat = substring(term.name, 0, 1);
```

Figure 3.3 Exemple d'appel de la fonction `substring`

3.3.4 `toLowerCase`

Cette fonction permet de mettre en minuscules les caractères alphabétiques du contenu d'une variable. Le premier argument de cette fonction est la variable, ensuite les deux autres arguments sont la position de départ et d'arrivée pour la transformation des caractères. Dans l'exemple suivant, le premier caractère de la valeur de `term.name` est mis en minuscule.

```
nodeName = toLowerCase(term.name, 0, 1);
```

Figure 3.4 Exemple de la fonction `toLowerCase`

3.3.5 `toNumber`

Cette fonction permet de convertir un nombre défini dans une variable littérale en format numérique. Ce nombre est inséré dans un objet `BigInteger`. Il est important de noter que si la variable littérale contient un ou des caractères non numériques, il y aura une erreur à l'exécution.

```
resultat = toNumber(term.position);
```

Figure 3.5 Exemple d'appel de la fonction `toNumber`

3.3.6 `toUpperCase`

Cette fonction permet de mettre en majuscules les caractères alphabétiques du contenu d'une variable. Le premier argument défini est la variable, par la suite, les deux autres arguments sont la position de départ et d'arrivée pour la transformation des caractères. Dans l'exemple suivant, le premier caractère de la valeur de `term.name` est mis en majuscule.

```
nodeName = toUpperCase(term.name, 0, 1);
```

Figure 3.6 Exemple d'appel de la fonction `toUpperCase`

3.3.7 `translate`

Cette fonction permet de remplacer un caractère par un autre caractère. De ce fait, cette fonction doit recevoir trois paramètres : une variable dans laquelle les caractères sont remplacés, le caractère qui est remplacé et le nouveau caractère.

```
resultat = translate(term.nam, "P", "T");
```

Figure 3.7 Exemple d'appel de la fonction `translate`

3.4 Les fonctions prédéfinies sur les listes

Nous avons exposé, dans la section précédente, que plusieurs fonctions existent pour donner certaines fonctionnalités au script. Nous constatons maintenant qu'une série de fonctions existe aussi pour exploiter les listes. Ces fonctions donnent une plus grande facilité à l'accès aux données.

3.4.1 count

Cette fonction retourne le nombre d'éléments faisant partie de la liste.

```
if(alternative.terms.count() > 0){
    //Une série d'instructions
}
```

Figure 3.8 Exemple d'appel de la fonction `count`

3.4.2 first

Cette fonction retourne le premier élément faisant partie de la liste.

```
firstNodeName = toLowerCase(Productions.first.name, 0, 1);
```

Figure 3.9 Exemple d'appel de la fonction `first`

3.4.3 last

Cette fonction retourne le dernier élément faisant partie de la liste.

```
lastNodeName = toLowerCase(Productions.last.name, 0, 1);
```

Figure 3.10 Exemple d'appel de la fonction `last`

3.5 Les procédures prédéfinies

L'interpréteur fournit un certain nombre de procédures prédéfinies. Celles-ci ont l'avantage, tout comme les fonctions prédéfinies, de faciliter la conception du script. Dans les sous-sections suivantes, nous énumérons les procédures définies par l'interpréteur.

3.5.1 close

La procédure `close` ferme le fichier qui est passé en paramètre. Cette procédure crée le fichier sur le plan physique. L'identificateur qui est passé en paramètre est défini comme un objet `BufferedWriter` dans l'interpréteur.

```
close(fileParser);
```

Figure 3.11 Exemple d'appel de la procédure *close*

3.5.2 macro

Cette procédure appelle la classe MacroExpander en passant par une série d'arguments. Cette procédure écrit, dans un fichier de sortie (premier argument), un bloc résultant d'un fichier pris en lecture (deuxième argument). La variable du fichier de sortie est le résultat de la fonction createfile et la variable du fichier en lecture est le résultat de la fonction loadMacro. Le bloc qui est pris en charge dans le fichier en lecture pour être écrit dans le fichier de sortie est celui décrit par le troisième argument. Par la suite, les arguments suivants remplacent certaines variables dans le bloc du troisième argument. Le nombre d'arguments de remplacement est variable, possiblement zéro. Il varie selon la macro exécutée.

```
fileParser = createfile(dir, "Parser.java");
macroFile = loadmacro("parser.txt");
macro(fileParser, macroFile, "ParserNewBodyDecl", node.current);
```

Figure 3.12 Exemple d'appel de la procédure macro

De surcroît, il est possible d'insérer une ligne vide dans le fichier de sortie en utilisant le nom de macro NewLine. Sinon, on peut créer un bloc qui comporte une ligne vide.

3.5.3 main

Si nous examinons la grammaire à fond, nous pourrons remarquer que celle-ci oblige d'avoir une procédure ou une fonction. Cependant, l'interpréteur impose d'avoir une procédure appelée main. Cette procédure particulière peut être définie à n'importe quel endroit dans le fichier de spécifications. Néanmoins, elle doit exister, puisque c'est le point d'entrée dans le script.

```
procedure main(){
      //Une série d'instructions
}
```

Figure 3.13 Exemple de déclaration de la procédure `main`

3.6 Ajout d'une fonctionnalité

Avec cette mise en œuvre utilisant *SableCC*, on peut facilement ajouter de nouvelles instructions à *Hiber*. Il existe trois types d'ajouts, soit pour une procédure, soit pour une fonction, soit pour une fonction sur les listes. Dans les sous-sections suivantes, nous définissons chaque emplacement où il est envisageable de faire ces ajouts.

3.6.1 Ajout d'une procédure

L'interpréteur reconnaît certaines procédures comme `macro` et `close`. Il est possible d'en ajouter d'autres qui peuvent être utiles et faciliter le développement. Pour introduire une procédure, il faut aller dans la classe `Interpreter` et insérer le code dans la méthode :

```
public void caseACallStmt(ACallStmt node)
```

C'est au cours de cette méthode que presque toutes les procédures sont définies. La seule procédure qui n'est pas définie dans cette méthode est la procédure `main`. Celle-ci est définie au cours de la méthode :

```
public void caseAScript(AScript node)
```

Puisque c'est la première méthode exécutée et que la procédure `main` est le point d'entrée dans le script, il est primordial d'insérer cette définition à cet endroit.

3.6.2 Ajout d'une fonction

Introduire des nouvelles fonctions dans l'interpréteur est sans doute la fonctionnalité la plus profitable puisqu'il est facile d'imaginer plusieurs fonctions qui ne sont pas offertes pour l'instant par cet interpréteur, mais qui peuvent être utiles. Par exemple, il est facile d'ajouter

une fonction pour renverser une sous-chaîne d'une variable, ou bien une autre fonction qui donne la longueur d'une variable. Tous les ajouts se font dans la classe `Interpreter` dans la méthode :

```
public void outACallExp(ACallExp node)
```

C'est au cours de cette méthode que toutes les fonctions sont définies.

3.6.3 Ajout d'une fonction sur une liste

Similaire aux fonctions, il peut être intéressant d'ajouter certaines fonctions sur les listes pour en tirer plus d'avantages. Il est nécessaire d'insérer le code dans `Interpreter`. Cependant, il faut l'ajouter dans la méthode :

```
public void caseADotExp(ADotExp node)
```

Nonobstant, cette méthode peut être utilisée pour un autre usage que les listes. Il faut donc vérifier si la valeur du nœud est une instance d'une liste chaînée. Si c'est le cas, nous serons dans l'état d'une liste. Autrement, cette méthode peut servir, par exemple, à aller chercher la valeur d'un attribut d'un élément d'une liste. Cette partie est détaillée dans le chapitre sur la structure de données.

3.7 Application globale avec *Hiber*

Dans cette section, nous exposons un petit script construit dans le langage *Hiber* avec les fonctionnalités reconnues par l'interpréteur. Ce script commence par l'utilisation de la fonction `loadmacro`. Il charge dans la variable `macroFile` le fichier `parser.txt`.

Ensuite, il vérifie si la constante `PackageName`[3] est vide. Si c'est le cas, la variable `nodePackageName` prend la valeur littérale « `parser` ». Sinon, elle recueille la valeur « `.parser` ».

---

[3] La constante `PackageName` est expliquée en détails dans le chapitre sur la structure de données.

Enfin, nous assignons à la variable `dir` le nom d'un répertoire par la concaténation de la constante `DirName`[4] avec la valeur littérale « `/parser` ». Une fois cette instruction terminée, la fonction prédéfinie `createfile` est appelée. Celle-ci accepte deux paramètres. En premier lieu, un nom de répertoire est défini où nous lui assignons la variable `dir`. En deuxième lieu, le nom du fichier doit être créé. Dans ce cas, nous lui donnons la valeur littérale `"Parser.java"`. L'instance de fichier est dans la variable `fileParser`.

En dernier lieu, nous bouclons sur une liste nommée `Reduces`[5]. Dans cette boucle, nous exécutons une seule instruction. Celle-ci appelle la procédure prédéfinie macro en lui passant six paramètres. Le premier paramètre est l'instance d'un fichier où cette procédure doit écrire. Dans notre cas, nous assignons `fileParser`. Le deuxième identifie un endroit où il est possible d'accéder au fichier contenant les macros. Celui-ci est chargé dans la variable `macroFile`. Le troisième doit être le nom d'une macro existante. Puis, les trois derniers sont des valeurs venant de la liste `Reduces`. Il est intéressant de porter attention au quatrième paramètre d'où la fonction prédéfinie `first` est utilisée pour aller chercher le premier élément de la liste `Reduces`.

---

[4] La constante `DirName` est expliquée en détails dans le chapitre sur la structure de données.

[5] La liste `Reduces` est expliquée en détails dans le chapitre sur la structure de données.

```
procedure parser(){
   macroFile = loadmacro("parser.txt");
   if (PackageName == ""){
         nodePackageName = "parser";
   }
   else{
         nodePackageName = PackageName + ".parser";
   }
   dir = DirName + "/parser";
   fileParser = createfile(dir, "Parser.java");
   foreach reduce in Reduces{
         macro(fileParser,
               macroFile,
               "ParserNoInliningReduce",
               reduce.first.index,
               reduce.productionNameIndex,
               reduce.startsWith,
               reduce.productionName);
   }
   close(fileParser);
}
```

Figure 3.14 Exemple de script qui utilise les fonctionnalités prédéfinies

3.8 Conclusion

Ce chapitre a commenté la grammaire *Hiber* avec son interpréteur intégré dans *SableCC*, en mettant l'accent sur les particularités de l'interpréteur et en examinant les sections sur le générateur, sur les vérifications sémantiques et sur l'interpréteur. Nous avons enchaîné, par la suite, avec la gestion des déclarations des variables, la gestion des fonctions et la gestion des procédures. Après cela, nous avons présenté les procédures et les fonctions qui sont prédéfinies. Finalement, nous avons expliqué comment il est possible d'ajouter de nouvelles fonctionnalités prédéfinies dans l'interpréteur pour les procédures, les fonctions

fournies par l'interpréteur et par les fonctionnalités sur les listes. Nous avons terminé ce chapitre en examinant un petit exemple d'application valide pour l'interpréteur.

CHAPITRE IV

STRUCTURE DE DONNÉES

Ce chapitre définit la structure de données fournie par *SableCC*. Cette structure permet de naviguer sur toutes les données nécessaires pour la génération des classes. Nous commençons par définir le squelette de cette structure. Nous enchaînons avec la génération et l'accès aux données. Par la suite, nous décortiquons en profondeur chaque partie de la structure de données dans l'ordre suivant : la table des constantes, les jetons appelés dans la structure `Tokens`, les productions, la liste des états appelée `StateLists`, les réductions appelées `Reduces`, la table des données lexicales appelée `LexerTable` et la table des données syntaxiques appelée `ParsingTable`. Avant de conclure, nous insistons sur la génération avec l'aide de la structure de données.

4.1 Structure

Pour pouvoir bien générer le code dans le langage destination choisi, il faut disposer de toutes les informations nécessaires. Puisque chaque grammaire a une structure unique, il est primordial de permettre une grande flexibilité et un accès intuitif sur chacune des composantes de la grammaire (jetons, productions, ...). Comme il est expliqué, les données sont insérées dans une structure où il est facile de naviguer et d'accéder à l'information voulue.

La structure de données que nous définissons contient toutes les informations nécessaires pour la génération des classes. Elle est conçue pour être simple à utiliser et facile à comprendre. Chaque élément est inséré dans cette structure au moins une fois. Lorsqu'il est présent plus d'une fois, c'est pour diminuer et faciliter le traitement dans le script. Dans

certaines situations, un élément peut être entièrement formaté en minuscule tandis qu'à d'autres moments, le premier caractère est en majuscule. Bien entendu, l'élément est dupliqué dans la structure lorsque son utilité peut avoir des effets de simplification dans le script. Sinon, il est possible de modifier cet élément avec les fonctions prédéfinies par l'interpréteur.

4.2 Génération de la structure de données

La génération de la structure de données se fait à partir de la classe *SableCC*. Celle-ci est générée lorsque *SableCC* est appelé avec l'argument `-o` suivi d'un argument correspondant à un fichier contenant un script utilisant le langage *Hiber*. Il construit une structure de données selon la grammaire qui est passée en paramètre lors de cet appel. Par la suite, cette structure de données est passée à l'interpréteur pour être utilisée avec le script.

La construction de la structure de données ressemble étroitement à la génération des classes par *SableCC*. Plusieurs classes qui servent à la génération des classes sont dupliquées et adaptées pour remplir les listes de la structure de données. *SableCC* est en mesure de construire six listes qui sont générées. Voici les différentes listes générées :

- Une liste pour les jetons;
- Une liste pour les productions;
- Une liste pour les états;
- Une liste pour les réductions;
- Une liste pour la table de transitions syntaxiques;
- Une liste pour la table de transitions lexicales.

Certaines informations ne peuvent pas être représentées dans les listes, mais elles s'avèrent tout de même nécessaires. Dans ces cas, nous avons une table de hachage dans laquelle ces données seront accessibles en tout temps. Puisque ces données ne sont jamais modifiées, nous disons que cette table contient des constantes.

4.2.1 L'accès aux données

L'accès aux données se fait via un script écrit en *Hiber*. Par exemple, pour pouvoir accéder à tous les éléments d'une liste, il faut boucler sur chaque élément de celle-ci. Ensuite, il est possible d'accéder aux valeurs d'un élément en ajoutant le nom de l'attribut désiré. Comme il a été expliqué dans le chapitre de l'interpréteur, le script permet d'utiliser un certain nombre de fonctions prédéfinies pour l'utilisation des listes. Il est alors possible d'employer ces fonctions pour accéder à certaines valeurs de la liste ou même pour modifier les valeurs. Pour accéder à chaque élément, il faut utiliser la fonction `foreach` de *Hiber*. Cette fonction permet de boucler sur une liste. À chaque itération de cette liste, il est toujours possible d'accéder aux valeurs de la variable reçue en ajoutant un point suivi du nom de l'attribut.

```
foreach token in Tokens{
        var = token.name;
}
```

Figure 4.1 Exemple d'accès à la valeur `name` de chaque élément de la liste `Tokens`

4.3 Table de constantes

Comme il a été mentionné précédemment, la table de constantes est une table de hachage. Cette table est utilisée pour les constantes `ActivateFilter` et `GrammarHasTransformations` qui servent à vérifier si la méthode `filter()` dans la classe `Parser` doit être présente et pour définir si l'arbre syntaxique abstrait a des transformations ou non. Dans ces deux cas, ces constantes sont de type booléen.

Il existe deux autres constantes qui ne font pas partie de la table de hachage. Celles-ci sont passées en paramètres au constructeur de l'interpréteur. La constante `PackageName` permet d'accéder au nom du paquetage identifié dans la grammaire. Si la grammaire fait abstraction de cette valeur, il n'y aura aucun paquetage défini lors de la génération des classes. Par la suite, la constante `DirName` permet d'accéder au nom du répertoire où les fichiers seront construits. La génération se fait à partir du répertoire d'où le script est exécuté si l'argument `[-d répertoire]` n'est pas désigné lors de l'exécution de *SableCC*. S'il existe un nom de paquetage, le répertoire sera généré selon celui-ci.

4.4 Tokens

La liste `Tokens` représente tous les jetons de la grammaire. Les attributs des éléments de cette liste sont `name` pour accéder au nom du jeton; `text` pour accéder à la valeur attribuée au nom; `isFixed` pour accéder à une valeur booléenne qui identifie si le texte est constant; `isIgnored` pour accéder à une valeur booléenne si le jeton est dans la section des jetons ignorés, puis l'attribut `index` qui permet d'accéder à la position du jeton par ordre d'apparition dans la grammaire. Il faut noter que l'index commence à zéro.

```
name = TSemicolon
text = ;
isFixed = true
isIgnored = false
index = 16
```

Figure 4.2 Exemple de valeurs des attributs d'un jeton[6]

4.4.1 Transitions

La liste `Tokens` contient également un attribut représentant une liste. Cet attribut se nomme `Transitions`. Cette liste retourne un ensemble de transitions utilisé lors de la construction de la classe `Lexer`. Cette liste de transitions existe lorsque la grammaire utilise plus d'un analyseur lexical. Chaque élément de cette liste est une table de hachage composée de trois attributs, `sourceState` qui identifie l'index de l'état de la transition; `sourceStateName` qui permet d'accéder au nom de l'état initial de la transition; `destinationState` qui accède au nom de la destination de la transition.

---

[6] Cet exemple est fondé sur la grammaire de *SableCC* 3 qui se trouve en appendice C.

```
sourceState = 0
sourceStateName = NORMAL
destinationState = NORMAL
```

Figure 4.3 Exemple de valeurs des attributs d'une transition[7]

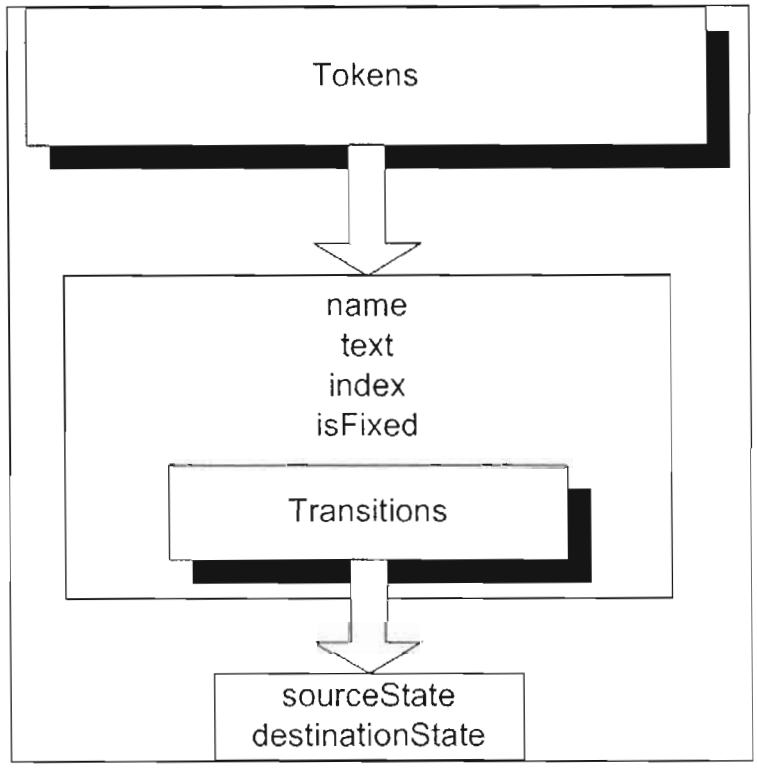

Figure 4.4 Représentation d'une liste de jetons

[7] Cet exemple est fondé sur la grammaire de *SableCC* 3 qui se trouve en appendice C.

4.5 Productions

La liste `Productions` représente toutes les productions de la grammaire. Cette liste contient l'attribut `name` pour accéder au nom de la production.

```
name = PStateList
```

Figure 4.5 Exemple de valeur de l'attribut d'une production[8]

4.5.1 alternatives

La liste `Productions` contient aussi un attribut qui représente une liste. Cet attribut est `alternatives`. Cette liste retourne l'ensemble des alternatives à l'intérieur de cette production. Chaque élément de cette liste est une table de hachage qui contient aussi l'attribut `name` permettant d'accéder au nom de l'alternative.

```
name = AStateList
```

Figure 4.6 Exemple de valeurs de l'attribut d'une alternative[9]

4.5.2 terms

La liste `alternative` a aussi un attribut qui représente une liste. Cet attribut est `terms`. Cette liste retourne tous les termes à l'intérieur de l'alternative prenant la forme d'une table de hachage. Chaque table comporte quatre attributs. Les attributs disponibles sont `name`, qui retournent le nom du terme; `type` permettant d'accéder au nom du type qui définit le terme; `isNode` retourne une valeur booléenne identifiant si le terme est un noeud dans l'arbre syntaxique, l'attribut `hasNext` retourne une valeur booléenne s'il existe un autre terme après celui-ci.

---

[8] Cet exemple est fondé sur la grammaire de *SableCC* 3 qui se trouve en appendice C.

[9] Cet exemple est fondé sur la grammaire de *SableCC* 3 qui se trouve en appendice C.

```
name = Id
type = TId
isNode = true
hasNext = true
```

Figure 4.7 Exemple de valeurs des attributs d'un terme[10]

Dans certains cas, il est pratique d'avoir une liste d'alternatives. Cependant, comme le but premier de cette structure est d'être simple et facile à naviguer, cette liste n'a pas été créée. Il est possible de simuler la liste d'alternatives avec la liste de productions. Puisque chaque production contient ses alternatives, il suffit de boucler chaque production pour retrouver toutes les alternatives.

[10] Cet exemple est fondé sur la grammaire de *SableCC* 3 qui se trouve en appendice C.

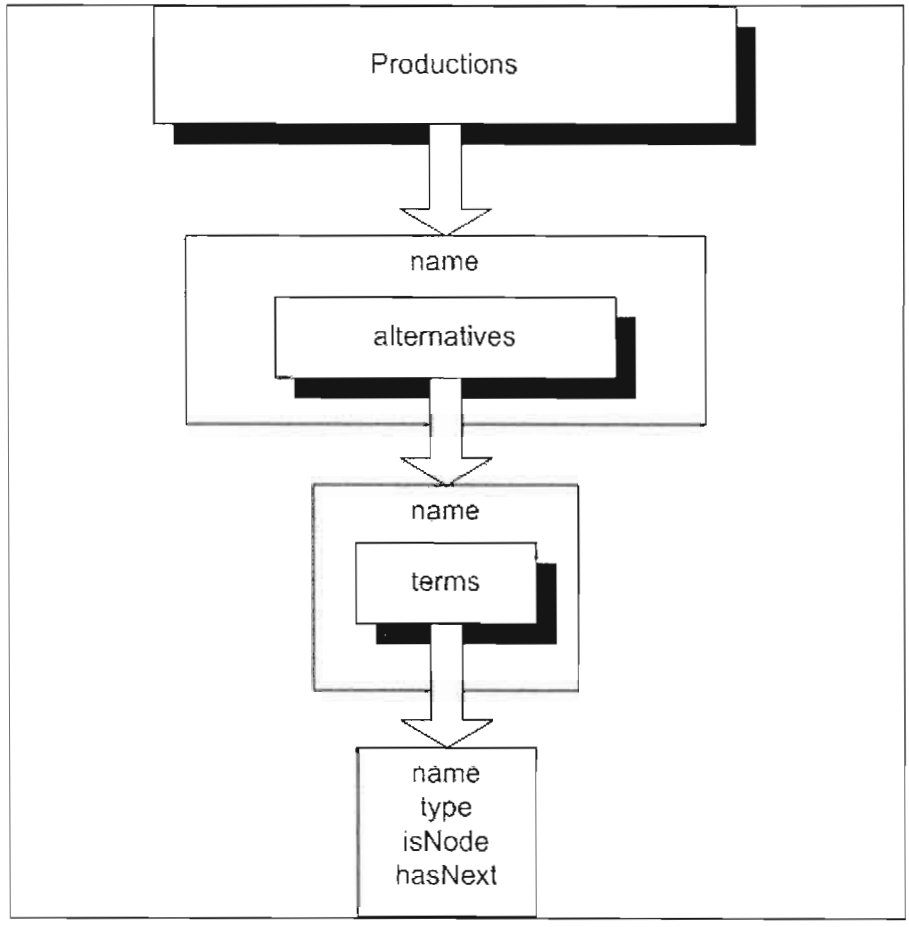

Figure 4.8 Représentation d'une liste de productions

4.6 `StateLists`

La liste `StateLists` représente toutes les transitions dans la grammaire. Ces données sont utilisées lors de la construction de la classe `Lexer`. Les attributs de cette liste sont `name` pour accéder au nom de la transition et `index` qui permet l'accès à un numéro unique propre à cette transition.

```
name = NORMAL
index = 0
```

Figure 4.9 Exemple de valeurs des attributs d'un `StateLists`[11]

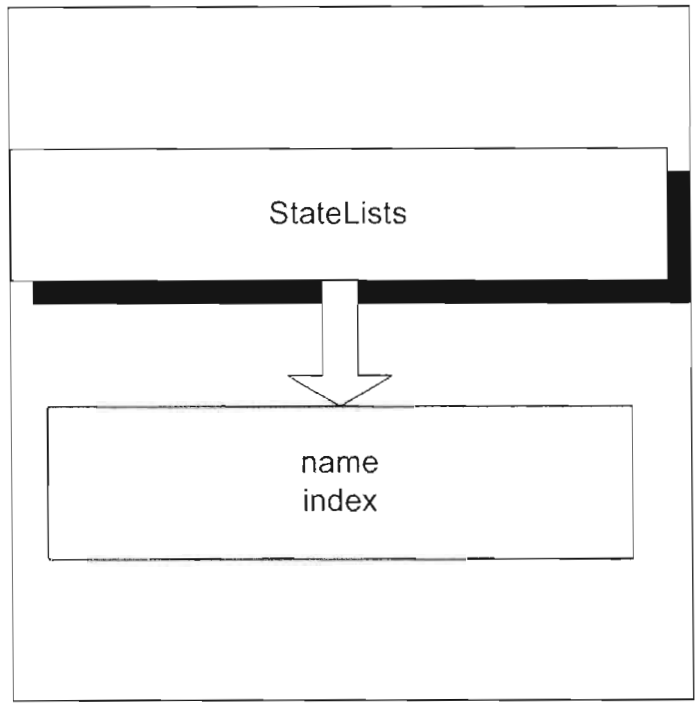

Figure 4.10 Représentation d'une liste de `StateLists`

4.7 `Reduces`

4.7.1 Introduction

Pour bien comprendre ce que la structure de données contient dans la section *Reduces*, nous allons définir un petit exemple auquel nous allons référer dans cette section. Les deux prochaines figures représentent une petite grammaire et la structure de données résultante de

[11] Cet exemple est fondé sur la grammaire de *SableCC* 3 qui se trouve en appendice C.

cette grammaire. Cette petite grammaire accepte seulement deux chaînes de caractères comportant chacune au moins un caracète et commençant par une lettre. Les caractères acceptés sont les lettres majuscules ou minuscules et les chiffres. Tous les attributs de la seconde figure seront expliqués plus loin dans cette section.

```
Package prog.sablecc.sablecc;

Helpers
  all = [0 .. 0xffff];
  letter = [['a'..'z']+['A'..'Z']];
  digit = ['0'..'9'];

Tokens
  id  = letter (letter | digit)*;

Productions
  prog = decl{->New prog(New type(decl.type), New name(decl.name))};
  decl {->[type]:id [name]:id} = [type]:id [name]:id {->type name};

Abstract Syntax Tree
  prog = type name;
  type = id;
  name = id;
```

Figure 4.11 Exemple d'une petite grammaire

```
index:0
productionNameIndex:0
productionName:AProg
startsWith:false
nodes
  nodeType:caseAElem
    current:1
  nodeType:inAAltTransform
    typeNameStartsWith:others
    typeName:PProg
    typeNameLowerCase:pprog
    position:1
  nodeType:inANewTerm
  nodeType:inAParams
    typeNameStartsWith:others
    typeName:PType
    typeNameLowerCase:ptype
    position:2
  nodeType:inAParams
    typeNameStartsWith:others
    typeName:PName
    typeNameLowerCase:pname
    position:4
```

```
    nodeType:inANewTerm
    nodeType:inAParams
      typeNameStartsWith:others
      typeName:TId
      typeNameLowerCase:tid
      position:3
    nodeType:inASimpleTerm
      typeNameLowerCase:tid
      position:3
      type:TId
      elementPosition:1
      positionMap:0
    nodeType:outANewTerm
      typeName:ptype
      newAltName:AType
      position:2
      paramTypeName:null
      paramPosition:null
      params:[{paramPosition=3, paramTypeName=tid}]
    nodeType:inANewTerm
    nodeType:inAParams
      typeNameStartsWith:others
      typeName:TId
      typeNameLowerCase:tid
      position:5
    nodeType:inASimpleTerm
      typeNameLowerCase:tid
      position:5
      type:TId
      elementPosition:1
      positionMap:1
    nodeType:outANewTerm
      typeName:pname
      newAltName:AName
      position:4
      paramTypeName:null
      paramPosition:null
      params:[{paramPosition=5, paramTypeName=tid}]
    nodeType:outANewTerm
      typeName:pprog
      newAltName:AProg
      position:1
      paramTypeName:null
      paramPosition:null
      params:[{paramPosition=2, paramTypeName=ptype},
              {paramPosition=4, paramTypeName=pname}]
    nodeType:outAAltTransform
      terms:[{typeName=pprog, position=1}]
      typeName:null
      position:null
  index:1
  productionNameIndex:1
```

```
productionName:ADecl
startsWith:false
nodes
  nodeType:caseAElem
    current:2
  nodeType:caseAElem
    current:1
  nodeType:inAAltTransform
    typeNameStartsWith:others
    typeName:TId
    typeNameLowerCase:tid
    position:1
  nodeType:inAAltTransform
    typeNameStartsWith:others
    typeName:TId
    typeNameLowerCase:tid
    position:2
  nodeType:inASimpleTerm
    typeNameLowerCase:tid
    position:1
    type:TId
    elementPosition:1
    positionMap:0
  nodeType:inASimpleTerm
    typeNameLowerCase:tid
    position:2
    type:TId
    elementPosition:2
    positionMap:0
  nodeType:outAAltTransform
    terms:[{typeName=tid, position=1},
           {typeName=tid, position=2}]
    typeName:null
    position:null
```

Figure 4.12 Section `Reduces` de la structure de données de la grammaire de la Figure 4.11

4.7.2 `Reduces`

La liste `Reduces` représente toutes les réductions de la grammaire. Cette partie apporte quelques complications car elle effectue certains traitements dynamiques lors du parcours de l'arbre syntaxique. Les attributs de cette liste sont : `index`, qui est un nombre unique et identifie chaque production; `productionNameIndex` qui permet d'accéder à la

position de la production dans la grammaire; `productionName` qui est autorisé à accéder au nom de la production générée; `startsWith` qui retourne vrai si le nom de la production `productionName` commence par le littéral non terminal « `ANonTerminal$` » plutôt que par le littéral terminal « `ATerminal$` ».

```
index = 0
productionNameIndex = 0
productionName = AProg
startsWith = false
```

Figure 4.13 Exemple de valeurs des attributs d'un `Reduces` selon la grammaire de la Figure 4.11

4.7.3 `nodes`

La liste `Reduces` contient un attribut qui représente une liste. Cet attribut est `nodes`. Cette liste retourne un ensemble de noeuds utilisé surtout par l'analyseur syntaxique. Celui-ci contient un attribut `nodeType` qui retourne un type de noeud. Chaque type de noeud a ses propres attributs.

```
nodeType = caseAElem
```

Figure 4.14 Exemple de valeurs de l'attribut d'un `nodes` selon la grammaire de la Figure 4.11

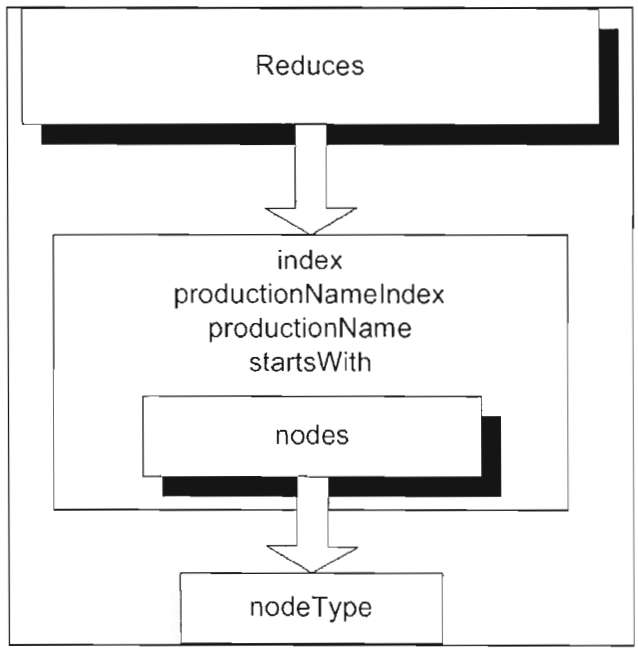

Figure 4.15 Représentation d'une liste de `Reduces`

Avant de décrire chaque type de nœuds auquel nous pouvons accéder durant la navigation, nous expliquons comment la génération fonctionne avec l'exemple de la grammaire de la Figure 4.11. Les données qui se retrouvent dans la section `Reduces` de la structure servent à la génération de la classe `Parser`.

Toutes les classes `Parser` générées par *SableCC* sont bâties sous le même format. Elles contiennent certaines méthodes communes et un certain nombre de méthodes générées de façon plus dynamique. Une méthode par alternative sera créée dans la classe `Parser` et le contenu de cette méthode dépendra de l'alternative avec laquelle elle est associée.

En observant le résultat de cette génération en *Java*, cette classe correspond à la classe `Parser.java` et aux méthodes `ArrayList newX()`, où `X` est la position de l'alternative dans la grammaire. Les alternatives sont numérotées de 0 à N, N étant la dernière alternative. Dans le cas de cette petite grammaire, il y a deux alternatives : `prog` et `decl`. La construction du premier nœud de l'alternative, dans ce contexte, se fait de la façon suivante. Une fois que l'analyseur syntaxique est arrivé à la fin de son traitement, nous avons

empilé un élément, la production `decl`. Puisque la pile de l'analyseur syntaxique ne contient que des listes, nous devons dépiler cette liste de la pile. Par la suite, il est possible de construire le nœud de type `prog`. Pour construire ce nœud, nous avons besoin de deux paramètres de type : `type` et `name`. Nous savons que ces éléments sont dans la liste que nous venons de dépiler et dans l'ordre dans lequel ils apparaissent dans la grammaire : le premier élément est `type` et le second est `name`. Une fois que ces éléments sont construits, il nous reste à construire une instance du nœud, à l'ajouter dans la liste et à retourner cette liste.

Pour la deuxième production, le déroulement de la génération ressemble à la première production. Comme mentionné dans les explications de la première production, l'analyseur syntaxique ne contient que des listes, nous devons dépiler ces listes de la pile. Cette deuxième production contient deux listes à dépiler, une pour le jeton `type` et une pour le jeton `name`. Par la suite, il est possible de construire les nœuds de type `type` et `name`. Il est possible d'aller chercher les valeurs de ces deux nœuds dans les listes que nous avons dépilées en faisant attention à l'ordre car l'élément du type `type` se retrouve dans la deuxième liste dans ce cas-ci. Puis, il ne reste qu'à construire l'instance de chaque nœud, à l'ajouter dans la liste et à retourner cette liste. La prochaine figure montre cette génération.

```
ArrayList new0() /* reduce AProg */
{
    ArrayList nodeList = new ArrayList();

    ArrayList nodeArrayList1 = (ArrayList) pop();
    PProg pprogNode1;
    {
       PType ptypeNode2;
       PName pnameNode4;
       {
             TId tidNode3;
             tidNode3 = (TId)nodeArrayList1.get(0);

             ptypeNode2 = new AType(tidNode3);
       }
       {
             TId tidNode5;
             tidNode5 = (TId)nodeArrayList1.get(1);

             pnameNode4 = new AName(tidNode5);
       }

       pprogNode1 = new AProg(ptypeNode2, pnameNode4);
    }
    nodeList.add(pprogNode1);
    return nodeList;
}

ArrayList new1() /* reduce ADecl */
{
    ArrayList nodeList = new ArrayList();

    ArrayList nodeArrayList2 = (ArrayList) pop();
    ArrayList nodeArrayList1 = (ArrayList) pop();
    TId tidNode1;
    TId tidNode2;
    tidNode1 = (TId)nodeArrayList1.get(0);
    tidNode2 = (TId)nodeArrayList2.get(0);
    nodeList.add(tidNode1);
    nodeList.add(tidNode2);
    return nodeList;
 }
```

Figure 4.16 Exemple de la construction des nœuds selon la grammaire de la Figure 4.11

Puisque chaque type de nœud a ses propres particularités, nous allons les examiner.

Le type `inAAltTransform` est utile pour retrouver le nombre de listes dans la pile de l'analyseur syntaxique. Il possède quatre attributs. L'attribut `position` retourne la position de l'alternative; `typeName` permet d'accéder au type du noeud de l'alternative; `typeNameStartsWith` permet d'indiquer le type de nœud. Par exemple, si le `typeNameStartsWith` commence par « L », celui-ci correspond à une liste et l'attribut `typeNameLowerCase` permet d'accéder au type de noeud formaté en minuscule.

```
position = 1
typeName = PProg
typeNameStartsWith = others
typeNameLowerCase = pprog
```

Figure 4.17 Exemple de valeurs de l'attribut d'un `nodeType` de type `inAAltTransform` selon l'exemple de la Figure 4.11

Le type `inAParams` est utile pour identifier les productions se retrouvant dans la liste. Il possède quatre attributs. L'attribut `position` retourne la position de l'alternative; `typeName` permet d'accéder au type du noeud de l'alternative; `typeNameStartsWith` permet d'indiquer le type de nœud. Par exemple, si le `typeNameStartsWith` commence par « L », celui-ci correspond à une liste et l'attribut `typeNameLowerCase` permet d'accéder au type de noeud formaté en minuscule.

```
position = 2
typeName = PType
typeNameStartsWith = others
typeNameLowerCase = ptype
```

Figure 4.18 Exemple de valeurs de l'attribut d'un `nodeType` de type `inAParams` selon l'exemple de la Figure 4.11

Le type `outAAltTransform` est utile pour ajouter l'élément à la liste. Il possède l'attribut `terms` qui permet d'accéder à une liste de termes. Chaque terme est une table de

hachage qui contient deux attributs : `typeName` qui permet d'accéder au type du terme et `position` qui retourne la position du terme.

```
term = {typeName=pprog, position=1}
typeName = null
position = null
```

Figure 4.19 Exemple de valeurs de l'attribut d'un `nodeType` de type `outAAltTransform` selon l'exemple de la Figure 4.11

Les types `inASimpleTerm` et `inASimpleListTerm` sont utiles pour identifier les types d'éléments se retrouvant dans la liste venant de la pile. Ils possèdent cinq attributs: `typeNameLowerCase` qui permet d'accéder au type du noeud formaté en minuscule; `position` qui retourne la position; `type` qui permet d'accéder au type du nœud; `elementPosition` qui permet d'accéder à la position exacte de l'élément dans le nœud et `positionMap` qui retourne la position lorsque les termes sont empilés.

```
typeNameLowerCase = tid
position = 3
type = TID
elementPosition = 1
positionMap = 0
```

Figure 4.20 Exemple de valeurs de l'attribut d'un `nodeType` de type `inASimpleTerm` selon l'exemple de la Figure 4.11

Les types `inANewTerm` et `inANewListTerm` n'ont aucun attribut. Cependant, ils peuvent être utiles pour faire certains traitements ou appeler certaines macros.

Le type `inAListTerm` est utile pour retrouver les termes dans une liste. Il possède l'attribut `terms` qui permet d'accéder à une liste de termes. Chaque élément de cette liste est une table de hachage qui possède quatre attributs. L'attribut `position` retourne la position du terme; `typeName` permet d'accéder au type du terme et `typeNameLowerCase` offre l'accessibilité au type du terme formaté en minuscule; quant à `typeNameStartsWith`, il peut retourner trois valeurs. Le premier retour possible est `L` si le nom du type commence par la lettre « L ». Cette valeur représente une liste, ce qui implique que `typeName` et

typeNameStartsWith seront nuls. La deuxième valeur est nulle si *typeName* est nul. Dans ce cas, la seule valeur qui est retournée est la position. La troisième valeur que peut contenir typeNameStartsWith est « other » avec des valeurs pour chaque attribut.

Le type outAListTerm permet d'ajouter un terme à la liste si terme n'est pas nul. Il possède l'attribut terms qui permet d'accéder à une liste de termes. Chaque élément de cette liste est représenté par une table de hachage pouvant contenir jusqu'à quatre attributs. L'attribut typeNameStartsWith retourne L si le nom du type commence par la lettre « L ». Cette valeur représente une liste. Si ce n'est pas une liste, la valeur vide est retournée. L'attribut listPosition retourne la position de la liste, position retourne la position du terme définissant la liste dans un tableau. L'attribut typeNameLowerCase permet d'accéder au type formaté en minuscule.

Le type outANewTerm est utilisé pour construire un nouveau nœud. Il possède quatre attributs: typeName rend possible l'accès au type; position retourne la position du terme; newAltName permet d'accéder au nom de l'alternative; params offre l'accessibilité à une liste de paramètres. Chaque élément de la liste de paramètres contient deux attributs. L'attribut paramTypeName permet d'accéder au type du paramètre et paramPosition retourne sa position.

```
typeName = ptype
position = 2
newAltName = AType
paramTypeName = null
paramPosition = null
params = {paramPosition=3, paramTypeName=tid}
```

Figure 4.21 Exemple de valeurs de l'attribut d'un nodeType de type outANewTerm selon l'exemple de la Figure 4.11

Le type outANewListTerm est utilisé pour construire un nouveau nœud. Il possède quatre attributs: typeName rend possible l'accès au type; position retourne la position du terme; newAltName permet d'accéder au nom de l'alternative; params offre l'accessibilité à une liste de paramètres. Chaque élément de la liste de paramètres contient

deux attributs. L'attribut `paramTypeName` permet d'accéder au type du paramètre et `paramPosition` retourne sa position.

Le type `caseAElem` permet d'identifier la position de l'élément dans la pile de l'analyseur syntaxique. Il possède seulement un attribut: `current` qui rend l'accès à la position de cet élément par rapport aux autres éléments. Le premier élément a forcément la position « 1 ».

```
current = 1
```

Figure 4.22 Exemple de valeurs de l'attribut d'un `nodeType` de type `caseAElem` selon l'exemple de la Figure 4.11

4.8 `LexerTable`

La liste `LexerTable` représente toutes les données nécessaires aux tables lexicales[12]. Cette liste comporte deux attributs qui représentent une liste.

4.8.1 `StatesArray`

Cette liste sert à représenter tous les états de la grammaire. Cette liste se présente sous une forme de tableau à trois dimensions. Nous avons donc en premier l'état initial. Par la suite, nous avons une autre liste nommée `States` qui contient tous les états. Enfin, pour chaque état, nous avons une liste de transitions sur chaque élément. Cette liste est nommée `Transitions`. Chaque transition comporte trois attributs : `start`, `end` et `destination`, dans lesquels ces attributs signifient l'état de départ, l'état d'arrivée et la destination de chaque transition. La figure suivante représente la méthode `gotoTable` générée en *Java* selon la grammaire de la figure 4.11. Dans cet exemple, chaque état est représenté sur une ligne. Et chaque transition comporte trois attributs. Cet exemple comporte quatre états où le premier état a deux transitions.

---

[12] Ces tables sont identifiables dans la classe `Lexer` de *SableCC*. Elles portent les noms de `gotoTable` et `accept`.

```
    private static int[][][][] gotoTable;
/*  {
        { // INITIAL
            {{65, 90, 1}, {97, 122, 1}, },
            {{48, 57, 2}, {65, 90, 3}, {97, 122, 3}, },
            {{48, 122, -3}, },
            {{48, 122, -3}, },
        }
    };*/
```

Figure 4.23 Exemple de la méthode `gotoTable` générée en *Java* selon la grammaire de la figure 4.11

4.8.2 `AcceptTables`

Cette liste sert à représenter la table des états d'acceptation. Elle contient donc une liste `States` où chaque état de cette liste contient un attribut `accept`.

```
accept = {-1, 0, 0, 0, },
```

Figure 4.24 Exemple de valeurs de l'attribut `accept` selon la grammaire de la Figure 4.11.

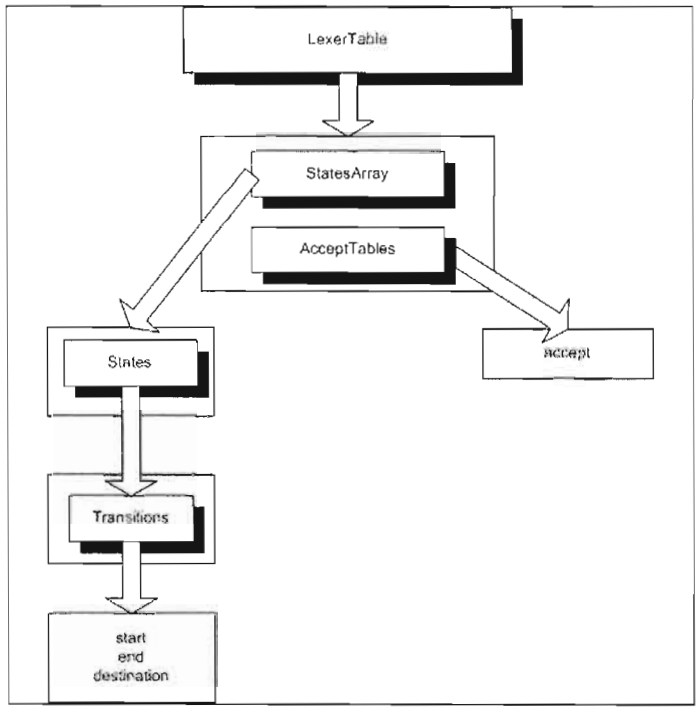

Figure 4.25 Représentation d'une liste de `LexerTable`

4.9 `ParsingTable`

La liste `ParsingTable` représente toutes les données nécessaires aux tables de l'analyseur syntaxique[13]. Cette liste comporte quatre attributs qui représentent trois listes et une table de hachage.

4.9.1 `ActionTable`

Cette liste comporte une autre liste, `listActionTable`. Cette dernière liste comporte une série de transitions. Ces transitions ont trois attributs : `from`, `actionNbr` et

---

[13] Ces tables sont identifiables dans la classe « `Parser.java` » de *SableCC*. Elles portent les noms de « `actionTable` », « `gotoTable` », « `errorMessages` » et « `errors` ».

to qui déterminent l'état de départ, l'état de la transition et l'état d'arrivée. L'état de transition peut correspondre à quatre valeurs : soit 0 pour un SHIFT, soit 1 pour un REDUCE, soit 2 pour un ACCEPT et soit 3 pour un ERROR.

```
from = -1
actionNbr = 3
to = 0
```

Figure 4.26 Exemple de valeur de l'attribut listActionTable selon la grammaire de la Figure 4.11.

4.9.2 GotoTable

Cette liste comporte une autre liste, listGoto, pour chaque non terminal. Cette dernière liste comporte une série de transitions. Ces transitions ont deux attributs : from et to qui déterminent l'état de départ et l'état d'arrivée. On désigne l'état d'arrivée le terme successeur.

```
from = -1
to = 2
```

Figure 4.27 Exemple de valeur de l'attribut listGoto selon la grammaire de la Figure 4.11.

4.9.3 ErrorMessages

Cette liste fournit tous les messages d'erreurs envisageables. Cette liste a un attribut *line* qui contient un message d'erreur.

```
line = "expecting: id",
```

Figure 4.28 Exemple de valeur de l'attribut line selon la grammaire de la Figure 4.11.

4.9.4 `Errors`

Cet attribut est stocké dans une table de hachage qui représente une série de nombres où chaque nombre est une référence pour connaître le bon message d'erreur de l'action.

```
errors = 0, 0, 1, 1, 1,
```

Figure 4.29 Exemple de valeur de l'attribut `errors` selon la grammaire de la Figure 4.11.

4.9.5 `Terminals`

Cette liste comporte deux attributs : `name` et `index`. Ces attributs représentent chaque terminal par son nom et sa position.

```
name = TId
Index = 0
```

Figure 4.30 Exemple de valeur des attributs d'un `Terminal` selon la grammaire de la Figure 4.11.

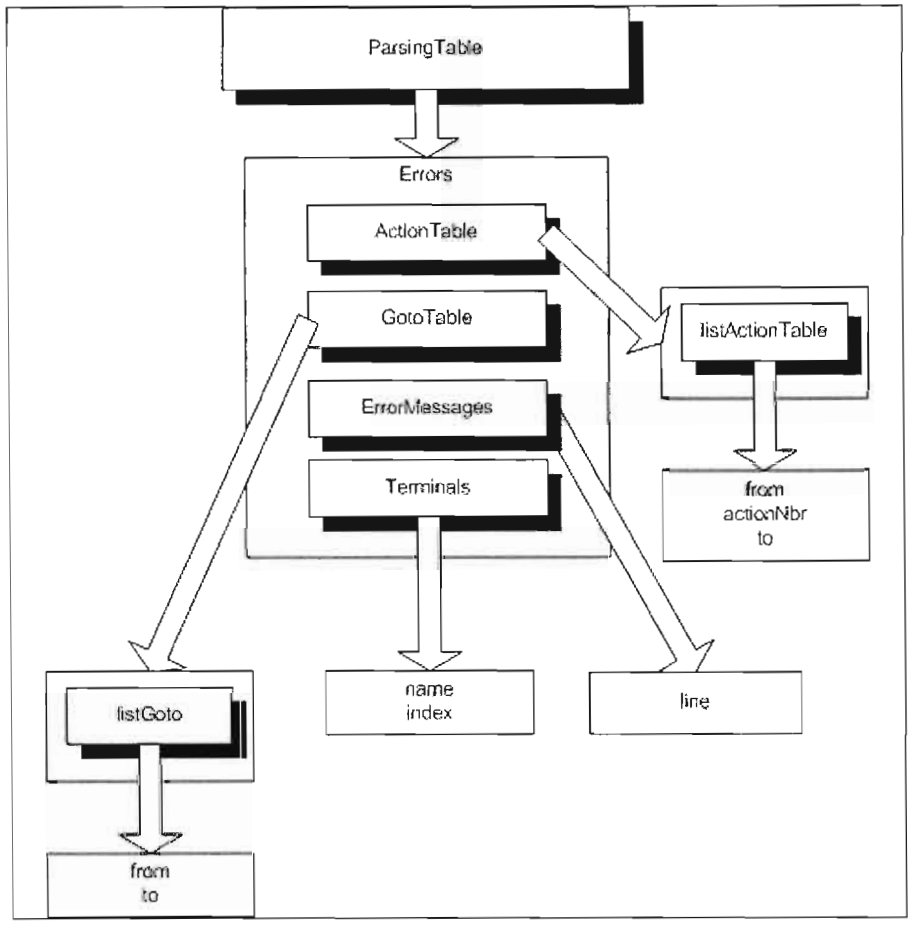

Figure 4.31 Représentation d'une liste de `ParsingTable`

4.10 Génération avec la structure de données

Pour pouvoir réaliser la génération, un ou plusieurs gabarits doivent exister. De ce fait, nous avons besoin de deux composantes pour la génération :

- Un ou plusieurs gabarits;
- Une ou plusieurs listes dans la structure de données.

Par exemple, si nous voulons générer les classes pour les jetons, nous devons utiliser un gabarit qui fournit les motifs des classes pour les jetons, puis nous utilisons la ou les listes de données dans la structure qui contient l'information sur les jetons.

Avec cette structure, il est facile de faire le tour de tous les éléments possibles en peu de temps et d'accéder aux données voulues. À l'appendice B, il est possible de voir un court exemple qui utilise cette combinaison de gabarits et de structure de données pour générer du code. Cet exemple est un extrait de la génération des classes en Java concernant la procédure `parser`. Cette procédure nous montre une boucle sur une liste avec l'instruction `foreach` et nous montre aussi une partie plus dynamique lorsque nous regardons le type du noeud avec l'instruction `switch`.

4.11 Conclusion

Ce chapitre a permis d'explorer la structure de données définie par *SableCC*. Nous avons examiné comment la structure est générée. La structure a dû être étudiée tant d'une vue globale que détaillée. Nous avons élaboré comment il est possible de générer les classes en utilisant cette structure de données.

CHAPITRE V

LE SCRIPT

Les chapitres antérieurs décrivent la grammaire *Hiber* et son interpréteur, ainsi que la structure de données qui y est générée. Rappelons qu'il est possible de se servir de cette structure avec l'interpréteur. Maintenant, nous allons examiner comment écrire un script dans le langage *Hiber* avec l'aide de cette structure de données. Pour commencer, nous allons définir la structure du script. Nous poursuivrons avec les classes à générer et nous examinerons les fichiers qui contiennent les macros. Nous continuerons avec un exemple sur la génération de code. Par la suite, nous exposerons les classes communes d'un langage à un autre. Nous enchaînerons avec les classes utilitaires et nous terminerons en exposant le contenu d'un script.

5.1 La structure du script

Avant de construire le script, il est primordial de bien comprendre ce que nous voulons générer. Il faut aussi définir un ou plusieurs fichiers servant de patrons lors de la génération de code. Ces patrons contiennent des macros représentant des motifs. Ces fichiers sont chargés en mémoire par la fonction `loadmacro` et ils sont ensuite utilisés par la procédure `macro`, puis il est important de déterminer l'ordre des macros qui doivent être appelées et les arguments que les macros doivent comporter.

Une fois que toutes ces étapes sont déterminées, il est possible de bien définir le script. Chaque script doit avoir sa procédure `main`, qui est considérée comme son point d'entrée. Il est possible de créer autant de variables globales, de procédures et de fonctions que nous en avons besoin.

Normalement, dans un script, l'ordre des instructions est représenté selon les étapes suivantes :

1. Chargement d'un ou plusieurs fichiers patrons;
2. Création d'un ou plusieurs fichiers;
3. Écriture dans un ou plusieurs fichiers;
4. Fermeture d'un ou plusieurs fichiers.

Ces étapes peuvent varier dans certaines situations. Le chargement des fichiers patrons en mémoire se fait grâce à la fonction `loadmacro`. La création des fichiers se fait à l'aide de la fonction `createfile`. Pour remplir ces fichiers, nous devons utiliser la procédure `macro`. Pour terminer, la procédure `close` doit être appelée pour enregistrer la création du fichier ainsi que tous les ajouts de code faits par les macros.

5.2 Les classes générées

Puisque nous voulons générer des classes dans plusieurs langages orientés objets, nous devons avoir la flexibilité de choisir la génération à l'emplacement voulu, avec le bon nom et le contenu souhaité. Nous considérons ces critères car l'emplacement de chaque fichier et leurs noms peuvent différer totalement d'un langage à un autre.

La première étape de la génération d'une classe consiste à définir son nom et son emplacement. Cela se fait avec la fonction prédéfinie par l'interpréteur `createfile`. Cette fonction prend deux paramètres soit le nom du répertoire où le fichier doit être créé et le nom du fichier. Ensuite, il est possible d'ajouter le contenu du fichier avec la procédure prédéfinie `macro`. Pour sauver toute la création de ce fichier, il faut alors utiliser la procédure prédéfinie `close`.

5.3 Les fichiers patrons

Les fichiers utilisés par les macros contiennent des motifs pour la génération des classes. Il est donc intéressant de bien diviser chaque partie redondante d'une classe en

macros pour qu'elles puissent être réutilisées. Toutes les parties statiques d'une classe peuvent être mises sous une ou plusieurs macros. Lorsque ces macros sont appelées, celles-ci ne reçoivent donc aucun paramètre.

Il est aussi possible d'appeler des macros qui génèrent un code plus dynamique. Ces macros reçoivent une série d'arguments. Ces arguments sont remplacés aux endroits où ils sont mentionnés dans la macro appelée. Le nombre d'arguments qu'une macro peut recevoir est illimité. Le nombre d'arguments doit correspondre entre ce que la macro s'attend de recevoir et l'appel de la macro. Si le nombre d'arguments est supérieur à celui que la macro s'attend à avoir, les derniers arguments sont ignorés. Cependant, dans le cas contraire, une erreur apparaît à l'exécution.

Le nombre de fichiers et le nombre de macros ne sont pas importants. Chaque macro doit être définie par `Macro :` suivi du nom de la macro. Lors de l'appel d'une macro, le nom de cette dernière doit exister et être unique. Pour désigner la fin de la macro, le symbole `$` doit apparaître en début de ligne. Lorsqu'une macro est dynamique, elle peut contenir des arguments qui sont énumérés par `$X$`, où `X` est la position du paramètre en commençant par zéro. Nous pouvons apercevoir, dans l'exemple suivant, la définition d'une macro pour les jetons ayant une longueur variable dans le langage *Java*. Cette macro reçoit trois paramètres.

```
// 0 : xxx.node
// 1 : xxx.analysis
// 2 : Txxx
Macro:VariableTextToken
package $0$;
import $1$.*;

public final class $2$ extends Token{
    public $2$(String text){
        setText(text);
    }

    public $2$(String text, int line, int pos){
        setText(text);
        setLine(line);
        setPos(pos);
    }

    public Object clone(){
      return new $2$(getText(), getLine(), getPos());
    }

    public void apply(Switch sw){
        ((Analysis) sw).case$2$(this);
    }
}
$
```

Figure 5.1 Macro `VariableTextToken`

5.4 La génération

Tout d'abord, il faut identifier ce que nous voulons générer. Prenons, par exemple, la génération des jetons en *Java*. Nous voulons créer une classe appelée `Token.java` où celle-ci a trois attributs soit la valeur du jeton, sa ligne et sa position. Elle a aussi les méthodes pour aller chercher et mettre à jour chaque attribut. De surcroît, cette classe hérite de la classe `Node`.

Le contenu des classes `Node` et `Token` est quasiment fixe. Les seules variations qu'elles peuvent contenir sont le nom du paquetage et les importations. Il est donc simple de créer pour chacune de ces classes une macro où la génération de ces classes est faite aisément.

Nous voulons ensuite générer une classe pour chaque jeton. Chacune de ces classes doit hériter de la classe `Token`. Aussi, ces classes ont le nom du jeton identifié dans la grammaire, précédé par la lettre « T » pour identifier que c'est une classe représentant un jeton. Le contenu de ces classes varie selon deux types de jetons. Si le jeton est de longueur fixe, la structure de la classe est déjà définie, sinon le jeton n'est pas défini dans la structure de la classe et il est géré de façon plus dynamique.

Pour générer les jetons, nous devons donc boucler sur la liste `Tokens`. La première étape est de créer le fichier avec le bon nom. Pour chaque jeton, nous devons par la suite vérifier si sa longueur est fixe avec l'attribut `isFixed`. Selon ce résultat, il faut appeler la bonne macro.

```
procedure token(){

  macroFile = loadmacro(folder + "tokens.txt");

  foreach token in Tokens{
            if (PackageName == ""){
                  nodePackageName = "node";
                  import = "analysis";
            }
            else{
                  nodePackageName = PackageName + ".node";
                  import = PackageName + ".analysis";
            }

            dir = DirName + "/node";

            file = createfile(dir, token.name + ".java");

            if(token.isFixed){
                  macro(file, macroFile, "FixedTextToken",
                  nodePackageName, import, token.name, token.text);
            }
            else{
                  macro(file, macroFile, "VariableTextToken",
                        nodePackageName, import, token.name);
            }
            close(file);
      }
}
```

Figure 5.2 Procédure de la génération de jeton en *Java*

Il est nécessaire de faire ces mêmes étapes d'analyse pour les autres classes que nous voulons générer pour les productions, les alternatives, l'analyseur lexical et l'analyseur syntaxique.

5.5 Classes communes

Lors de la génération des classes d'un langage à un autre, un grand nombre de classes sont communes. Ce que nous entendons par communes, c'est qu'elles possèdent le même nom, qu'elles ont pratiquement les mêmes attributs et les mêmes méthodes. La portabilité de l'objet est respectée. Seulement la syntaxe du langage peut apporter certaines variantes. Nous faisons abstraction de certains concepts objets. Nous assumons que chaque attribut est privé et que chaque attribut a une méthode publique pour aller chercher sa valeur et une autre méthode publique pour mettre sa valeur à jour.

5.5.1 Jetons

Il existe deux sortes de jetons soit les variables dont la valeur n'est pas fixe, soit les jetons fixes dont la valeur est définie. Dans le premier cas, cette classe possède deux constructeurs : l'un qui permet d'attribuer la valeur du jeton et l'autre qui permet d'attribuer une valeur, une ligne et une position au jeton. Dans le deuxième cas, cette classe possède aussi deux constructeurs. Cependant, ceux-ci ne reçoivent pas de valeur car elle est déjà définie.

Toutes ces classes héritent de la classe `Token`. Cette classe possède deux attributs : la ligne et la position. Ces deux attributs permettent d'identifier la position du jeton dans le code. Aussi la classe `Token` hérite-elle de la classe `Node`. Cette dernière permet d'identifier le nœud dans l'arbre et les nœuds qui sont reliés.

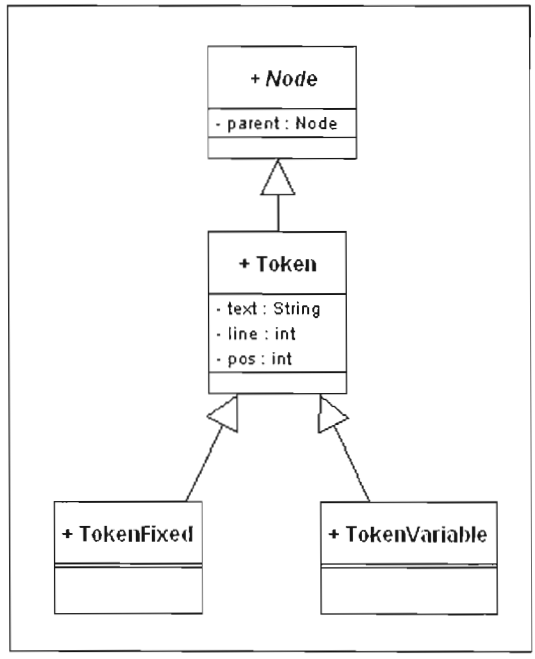

Figure 5.3 Modélisation des jetons

5.5.2 Productions et alternatives

Chaque alternative de la grammaire dispose de sa propre classe. La classe de l'alternative comprend autant d'attributs que l'alternative contient de termes. Chaque attribut peut être un terme ou bien une liste. Dans ce dernier cas, il faut user d'un moyen qui nous permet de gérer des listes selon le langage que nous voulons générer. De plus, chaque classe possède deux constructeurs : un par défaut et un autre qui permet d'initialiser chaque terme de l'alternative. Aussi chaque classe hérite-t-elle de la production à laquelle elle appartient.

Chaque production de la grammaire possède sa propre classe. Cette classe ne détient aucun attribut et elle hérite de la classe `Node`. Cette dernière classe a déjà été détaillée dans la sous-section des jetons.

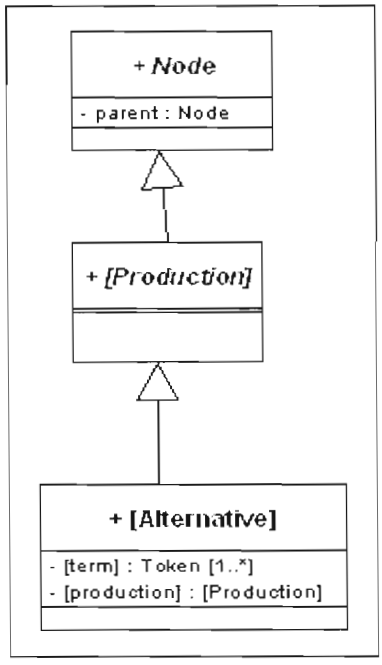

Figure 5.4 Modélisation des productions et des alternatives

5.5.3 Analyseur lexical

La section de l'analyseur lexical ne comporte principalement que deux classes : `State` et `Lexer`. Pour le cas de la première classe, celle-ci permet d'identifier dans quel état nous nous trouvons. Cette classe est composée de son constructeur qui reçoit l'état et une liste de noeuds.

Par la suite, la classe `Lexer` reconnaît tous les jetons dans le script. Autre que son constructeur, cette classe comporte principalement huit méthodes. La méthode `peek` permet d'aller chercher le jeton. Cette méthode appelle les méthodes `getToken` et `filter`. Ensuite, `next` va chercher le prochain jeton; `getChar` retourne le prochain caractère; `getText` retourne la chaîne de caractères du jeton; `pushBack` permet de reculer d'un caractère; la méthode `unread` permet de reculer d'un jeton.

Une autre classe peut être construite pour s'occuper de la gestion des erreurs. Celle-ci permet un affichage plus juste des messages d'erreurs.

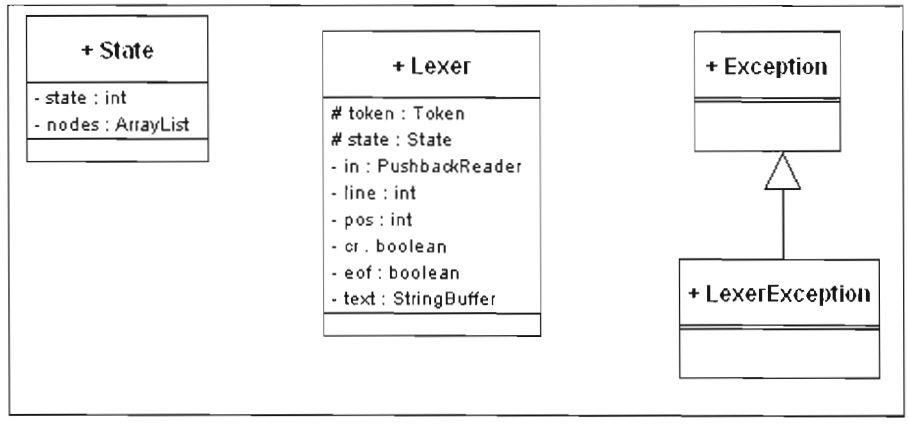

Figure 5.5 Modélisation de l'analyseur sémantique

5.5.4 Analyseur syntaxique

La section de l'analyseur syntaxique ne comporte principalement que deux classes : `State` et `Parser`. La première classe permet d'identifier dans quel état nous nous trouvons et la liste de nœuds associés. Cette classe est seulement composée d'un constructeur qui reçoit l'état et une liste de noeuds.

Par la suite, la classe `Parser` reconnaît la production et l'alternative selon la liste de tous les jetons qu'il reçoit de l'analyseur lexical. Tout d'abord, son constructeur reçoit l'analyseur lexical en paramètre. Cette classe comporte principalement six méthodes. La méthode `push` permet de déposer une liste de nœuds sur la pile; `goTo` retourne la valeur associée à la destination; `state` retourne l'état; `pop` retourne le premier élément sur la pile; `index` permet d'identifier la position du jeton dans la grammaire; la méthode `parse` est le cœur de cette classe. Cette méthode navigue selon tous les jetons qu'elle reçoit. Si la syntaxe n'est pas respectée, une erreur est déclarée pour le jeton en question. Sinon la syntaxe est correcte et cette méthode retourne un objet de type `Start`.

Une autre classe peut être construite pour gérer les erreurs. Celle-ci permet un affichage plus juste des messages d'erreurs.

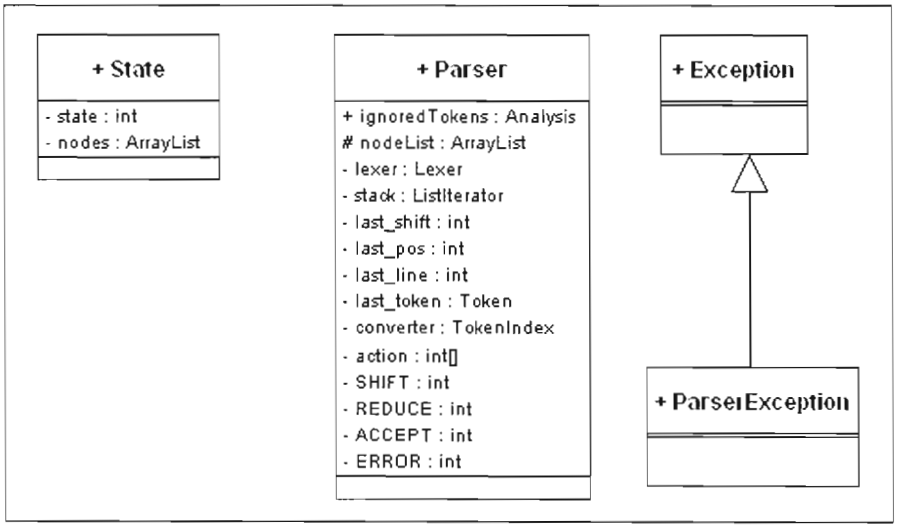

Figure 5.6 Modélisation de l'analyseur syntaxique

5.5.5 Les classes d'analyses

La section des classes d'analyse comporte principalement deux classes : `DepthFirstAdapter` et `ReversedDepthFirstAdapter`. La première permet de naviguer sur un arbre de façon à faire une recherche en profondeur. La seconde permet aussi de faire une recherche en profondeur, sauf dans l'ordre inverse. Ces classes appellent la bonne méthode correspondant au nœud et à l'état du nœud. *SableCC* permet trois états : `inNoeud` lorsqu'on entre dans le nœud; `caseNoeud` lorsqu'on est dans le nœud; `outNoeud` lorsqu'on sort du nœud. Les méthodes `inNoeud` et `outNoeud` existent, mais elles ne font rien dans cet état. Il faut les surcharger si ces états sont désirés. Cependant, dans l'état `caseNoeud`, une validation a lieu pour s'assurer que les autres nœuds enfants ne sont pas nuls. Si un traitement particulier devait avoir lieu, c'est dans l'interpréteur que ces méthodes seraient surchargées.

Ces deux classes héritent de la classe `AnalysisAdapter`. Celle-ci définit les méthodes `caseNoeud`. Cette classe hérite par la suite de l'interface `Analysis` qui oblige la déclaration des méthodes `caseNoeud`.

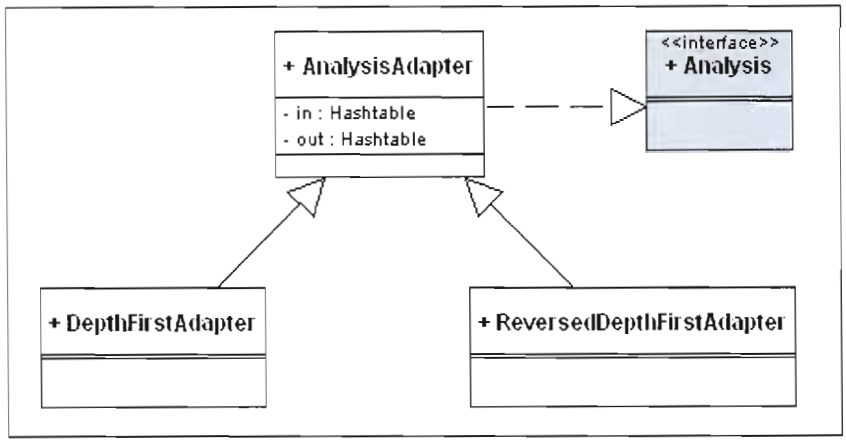

Figure 5.7 Modélisation de classes d'analyses

5.6 Classes utilitaires

Dans cette partie, nous mentionnons qu'il est toujours possible d'ajouter certaines classes pouvant faciliter le développement dans un nouveau langage. Par exemple, en *Java*, nous avons une très grande librairie assez diversifiée. Cependant, en *C++*, nous n'avons pas ces classes. Nous avons eu besoin d'ajouter en *C++* une classe représentant une liste qui est similaire à la classe `ArrayList` en *Java*.

Aussi, *SableCC* requiert deux classes : `Start` et `EOF`. Ces deux classes vont ensemble. La première classe permet d'identifier et de commencer par le premier nœud, puis `EOF` permet d'arrêter lorsque le dernier nœud est de l'instance de la classe `EOF`. Cette dernière classe est une classe qui hérite de la classe `Token`.

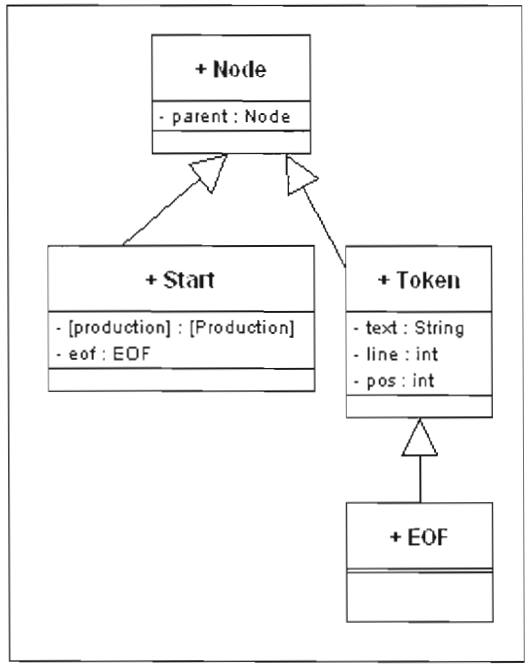

Figure 5.8 Modélisation de classes utilitaires

5.7 Création du script

Idéalement, pour bien rédiger le script, nous devons écrire chaque section précédente séparément. Bien entendu, il est possible de vérifier la construction de chaque classe d'une section lorsque celle-ci est complétée. Cependant, il n'est pas possible de s'assurer de la bonne génération tant que la majorité des sections ne sont pas construites. La seule section qui n'est pas nécessaire est celle de l'analyseur syntaxique. Lorsque l'analyseur lexical est construit, il est possible d'examiner le bon fonctionnement des classes pour les jetons, les productions, les alternatives et les classes d'analyses. Par la suite, l'analyseur syntaxique peut être ajouté.

Également, pour simplifier le script, il est avantageux au début de regrouper les constantes dans des variables globales. Le répertoire où se trouvent les fichiers patrons, les répertoires où sont créées les classes peuvent donc être dans des variables globales. Une autre façon de simplifier le code est de diviser chaque partie en procédure, étant donné que nous avons vu que la génération des classes peut se faire en plusieurs parties.

5.8 Conclusion

Dans ce chapitre, il a été question de la structure du script selon la grammaire *Hiber* et de son interpréteur. La définition de la structure du script, les classes générées, les fichiers qui contiennent les macros, l'exemple sur la génération de code, les classes communes d'un langage à un autre, les classes utilitaires et le contenu d'un script ont constitué les éléments de ce chapitre.

CHAPITRE VI

EXPÉRIMENTATION

Dans ce chapitre, nous décrivons les tests et les expérimentations utilisés afin de vérifier la mise-en-œuvre de l'implementation de Hiber et le bon fonctionnement des scripts dans les langages orienté objets. Pour commencer, nous décrivons la première génération de code faite en *Java*. Nous définissons les ajustements qui ont eu lieu sur la structure de données. Par la suite, nous introduisons une génération de code dans un deuxième langage. Ce second langage est le *C++* et nous enchaînons une fois de plus avec les ajustements dans l'interpréteur. Ensuite, nous ajoutons un troisième langage, *Python*. Nous finissons avec une génération de code élaborée dans plusieurs grammaires utilisant trois langages pour bien vérifier l'exactitude de la génération de code.

6.1 La première génération

Le premier objectif fixé est de reproduire, en *Java*, les mêmes classes que *SableCC* génère. Une fois la construction de la structure de données et de l'interpréteur complétée, il est donc possible de rédiger les fichiers patrons pour le langage *Java* et d'écrire le script. Il faut souligner que les fichiers patrons sont pratiquement identiques aux fichiers initiaux de *SableCC*. Certaines petites différences peuvent apparaître, dues à l'affichage dans les classes.

La génération de code débute par les jetons. Ceux-ci sont plutôt simples à générer et il en existe seulement deux types. Ils sont soit fixes, soit variables. Cette façon d'aborder la génération des classes permet de vérifier le bon fonctionnement de la structure de données, des procédures `macro` et `close`, des fonctions prédéfinies `create` et `loadmacro`, sans

oublier le plus important : l'interpréteur de *Hiber* dans lequel l'affectation de variables, la condition `if` et la boucle `foreach` sont toutes appelées.

Une fois que les jetons sont bien générés, c'est maintenant les productions et les alternatives qui peuvent s'enchaîner. Ces classes sont simples à produire car elles se développent sur le même principe que les jetons : elles comportent peu de variations et la génération se fait selon l'ordre d'apparition dans la grammaire. Ensuite, les classes qui favorisent la navigation de l'arbre syntaxique et les classes utilitaires sont créées. Pour terminer la génération de code, les classes pour l'analyseur lexical et syntaxique sont générées.

Une série de tests a eu lieu grâce à l'utilisation de plusieurs grammaires disponibles sur le site Internet de *SableCC*. La liste des grammaires utilisées est énumérée dans le tableau suivant. Pour chaque grammaire disponible, ces tests comportent deux générations. Une génération de la structure de classes est exécutée à partir de la version originale de *SableCC* et une génération de la structure de classes accompagnée du script. Ensuite, une comparaison des fichiers a eu lieu avec la commande `diff`. Cette commande a permis de lister toutes les différences entre deux fichiers. Dans tous les cas, les fichiers sont identiques.

Tableau 6.1 Grammaires utilisées à des fins de comparaison entre la version originale de *SableCC* et celle du script

FIPA-ACL.grammar	*Java2-1.4.grammar*	*Sablecc-2.x.grammar*
FIPA-SL.grammar	*KIF.grammar*	*sablecc-3x.sablecc3*
IDL.grammar	*KIF-pattern-matcher.grammar*	*SimpleC.grammar*
ISO-C.grammar	*mini_basic.grammar*	*SL.grammar*
Java2-1.1.grammar	*Object_Query_Language.grammar*	*SmallPascal.grammar*
Java2-1.02.grammar	*PHP4.grammar*	*XML-RPC.grammar*

Une fois que toutes les classes sont bien générées, il manque seulement deux fichiers pour être identiques à la génération que *SableCC* produit. Les fichiers binaires `lexer.dat` et `parser.dat` ne sont pas créés. Cependant, il est envisageable de copier les originaux de *SableCC* car les fichiers qui sont générés sont identiques.

6.2 Ajustement de la structure de données

Durant la première génération de code, certaines données étaient manquantes. Dans certains cas, il est intéressant d'obtenir les données formatées différemment afin d'éviter une série de transformations. Il y avait deux possibilités : soit de déterminer une fonction prédéfinie dans l'interpréteur, soit d'ajouter la nouvelle donnée dans la structure.

Le premier cas est une option très intéressante, car elle permet la réutilisation de cette fonction à plusieurs autres endroits dans le script. Dans le second cas, le formatage est déjà appliqué dans la version originale de *SableCC*. Puisqu'il reste seulement l'insertion dans la structure de données, certaines données ont été ajoutées en double dans cette même structure.

6.3 La deuxième génération

La deuxième génération de code consiste à générer la structure de fichier dans un autre langage destination. Le *C++* est désigné puisqu'il s'agit d'un langage très utilisé et que sa syntaxe ressemble étroitement à la syntaxe de *Java*. La création des fichiers patrons pour ce langage est fondée sur les fichiers patrons de l'outil *Alternative Output*[14] développé par Indrek Mare[15].

La structure du script pour le *C++* est très semblable à celle du script pour le *Java*. Toutefois, le contenu est très différent. Le nombre de macros est beaucoup plus élevé. En

---

[14] L'outil *Alternative Output* est exploré dans le chapitre 7.

[15] Il est possible d'en apprendre davantage en consultant le site Web de Indrek Mare à l'adresse suivante : http://www.mare.ee/indrek/.

C++, il y a deux types de fichiers : ceux qui contiennent les entêtes des classes « `.h` » et ceux qui contiennent le corps des classes « `.cpp` ». Une fois que les fichiers patrons et le script sont développés, la génération de code donne un résultat très similaire à celui d'Indrek. Les quelques différences sont sur le formatage des noms de variables et de méthodes.

Malgré que le *C++* soit un langage similaire à *Java*, sa complexité est beaucoup plus élevée. En *Java*, il n'y a aucune gestion de pointeurs et de mémoires, tout est transparent. De plus, la machine virtuelle a son propre ramasse-miettes pour désaffecter la mémoire. Cependant en *C++*, c'est le rôle du programmeur de gérer la mémoire. Puisque *SableCC* peut générer une structure de fichiers pour différentes grammaires, il est très compliqué de pouvoir écrire des méthodes pour désaffecter la mémoire dans diverses situations. Il n'y a donc aucune gestion de la mémoire puisqu'il y a un risque de désallouer de la mémoire à des variables pouvant être encore utilisées.

En ce qui a trait au bon fonctionnement de l'exécution, il faut exécuter et comparer les résultats. Malgré la comparaison entre les classes générées par l'outil *Alternative Output* et la génération des classes par le script, nous voulons nous assurer du résultat selon l'arbre lexical et l'arbre syntaxique. Deux nouvelles classes ont donc été construites pour faire ces comparaisons. Une en *Java* et l'autre en *C++*. Chacune doit écrire l'arbre lexical et l'arbre syntaxique dans un fichier. Par la suite, il ne reste qu'à comparer les différences des deux fichiers. La seule divergence étant qu'en *C++*, la classe `EOF` s'appelle `TEOF` car, dans le premier cas, c'est un mot réservé en *C++*.

Pour faciliter la compilation des classes, une nouvelle section a été ajoutée dans le script. Cette section sert à la création de fichiers pour la compilation des classes. Un fichier patron a été ajouté pour cette partie. Ces fichiers sont fonctionnels seulement dans un environnement linux. De plus, les programmes suivants : *Autoconf* (Autoconf) et *AutoGen* (AutoGen) doivent être installés.

6.4 Ajustement des fonctions prédéfinies

Durant la construction du script pour le *C++*, certains formatages sont nécessaires sinon certaines fonctionnalités sont manquantes. Puisque ces formatages ne sont pas déjà

présents dans la génération de code avec *SableCC* et que l'objectif de la structure de données est toujours de garder la structure de données simple et facile à utiliser, de nouvelles fonctionnalités ont été ajoutées dans l'interpréteur. Par exemple, chaque valeur dans la structure de données est présentée sous forme de type texte. La fonctionnalité `toNumber` est créée pour convertir un index d'une forme de type texte en une forme de type nombre. La fonctionnalité `substring` est ajoutée aussi pour soustraire une sous-chaîne d'une chaîne de caractères.

6.5 La troisième génération

La troisième génération consiste, tout comme celle du *C++*, à générer la structure de fichier dans un autre langage destination. *Python* a été choisi puisqu'il est un langage qui possède une simplicité d'apprentissage et dont le code est facile à lire. De surcroît, comme *Java* et *C++* ont une certaine similitude, *Python* permet de vérifier une autre syntaxe d'un langage orienté objets qui est très distincte. La création des fichiers patrons pour ce langage est fondée sur les fichiers patrons de l'outil *Alternative Output*[16]. Ces fichiers sont développés par Fidel Viegas.

Pour démontrer que la génération de code en *Python* convient parfaitement, une comparaison a été faite avec la génération de l'outil *Alternative Output*. Cette comparaison est identique. Pour augmenter le niveau de fiabilité, une nouvelle classe a été construite. Cette classe permet d'écrire l'arbre lexical et l'arbre syntaxique dans un fichier. Il est possible de comparer ces arbres avec *Java* car une classe similaire a été créée lors de la comparaison en *C++*. Chacune de ces classes doit écrire l'arbre lexical et l'arbre syntaxique dans un fichier. Par la suite, il reste simplement à comparer les différences entre ces deux fichiers pour se rendre compte qu'ils sont identiques.

Lors de la création du script pour *Python*, nous n'avons pas eu besoin d'ajouter d'éléments dans la structure de données ou de joindre de nouvelles fonctions prédéfinies dans

---

[16] L'outil *Alternative Output* est exploré dans le chapitre 7.

l'interpréteur. Tout ce dont on a besoin est préalablement présent dans l'un ou l'autre, ce qui démontre que la structure et l'interpréteur atteignent un niveau de fiabilité élevé.

6.6 Génération avec différentes grammaires

Pour s'assurer du bon résultat, la génération de code a été exécutée dans tous les trois langages suivants : *Java*, *C++* et *Python*. Chacun de ces langages a été vérifié en utilisant toutes les grammaires de la table 6.1. Dans tous les cas, les arbres pour l'analyseur lexical et pour l'analyseur syntaxique sont écrits dans un fichier, puis ils sont comparés aux arbres originaux de *SableCC* lors de la génération en *Java*.

La première grammaire utilisée pour les tests est celle de *Java 2 - 1.4*[17]. Cette grammaire permet de vérifier plusieurs points grâce à sa complexité et à sa grosseur. Plusieurs centaines de classes sont créées en utilisant cette grammaire. Elles sont toutes séparées dans plusieurs fichiers ou bien elles se trouvent dans un seul fichier. Dans le dernier cas, il est possible de constater que, malgré un très grand nombre de lignes dans un fichier, la génération de code fonctionne toujours très bien.

La deuxième grammaire utilisée est celle de *SableCC 3*[18]. Cette grammaire permet de travailler avec une grammaire plus ou moins complexe. Cependant, elle autorise l'utilisation des transitions. Cette particularité a permis de trouver certains problèmes dans la génération de code en *C++* et en *Python*. Ces problèmes sont maintenant solutionnés.

La troisième grammaire utilisée est celle de *Hiber*. Celle-ci est très simple et elle permet de démontrer que même les grammaires élémentaires peuvent bien fonctionner. Grâce à ces trois grammaires, il a été possible de montrer leurs bons fonctionnements en utilisant ces trois langages.

---

[17] Cette grammaire est disponible sur le site Internet de *SableCC* (SableCC).

[18] Cette grammaire est disponible sur le site Internet de *SableCC* (SableCC).

6.7 Conclusion

Dans ce chapitre, les tests et les expérimentations utilisés ont permis de vérifier le bon fonctionnement des scripts. Puisque *SableCC* a généré les classes en *Java*, la première génération de code est fondée sur les patrons de *SableCC*, en *Java*. Une fois la première génération de classes terminées, nous avons défini les ajustements qui ont eu lieu sur la structure de données. Par la suite, nous avons introduit un deuxième langage dans lequel il a été possible de générer le code et nous avons enchaîné en tenant compte des ajustements dans l'interpréteur. Puis, nous avons ajouté un troisième langage. Avant de conclure, nous avons abordé trois grammaires utilisées pour vérifier l'exactitude de la génération de code.

CHAPITRE VII

TRAVAUX RELIÉS

Dans ce chapitre, nous exposons d'autres outils qui ont une similitude avec l'interpréteur *Hiber* de *SableCC*. Nous examinons deux outils commerciaux : *DMS* et *Visual Parse++*, dans lesquels il n'est pas possible de consulter leur code source. Ensuite, nous enchaînons avec les outils suivants : *Bison*, *CodeWorker*, *ANTLR*, *StringTemplate*, *BYACC* et *Alternative Output*. Ces outils sont tous avec le code source disponible. Puisque *Alternative Output* est un outil construit à partir de *SableCC*, cette section est plus détaillée pour approfondir cet outil. À la fin de chaque section sur un outil, nous énumérons les avantages et les désavantages de cet outil par rapport à *Hiber* avec *SableCC*.

7.1 Autres applications

Une série d'applications dites commerciales existent aussi. Bien entendu, elles ne sont pas disponibles gratuitement et elles donnent très peu d'informations sur leurs méthodes de traitements et sur la construction de leurs outils. Dans les sous-sections suivantes, nous examinons les outils *DMS* et *Visual Parse++*. Nous énumérons les caractéristiques que nous pouvons faire ressortir malgré le peu de documentations disponibles.

7.1.1 *DMS*

DMS est un ensemble d'outils qui permet d'automatiser une analyse personnalisée selon le programme source, de modifier, de traduire, de générer des applications contenant un mélange arbitraire de langages. En résumé, *DMS* lit les descriptions formelles d'un ou de plusieurs langages, les analyse et les transforme. Ce sont des outils commerciaux appartenant à *Semantic Designs* (Semantic).

Sans approfondir tous les outils, nous fixons l'attention sur *Clone DR*. Ce dernier permet de trouver des blocs identiques, ou presque identiques, et de remplacer ces blocs par un appel abstrait comme des macros ou des procédures. Le code source passe dans un analyseur syntaxique pour former un arbre syntaxique abstrait, les langages acceptés sont le *Java*, le *C*, le *C++* et le *Cobol*. Par la suite, *Clone DR* permet de détecter si certaines parties de code sont identiques ou presque. Si c'est le cas, *Clone DR* modifie le code pour permettre la réutilisation des blocs redondants (Voir la figure suivante).

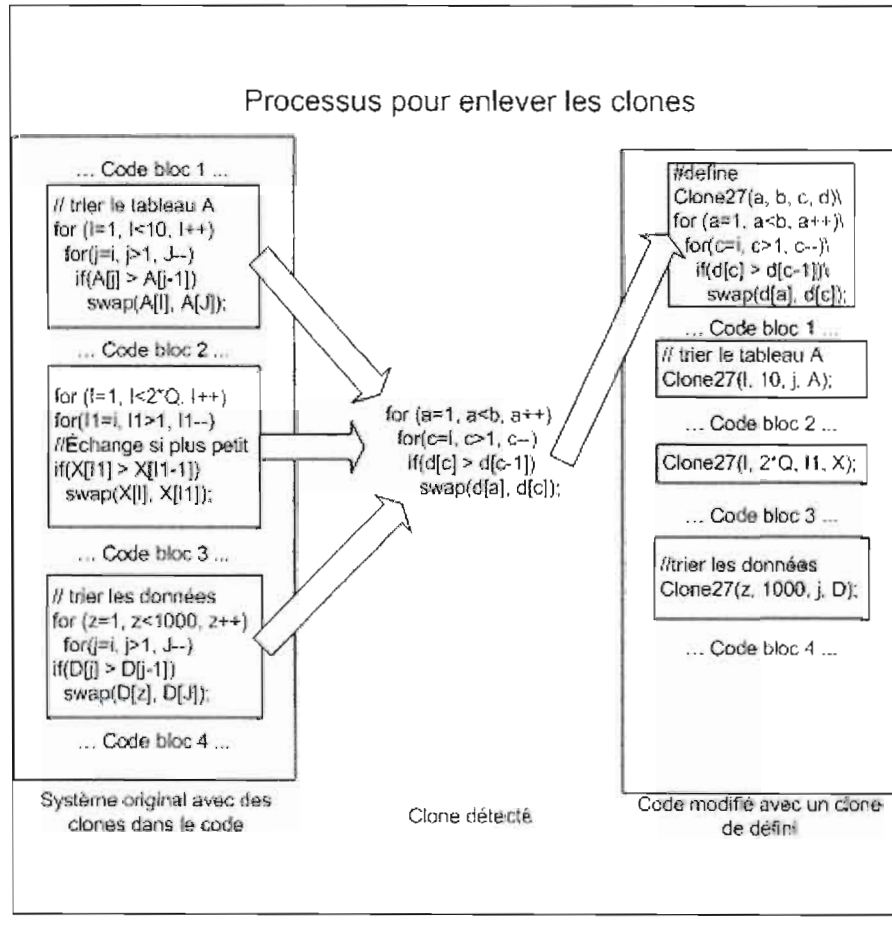

Figure 7.1 Remplacement des blocs de code identiques (Tirée de Semantic)

Clone DR peut identifier les blocs identiques, ou presque identiques, grâce à l'arbre syntaxique abstrait. Il examine l'arbre en essayant de trouver des sous-arbres identiques.

Pour augmenter la performance de la recherche, c'est un autre outil de *DMS* qui est utilisé, *PARLANSE*. C'est l'acronyme de « *A PARallel LANguage for Symbolic Expression* ». Cet outil permet une recherche en parallèle.

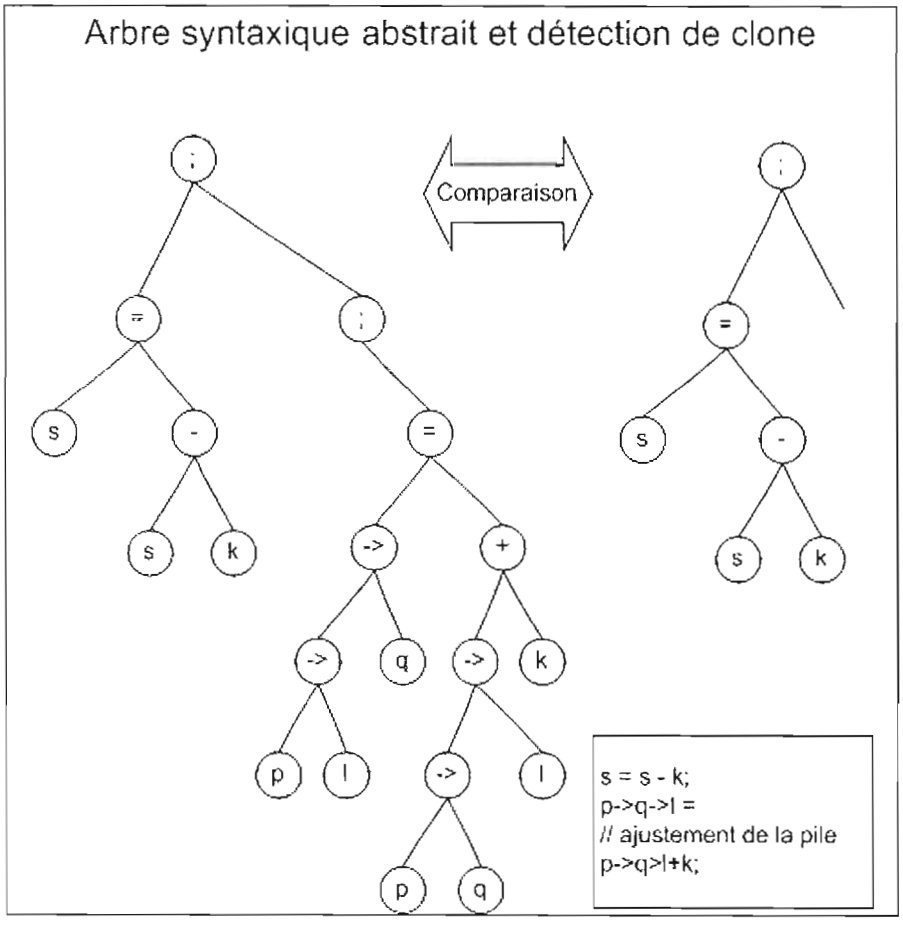

Figure 7.2 Détection des blocs identiques avec l'arbre syntaxique abstrait (Tirée de Semantic)

DMS fournit plusieurs fonctionnalités, en voici quelques unes :

- Une fonctionnalité intéressante est la disponibilité de plusieurs analyseurs lexicaux et analyseurs syntaxiques fondés sur l'Unicode en tenant compte d'une correction automatique des erreurs. Ce type d'analyseur peut manipuler n'importe quel type de

grammaires, même celles qui présentent des ambiguïtés. Selon leur site Web, ce type d'analyseur serait beaucoup plus puissant que *YACC/LALR*;

- Construction automatique d'un arbre syntaxique abstrait;
- *Clone DR* pour remplacer des blocs de code identiques ou presque identiques;
- *PARLANSE* est un langage parallèle pour expression symbolique afin d'écrire des séquences complexes d'actions.

Avantages par rapport à *Hiber* avec *SableCC*:

- *CloneDR* permet de remplacer des blocs de code identiques, ou presque identiques;
- Il permet la génération d'arbres syntaxiques abstraits pour des langages comme le *Java*, le *C*, le *C++* et le *Cobol*.

Désavantages par rapport à *Hiber* avec *SableCC*:

- Le code source de cet outil n'est pas disponible;
- Il n'est pas gratuit;
- Il offre peu de documentation concernant en ce qui a trait à l'implantation de l'outil.

7.1.2 *Visual Parse++*

Visual Parse++ est un outil d'analyseur syntaxique qui fournit un support aux principaux langages de programmation : *Java*, *C*, *C++*, *C#*, *Delphi* et *Visual Basic*. C'est une application qui est très portable; il est possible de l'installer sur plusieurs plateformes : Unix, Linux, Windows 98, Windows NT, Windows 2000 et Sun OS. En plus, *Visual Parse++* est parfaitement compatible avec l'Unicode et offre un environnement graphique en trois dimensions. Il est facile de naviguer et de déboguer sur l'arbre syntaxique en examinant chaque nœud sous forme graphique à l'écran.

Cependant, ce qui démarque cette application, c'est qu'elle admet les grammaires de type LALR(k). Ce type accepte une grande majorité de grammaires en gardant le nombre d'états inférieurs aux grammaires de type LR(k). En conséquence, l'analyseur syntaxique est plus petit. Il n'est cependant pas possible de transmettre plus de détails concernant cette application car *Stand Stone*, le propriétaire de ce produit, trace seulement les grandes lignes de marketing.

Avantages par rapport à *Hiber* avec *SableCC*:

- *Visual Parse++* accepte les grammaires de type LALR(k);
- Il accepte les principaux langages de programmation : *Java*, *C*, *C++*, *C#*, *Delphi* et *Visual Basic*;
- Il facilite avec un environnement graphique pour naviguer et déboguer sur l'arbre syntaxique.

Désavantages par rapport à *Hiber* avec *SableCC*:

- *Visual Parse++* ne rend pas le code source disponible;
- Il n'est pas gratuit;
- Il offre peu de documentation concernant en ce qui a trait à l'implantation de l'outil.

7.2 *Bison*

Bison est un générateur d'analyseurs syntaxiques qui accepte les grammaires de type LALR(1). Une fois la grammaire acceptée, elle est convertie en *C*. *Bison* est un outil construit en *C* et il utilise *M4* pour bien générer la grammaire en *C*. *M4* est un langage de traitement de macros. C'est une implantation des macros processeurs sous *Unix*. Un macro processeur est un outil de remplacement de textes. *M4* est un outil de génération beaucoup plus puissant que `MacroExpander` de *SableCC*. Cependant, il est plus complexe et plus difficile à utiliser. Il offre les possibilités suivantes :

- remplacement de textes;
- substitution de paramètres;
- inclusion de fichiers;
- manipulation de chaînes de caractères;
- évaluation conditionnelle;
- expressions arithmétiques;
- interfaçage avec le système;
- diagnostics pour le programmeur.

Les grammaires de *YACC* sont compatibles avec *Bison*. *YACC* représente aussi un outil de génération d'analyseurs syntaxiques. Un utilisateur familier avec *YACC* ne devrait pas avoir de problème avec l'utilisation de *Bison*. Le seul prérequis est d'être fluide avec le langage *C*, car les actions sont spécifiées en *C* (Le terme action représente le code écrit par le programmeur pour être exécuté à un moment spécifique par l'analyseur syntaxique). Ces deux applications sont fournies par le projet GNU[19].

Plusieurs versions de ces applications sont disponibles. La version originale de Bison ne génère pas le code dans d'autres langages. Certaines versions construites à partir de Bison sont maintenant capables de générer dans un autre langage. Par exemple, *jb* (*jb* est l'acronyme de *Java Bison*) génère le code en *Java*; *jb* prend le fichier *C* en sortie de *Bison*, il le parcourt et le convertit selon de nombreux patrons.

---

[19] GNU est l'acronyme récursif anglais de « *GNU's Not UNIX* », ce qui signifie « GNU n'est pas *Unix* ».

Avantages par rapport à *Hiber* avec *SableCC*:

- Utilisation de *M4* pour le traitement des macros;
- Compatibilité avec les grammaires de *YACC*;
- Disponibilité du code source.

Désavantages par rapport à *Hiber* avec *SableCC*:

- *Bison* ne permet pas la génération dans d'autres langages;
- L'utilisateur doit être familier en *C*.

7.3 *CodeWorker*

CodeWorker est un outil d'analyse syntaxique et un générateur de code polyvalent. C'est une application qui est conçue principalement pour automatiser la génération de code répétitifs. Ce n'est pas un outil de type « compilateur-compilateur ». *CodeWorker* interprète un langage de script. Ce langage possède ses propres fonctions et procédures. Aussi permet-il certaines instructions telles que `switch` et `foreach`. La syntaxe peut varier selon qu'il s'agisse d'écrire un analyseur syntaxique ou un modèle de génération de code.

Il existe trois familles de scripts :

- Les scripts d'enchaînement ou de liaison ayant l'extension « .cws »;
- Les scripts d'analyseurs syntaxiques BNF ayant l'extension « .cwp »;
- Les scripts de génération de code ayant l'extension « .cwt ».

Une quatrième famille peut exister; celle des scripts de transformations. Ces scripts sont un mélange des deux dernières familles. Cependant, ils s'apparentent plus à un script d'analyseur syntaxique. C'est pourquoi ces scripts ont la même extension que les scripts d'analyseurs syntaxiques.

Les scripts de l'analyseur syntaxique s'inspirent de la notation BNF. Dans *CodeWorker*, cette notation est bien étendue pour rendre l'analyseur syntaxique le plus efficace possible. Les extensions de cette notation sont utiles pour ignorer les espaces, pour vérifier si la fin du texte a été atteinte, pour forcer la continuité si une erreur se produit... Le type d'analyseur syntaxique est dit à descente récursive et il accepte les grammaires de la classe LL(k). Ce type d'analyseur ne résout pas les ambiguïtés, il est donc nécessaire d'arranger les règles de production.

Les scripts de génération de code permettent de générer le code de deux façons différentes. La manière traditionnelle qui implique la réécriture complète du code; la deuxième méthode qui consiste à injecter certaines parties de code à des endroits définis. Cette façon s'appelle expansion de code. Cette expansion consiste à garder le code ajouté ou modifié manuellement et à réécrire tout le reste. Pour bien identifier les emplacements dans le fichier texte, certains marqueurs indiqueront l'endroit. Un marqueur est une sorte de balise située entre les commentaires.

La génération du code dans *CodeWorker* est bien conçue pour tous les cas possibles. Par exemple, les commentaires reconnus dans *CodeWorker* sont les commentaires sur une ligne avec les deux barres obliques comme en *Java* et en *C++*. Cependant, l'outil permet l'utilisation dans un autre format de commentaires pour être compatible avec le langage souhaité.

Grâce à ses nombreuses fonctionnalités prédéfinies, *CodeWorker* contient certaines particularités qui le rendent très polyvalent en matière de génération du code. En revanche, cette application n'est pas un outil de type « compilateur-compilateur », car il ne fournit pas de structure de données pour pouvoir générer une structure de classes semblables à *SableCC*.

Avantages par rapport à *Hiber* avec *SableCC*:

- *Code Worker* permet l'exploitation d'autres extensions de la notation BNF;
- Il peut faire seulement une génération partielle;

- Il maintient un format de commentaires qui peut être paramétré;

- Il rend son code source disponible.

Désavantages par rapport à *Hiber* avec *SableCC*:

- *Code Worker* n'est pas de type « compilateur-compilateur »;

- Il ne sépare pas la logique et les règles de génération.

7.4 *ANTLR*

ANTLR est le descendant de *PCCTS*. Ces deux outils sont développés par Terence Parr, professeur du département d'informatique de l'université de San Francisco. *ANTLR* est l'acronyme de « *ANother Tool for Language Recognition* », ce qui signifie « Un autre outil pour la reconnaissance de langage »; *PCCTS* est l'acronyme de « *Purdue Compiler Construction Tool Set* », ce qui signifie « Trousse d'outils pour la construction de compilateur de Purdue ». Dans cette section, la version de *ANTLR* examinée, est la version 2.7.5. Cet outil permet d'accepter la description d'une grammaire et de générer une structure qui la représente dans le langage souhaité. Les grammaires acceptées sont de type LL(k). Les langages acceptés jusqu'à maintenant sont *Java*, *C++*, *C#* et bientôt *Python*.

ANTLR définit un concept de flots de jetons dans lequel chaque jeton est implanté comme un objet ayant la méthode `nextToken` et l'interface `TokenStream`. Ce concept permet de filtrer le flot, d'insérer dans le flot, de diviser le flot et de combiner plusieurs flots.

Il est possible d'adapter *ANTLR* pour qu'il puisse générer le code dans un autre langage. Pour réaliser cette adaptation, il faut tout d'abord bien comprendre comment fonctionne l'outil, car l'ajout de nouvelles classes est nécessaire. Normalement, il faut ajouter trois nouvelles classes :

- `NomLangageBlockFinishingInfo;`

- `NomLangageCharFormatter;`

- `NomLangageCodeGenerator.`

La première classe permet d'identifier les informations nécessaires à la fermeture d'un bloc. Un bloc peut être la condition `if` ou `switch` en *Java*. La deuxième classe est utilisée pour transformer la valeur des caractères ou bien aller chercher ces valeurs. Puis la troisième classe sert à la génération. Elle hérite de la classe `CodeGenerator`. Elle génère les classes : `MyParser`, `MyLexer` et `MyParserTokenTypes`.

Ce n'est pas tout ! Il faut ajouter trois autres fichiers dans le répertoire action à l'intérieur d'un autre répertoire ayant le nom du langage. Le premier fichier est un fichier texte qui comprend une série d'actions. Le deuxième est une interface comprenant une série de jetons. Enfin, le troisième fichier est une classe qui implante un analyseur lexical ayant comme interface le deuxième fichier. Par la suite, il faut modifier le fichier `ANTLR.java` pour ajouter le nouveau répertoire à ce nouveau langage.

Tout compte fait, il est bien possible d'ajouter une génération de code dans un autre langage. Cependant, la tâche est ardue et il faut être à l'aise avec l'outil. De plus, il faut modifier à chaque fois les classes actuelles.

Avantages par rapport à *Hiber* avec *SableCC*:

- *ANTLR* supporte EBNF;
- Il rend son code source disponible.

Désavantages par rapport à *Hiber* avec *SableCC*:

- *ANTLR* doit être modifié pour ajouter la génération dans un nouveau langage;
- Il ne reconnaît pas l'Unicode (16 bits), car dans le fichier `antlr.g`, une transformation vers les codes ASCII peut avoir lieu.

7.5 *StringTemplate*

L'outil *StringTemplate* est un moteur de modèles construit en *Java*. Il permet de générer des codes sources, des pages Web, des courriels ou tout autre format de textes. La prochaine version de *ANTLR* « *ANTLR v3.0* » comportera l'outil *StringTemplate*. Cet outil est né et a évolué durant le développement du site Web « http://www.jGuru.com ». Le besoin lors de la construction de ce site était de générer des pages dynamiques où un certain nombre de modèles étaient présents. Cet outil, une fois créé, a apporté un développement plus facile et une maintenance minime du code.

Un moteur de modèles est simplement un générateur de code qui utilise des modèles en format texte. Ce texte a certains trous où sont insérées certaines valeurs. Ces trous sont représentés par des attributs délimités par le symbole de dollar `$`. L'outil met à la disposition de l'utilisateur une série de fonctions prédéfinies. Par exemple, il est possible de faire des boucles avec l'instruction `foreach` et des vérifications de conditions avec l'instruction `if`. Si l'attribut est un agrégat avec des caractéristiques, il est possible de faire référence à ces dernières avec la nomenclature suivante : `attribut.caractéristique`. L'outil *StringTemplate* ignore les types des attributs. Il fait toujours l'appel de la manière suivante : `attribut.getCaractéristique`, sauf dans le cas où l'attribut serait un booléen.

StringTemplate est un outil très flexible utile à la génération. Il offre beaucoup plus de fonctionnalités que `MacroExpander` de *SableCC*. Il permet d'éviter d'écrire un script avec *Hiber* et de tout inclure dans le modèle. À ce stade, ce sont des points de vue personnels. À mon avis, le script *Hiber* permet de bien séparer la logique des règles de génération et le modèle, ce qui rend ces derniers beaucoup plus simples à comprendre.

Avantages par rapport à *Hiber* avec *SableCC*:

- La maintenance du code est minime;
- Le code source est disponible.

Désavantages par rapport à *Hiber* avec *SableCC*:

- *StringTemplate* ne sépare pas la logique et les règles de génération;
- Il n'est pas de type « compilateur-compilateur ». Cependant, si cet outil est ajouté à un outil comme *ANTLR*, il devient un atout important.

7.6 *BYACC*

BYACC est un analyseur syntaxique qui n'a aucune dépendance envers un compilateur en particulier, contrairement à un outil tel que *Bison*. *BYACC* est une variante de *YACC*, d'où son nom découle de *Berkeley YACC*. L'auteur de cet outil est le même auteur que l'analyseur syntaxique *Bison*. La structure de *BYACC* ressemble énormément à *YACC* : *BYACC* est construit en *C* et il utilise *M4*. Surtout, il est plus petit et rapide et il crée des analyseurs syntaxiques qui sont davantage rapides et petits.

Une variante de *BYACC* est *BYACC/J*. *BYACC*, tout comme *Bison*, construit un ou plusieurs fichiers en C afin de les compiler dans le but de produire un analyseur syntaxique pour les grammaires de type LALR. Cette dernière variante permet la génération des fichiers en *Java* lorsque l'argument `-J` est présent. Il ne faut aucune librairie spéciale pour faire fonctionner cet outil, seul un environnement de développement *Java* est nécessaire.

Avantages par rapport à *Hiber* avec *SableCC*:

- *BYACC* peut exécuter des fichiers *makefile*;
- Il possède des variantes qui sont codées en *C*, l'exécution est plus rapide;
- Il rend son code source disponible.

Désavantages par rapport à *Hiber* avec *SableCC* :

- *BYACC* ne permet pas la génération dans un autre langage;
- *BYACC/J* permet la génération d'un autre langage qui n'est pas une tâche simple et rapide.

7.7 *Alternative Output*

Dans cette section, nous examinons l'outil *Alternative Output*. Cet outil ajoute une nouvelle fonctionnalité à *SableCC*. Il permet de générer le code dans un autre langage que *Java*. Puisqu'il est directement relié à *SableCC* et qu'il a le même but que l'interpréteur *Hiber*, nous comparons tous les détails pertinents entre ces deux outils.

Alternative Output comporte un interpréteur *XSS*. C'est un langage dans lequel il est possible de construire ses propres patrons pour *SableCC*. Bien entendu, tous les patrons ayant l'extension « xss » passent dans l'interpréteur *XSS* d'*Alternative Output*. La combinaison de l'interpréteur *XSS* et des patrons produisent la structure à partir des données de l'arbre syntaxique. Il est possible de convertir l'arbre en format *XML* en utilisant le patron *XML*.

Présentement, *Alternative Output* permet la génération de code dans sept langages. Cet outil est construit par Indrek Mare, étudiant diplômé de l'université technologique de Tallinn. Les langages implémentés sont : *Java*, *C++*, *C#*, *OCaml*, *C*, *Python* et *XML*. Indrek a eu la collaboration de Fidel Viegas et de Patrick Lam pour implanter *Python* et *OCaml*. Pour bien comprendre cet outil, nous commençons par décrire la structure de l'interpréteur. Nous enchaînons en utilisant la structure de données définie par cet outil, puis nous faisons une description de l'outil et introduisons une nouvelle classe ajoutée à *SableCC*.

7.7.1 Structure de l'interpréteur

En premier lieu, nous survolons l'interpréteur et nous constatons que plusieurs mots sont réservés et plusieurs fonctionnalités sont introduites. Nous allons examiner les principaux mots réservés et les fonctionnalités. Cet interpréteur prend en considération les symboles suivants : `$` et `@`. Selon leur emplacement et leur syntaxe d'utilisation, ces symboles peuvent appeler certaines instructions, variables ou constantes.

7.7.1.1 Mots réservés

En premier lieu, il est possible de déclarer des variables qui sont globales tout au long de l'exécution. Cette déclaration se fait par le mot `param`.

```
$ param dest_dir = '.',
target_build, src = '.'
```

Figure 7.3 Déclaration d'une variable globale

Lors de la génération, chaque fichier patron nécessaire doit être inclus. Cette action se fait par le mot `include` suivi du nom du fichier.

```
$ include 'nodes.xss'
```

Figure 7.4 Déclaration d'une inclusion de fichier patron

Le contenu d'un fichier patron doit avoir un indicateur de début et de fin suivi d'un nom. Cet indicateur est représenté par le mot `template`.

```
$ template make_tokens()
//Contenu
$ end template
```

Figure 7.5 Déclaration d'un `template`

L'appel d'un patron se fait avec le mot `call` suivi du nom du contenu. Il est alors possible d'appeler un autre patron à l'intérieur d'un patron, mais il n'est pas nécessaire de terminer avec le patron courant.

```
$ call make_nodes()
```

Figure 7.6 Appel d'un fichier patron

Dans la prochaine sous-section, nous voyons que cet outil possède sa propre structure de données. À l'intérieur de cette structure, il est possible de boucler sur diverses listes. Pour ce faire, l'instruction `foreach` est utilisée suivie de la liste entre accolades.

```
$ foreach {//token[@text]}
//Contenu
$ end foreach
```

Figure 7.7 Fonction `foreach`

Il est possible de valider certaines conditions. Pour ce faire, l'utilisation des instructions `if` et `else` est disponible. Il faut noter que la partie du `else` est facultative.

```
$   if {@parser_index}
    TokenIndex = @parser_index
$   else
    TokenIndex = -1
$   end if
```

Figure 7.8 Fonctions `if` et `else`

7.7.1.2 Fonctionnalités

Plusieurs fonctionnalités sont ajoutées pour faire certaines transformations sur des littéraux. Nous en survolons les principales en démontrant, par moments, un exemple de leur utilisation.

`position` retourne la position de l'index.

```
Token token = new${position()-1}
```

Figure 7.9 Exemple d'utilisation de `position`

`count` retourne le nombre d'éléments dans la liste.

```
${count(parser_data/action_table/row)}
```

Figure 7.10 Exemple d'utilisation de `count`

`translate` remplace, dans une variable, un caractère par un autre caractère.

```
$ param pkgdir = {translate($package, '.', '/')}
```

Figure 7.11 Exemple d'utilisation de `translate`

`concat` permet de concaténer des caractères.

`substring` permet de prendre un sous-ensemble de caractères dans un littéral.

`substring_before` permet de retourner un ensemble de caractères avant un caractère en défini.

`substring_after` permet de retourner un ensemble de caractères après un caractère en défini.

`not` permet de faire la négation d'une expression.

```
$     if {not(@text)}
```

Figure 7.12 Exemple d'utilisation de `not`

`reverse` renverse une liste.

```
$   foreach {reverse(elem)}
```

Figure 7.13 Exemple d'utilisation de `reverse`

`string` convertit une variable en format littéral.

`touppercase` transforme le contenu d'une variable littérale en majuscule.

7.7.2 Structure de données

Une nouvelle classe s'est ajoutée aux classes originales de *SableCC*. Cette classe se nomme `DataTree` et elle permet de construire une structure de données. Les données sont regroupées sous la forme de la prochaine figure. Tous les noms ayant un autre nom en

dessous décalé vers la droite sont des listes, puis tous les noms portant un astérisque « * » à droite sont des éléments ayant d'autres attributs.

```
parser
  tokens
    token*
  prods
    prod*
      alt*
        elem*
  lexer_data
    state*
    accept_table
      state*
        i*
    goto_table
      state*
        goto*
  parser_data
    action_table
      row*
        action*
    goto_table
      row*
        action*
    errors
      i*
    error_messages
      msg*
    rules
      rule*
        action*
          arg*
```

Figure 7.14 Représentation de la structure de données

Au sein de l'élément `parser`, il est possible de constater qu'il y a quatre listes principales. Une liste contenant les informations utiles pour les jetons, la suivante pour les productions, une autre pour l'information concernant l'analyseur lexical et une dernière pour l'analyseur syntaxique. Peu de détails sont fournis dans la documentation de *Alternative Output* pour l'utilisation de cette structure. Il faut plutôt bien connaître le regroupement de ces listes et explorer la classe `DataTree` pour trouver le nom de tous les attributs. Pour bien comprendre la structure, il est recommandé de générer les données dans un fichier de type *XML* pour percevoir le regroupement des données.

7.7.3 Fonctionnement et utilisation

Au premier regard, il y a très peu de variations sur *SableCC*. Les fichiers originaux sont tous présents. Nous pouvons constater un nouveau paquetage contenant un interpréteur et d'autres fichiers patrons avec l'extension « xss ». Tous ces fichiers sont regroupés dans un répertoire avec le nom du langage auquel ils appartiennent. Aussi certaines classes se sont-elles ajoutées aux classes originales.

Chaque langage pouvant être généré par *Alternative Output* doit avoir un fichier qui a pour nom `main.xss`. Ce fichier est le point d'entrée pour la génération de code. Normalement, la structure de ce fichier commence par les variables globales, les insertions des patrons puis les appels vers les patrons. Puisque l'interpréteur lit ce fichier dans un ordre séquentiel, l'ordonnance des appels est importante.

Un fichier patron permettant de vérifier et d'aider, dans certains cas, à la compilation, qui est également disponible dans certains langages. Normalement, le nom de ce fichier est `utils.xss`.

Pour utiliser l'outil de génération de code, il suffit d'appeler *SableCC* en lui passant les paramètre suivants : `-t` pour désigner la source de génération; `-p` pour désigner une ou plusieurs « clés – valeurs », où ces dernières sont interprétées par l'interpréteur de l'outil et la grammaire. Dans l'exemple suivant, nous pouvons voir que la génération est en *Java*. Par la suite, la valeur « src » est assignée à la clé « src ». Chaque langage a sa propre série de clés.

L'utilisation de ces clés n'est pas toujours obligatoire et il est possible de se référer à la documentation de l'outil.

```
sablecc -t java,java-build -p src src grammar-file.sablecc3
```

Figure 7.15 Appel de l'outil *Alternative Output*

7.7.4 Nouvelle classe

Comme nous l'avons déjà mentionné, une nouvelle classe s'est ajoutée dans le paquetage original de *SableCC*. Cette classe est `DataTree.java` et elle construit la structure de données. De plus, cette classe détient des méthodes qui permettent de transformer la structure de données sous un format *XML*.

Cette classe dispose aussi d'une classe interne appelée `DataTreeMaker` qui construit la structure de données. C'est dans cette classe que chaque attribut des éléments est assigné. La génération dans le langage *C++* demande certaines données qui ne sont pas fournies dans l'arbre standard. Une nouvelle méthode rattachée à cette classe ajoute certains attributs pour le langage *C++*. Pour l'instant, ces données sont spécifiques seulement à la génération de code en *C++*.

7.7.5 Comparaison entre *Alternative Output* et l'interpréteur *Hiber*

Il est possible de comparer les fichiers patrons des langages *C++* et *Python* de l'implantation faite par l'outil *Alternative Output* et les patrons des scripts de *Hiber*. Pour ce qui est d'*Alternative Output*, la lecture est beaucoup plus complexe. Les patrons contiennent des balises de code, ce qui rend l'interprétation plus laborieuse et plus distincte que les fichiers originaux de *SableCC*. Dans *Hiber*, les standards par rapport aux fichiers originaux sont respectés. La génération utilise toujours la classe `MacroExpander`.

Les deux outils possèdent une structure de données très similaire. Cependant, dans le cas de l'outil *Alternative Output*, il faut bien connaître l'outil *SableCC* pour pouvoir générer un nouveau langage correctement. Aucune documentation n'explique la signification des champs dans la structure de données et très peu de documentation est disponible sur l'utilisation de cet outil.

Ce qui rend l'outil *Alternative Output* moins intéressant, c'est que la structure de données combinée aux balises ne fournit pas toutes les informations nécessaires à la génération du code. Une méthode a dû être ajoutée dans la classe `DataTree` pour fournir certaines informations nécessaires à la génération des fichiers *C++*.

Avantages par rapport à *Hiber* avec *SableCC*:

- *Alternative Output* permet la représentation de l'arbre à l'intérieur d'un format *XML*;
- Il peut générer dans plusieurs langages;
- Son code source est disponible.

Désavantages par rapport à *Hiber* avec *SableCC*:

- *Alternative Output* comporte une nouvelle méthode ajoutée pour le *C++*, ce qui démontre que la structure de données fournies par l'outil a des faiblesses;
- Il ne sépare pas la logique et les règles de génération;
- Il offre une documentation insuffisante sur la structure de données.

7.8 Conclusion

Ce chapitre a exposé d'autres outils ayant une similitude avec l'interpréteur *Hiber* de *SableCC*. La comparaison des outils suivants : *DMS*, *Visual Parse++*, *Bison*, *CodeWorker*, *ANTLR*, *StringTemplate*, *BYACC* et *Alternative Output* a permis de percevoir les avantages et les désavantages de chacun d'eux.

TRAVAUX FUTURS ET CONCLUSION

SableCC est un outil de type « compilateur-compilateur » qui permet de générer des analyseurs lexicaux, des analyseurs syntaxiques et des arbres syntaxiques en *Java*. Ces analyseurs sont toujours bien utiles pour lire certains textes formatés. Cependant, *SableCC* est limité à des analyseurs construits en *Java*. Permettre à cet outil de générer le code dans un autre langage rendrait *SableCC* plus polyvalent et plus intéressant à utiliser.

L'ajout de cette fonctionnalité s'est faite par la fusion de trois composantes. La première est le langage appelé *Hiber*; la deuxième est un interpréteur qui est en mesure de lire des scripts écrits dans le langage *Hiber*; la troisième est une structure de données générée par *SableCC* et qui est passée en paramètre à l'interpréteur. Grâce à ces trois composantes, il est possible de permettre à *SableCC* de générer dans un autre langage orienté objets.

La génération de code se fait de la même manière que la version originale de SableCC. Il faut des fichiers patrons qui contiennent des motifs du langage. C'est toujours la classe `MacroExpander` qui est utilisée pour l'appel des macros. Par contre, une nouvelle étape s'est ajoutée. Il faut maintenant écrire un script qui contient les règles de générations de code. Ce script est dans le langage *Hiber*. C'est dans celui-ci que nous indiquons le ou les fichiers patrons à utiliser, le ou les fichiers à créer et les macros à utiliser.

L'un des objectifs fixés est de rendre l'écriture du script la plus simple. Cet objectif est atteint puisque le langage *Hiber* ne comporte pas beaucoup de mots réservés. De plus, l'interpréteur fournit aussi une série de procédures et de fonctionnalités qui sont faciles à utiliser.

Un autre objectif vise à permettre la génération de code dans n'importe quel langage orienté objets. Pour le démontrer, nous avons usé de trois langages qui offrent des particularités intéressantes. Le premier est *Java*. Ce langage démontre que les trois composantes ajoutées à *SableCC* fonctionnent bien. Le deuxième est *C++*, d'où plusieurs

similitudes avec le langage *Java*. Puisque le *C++* est un langage très utilisé, il augmente le niveau d'intérêt pour la génération des classes. Le troisième est *Python*. Ce langage est très différent des deux premiers sur le plan syntaxique. De plus, la génération des classes se fait dans un seul fichier. Une série de tests dans lesquels nous avons utilisé trois grammaires démontrent le bon fonctionnement des trois langages. Ces tests confirment déjà la fiabilité des composantes.

Cette implantation permet de bien séparer les règles de générations, les données et les motifs, ce qui aide à la maintenance du code et à la réutilisation. Il serait souhaitable dans un travail à venir d'ajouter deux nouveaux langages : le *C#* et le *Perl*. Le *C#* est un langage qui gagne en popularité et le *Perl* est un langage de script sous Unix qui est très utilisé. De plus, ce travail a apporté une génération de code dans n'importe quel langage orienté objet. Certaines autres catégories de langages pourraient bénéficier de ce générateur. Par exemple, le *C* étant un langage procédural, il serait sûrement capable d'être généré. Il resterait à le démontrer.

APPENDICE A

GRAMMAIRE HIBER

Cet appendice représente la grammaire *Hiber*.

```
Package org.sablecc.sablecc;
Helpers
  all = [0 .. 0xffff];
  letter = [['a'..'z']+['A'..'Z']];
  digit = ['0'..'9'];

  cr = 13;
  lf = 10;
  tab = 9;
  eol = cr | lf | cr lf;
  not_cr_lf = [all - [cr + lf]];
  string_char = [not_cr_lf - '"'];
  not_star = [all - '*'];
  not_star_slash = [not_star - '/'];

  line_comment = '//' not_cr_lf* eol?;
  long_comment = '/*' not_star* '*'+ (not_star_slash not_star*
                 '*'+)* '/';

Tokens
  assign     = '=';
  l_par      = '(';
  r_par      = ')';
```

```
colon      = ':';
semi_colon = ';';
l_brace    = '{';
r_brace    = '}';
dot        = '.';
not        = '!';
plus       = '+';
comma      = ',';
mult       = '*';
div        = '/';
minus      = '-';
mod        = '%';
and        = '&&';
or         = '||';

gt         = '>';
ge         = '>=';
eq         = '==';
ne         = '!=';
le         = '<=';
lt         = '<';

foreach    = 'foreach';
in         = 'in';
reversed   = 'reversed';
if         = 'if';
else       = 'else';
global     = 'global';
switch     = 'switch';
case       = 'case';
default    = 'default';
return     = 'return';
procedure  = 'procedure';
function   = 'function';

string_literal = '"' (string_char | '""')* '"';
```

```
  ident  = letter (letter | digit)*;
  number = digit+;

  blank = (' ' | cr | lf | tab | line_comment | long_comment)+;

Ignored Tokens
  blank;

Productions
  Script = global_declarations? procedure_or_function+  {-> New
           script([global_declarations.ident],
           [procedure_or_function.procedure],
           [procedure_or_function.function]) };

  global_declarations {-> ident* } = global ident_list?
           semi_colon  {-> [ident_list.ident] };

  ident_list {-> ident* } = ident ident_list_tail*  {-> [ident,
           ident_list_tail.ident] };

  ident_list_tail {-> ident} = comma ident {-> ident};

  procedure_or_function {-> P.procedure? P.function?} =
           {procedure} P.procedure {-> procedure Null } |
           {function} P.function    {-> Null function };

  procedure = T.procedure ident l_par param_list? r_par l_brace
           stmt* r_brace {-> New procedure(ident,
           [param_list.ident], [stmt]) };

  function = T.function ident l_par param_list? r_par l_brace
           stmt* P.return r_brace {-> New function(ident,
           [param_list.ident], [stmt], P.return.exp) };

  return {-> exp } = T.return exp semi_colon {-> exp };

  param_list {-> ident* } = ident param_list_tail*  {-> [ident,
           param_list_tail.ident] };
```

```
param_list_tail {-> ident } = comma ident {-> ident };


stmt =  {assign} ident assign exp semi_colon {-> New
          stmt.assign(ident, exp) } |
        {global_assing} global ident assign exp semi_colon {->
          New stmt.global_assign(ident, exp) } |
        {foreach} foreach ident in reversed? exp stmt_brace {
          -> New stmt.foreach(ident, reversed, exp,
          [stmt_brace.stmt]) } |
        {if} if l_par exp r_par stmt_brace else_part? {-> New
          stmt.if(exp, [stmt_brace.stmt], [else_part.stmt]) }|
        {call} call semi_colon {-> New stmt.call(call.ident,
          [call.exp]) } |
        {switch} switch l_par exp r_par case_brace {-> New
          stmt.switch(exp, [case_brace.case],
          case_brace.default) };


stmt_brace {-> stmt* } = l_brace stmt* r_brace {-> [stmt] };


else_part {-> stmt* } = else stmt_brace{->[stmt_brace.stmt] };


case_brace {-> P.case+ P.default? } = l_brace P.case+
          P.default? r_brace {-> [case] default };


case = T.case string_literal additional_string_literal* colon
          stmt_brace {-> New case([string_literal,
          additional_string_literal.string_literal],
          [stmt_brace.stmt]) };


additional_string_literal {-> string_literal } = comma
          string_literal {-> string_literal };


default = T.default colon stmt_brace {-> New
          default([stmt_brace.stmt]) };


exp = {or} exp or and_exp {-> New exp.or(exp, and_exp.exp) } |
      {and} and_exp {-> and_exp.exp };


and_exp {-> exp } = {and} and_exp and compare_exp {-> New
                      exp.and(and_exp.exp, compare_exp.exp)} |
                    {compare}compare_exp{->compare_exp.exp };
```

```
compare_exp {-> exp } =
  {gt} [l_add_exp]:add_exp gt [r_add_exp]:add_exp {-> New
     exp.gt(l_add_exp.exp, r_add_exp.exp) } |
  {ge} [l_add_exp]:add_exp ge [r_add_exp]:add_exp {-> New
     exp.ge(l_add_exp.exp, r_add_exp.exp) } |
  {eq} [l_add_exp]:add_exp eq [r_add_exp]:add_exp {-> New
     exp.eq(l_add_exp.exp, r_add_exp.exp) } |
  {ne} [l_add_exp]:add_exp ne [r_add_exp]:add_exp {-> New
     exp.ne(l_add_exp.exp, r_add_exp.exp) } |
  {le} [l_add_exp]:add_exp le [r_add_exp]:add_exp {-> New
     exp.le(l_add_exp.exp, r_add_exp.exp) } |
  {lt} [l_add_exp]:add_exp lt [r_add_exp]:add_exp {-> New
     exp.lt(l_add_exp.exp, r_add_exp.exp) } |
  {add} add_exp {-> add_exp.exp };


add_exp {-> exp } =
  {plus} add_exp plus mult_exp {-> New exp.plus(add_exp.exp,
       mult_exp.exp) } |
  {minus} add_exp minus mult_exp {-> New
       exp.minus(add_exp.exp, mult_exp.exp) } |
  {mult} mult_exp {-> mult_exp.exp };


mult_exp {-> exp } =
  {mult} mult_exp mult l_unary_exp {-> New
        exp.mult(mult_exp.exp, l_unary_exp.exp) } |
  {div} mult_exp div l_unary_exp {-> New exp.div(mult_exp.exp,
        l_unary_exp.exp) } |
  {mod} mult_exp mod l_unary_exp {-> New exp.mod(mult_exp.exp,
        l_unary_exp.exp) }|
  {l_unary} l_unary_exp {-> l_unary_exp.exp };


l_unary_exp {-> exp } =
  {not} not l_unary_exp {-> New exp.not(l_unary_exp.exp) } |
  {r_unary_exp} r_unary_exp {-> r_unary_exp.exp};


r_unary_exp {-> exp } =
  {dot} r_unary_exp dot ident {-> New exp.dot(r_unary_exp.exp,
       ident) } |
  {base} base_exp {-> base_exp.exp};
```

```
base_exp {-> exp } =
  {string}string_literal{->New exp.string(T.string_literal)}|
  {number} number {-> New exp.number(number) } |
  {ident} ident {-> New exp.ident(ident) } |
  {par} l_par exp r_par {-> exp} |
  {call} call {-> New exp.call(call.ident, [call.exp]) };

call {-> ident exp* } =
  ident l_par arg_list? r_par  {-> ident [arg_list.exp] };

arg_list {-> exp* } =
  exp arg_list_tail* {-> [exp, arg_list_tail.exp] };

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

Abstract Syntax Tree

```
script =
  [globals]:ident* [procedures]:P.procedure*
   [functions]:P.function*;

procedure =
 [name]:ident [params]:ident* [stmts]:stmt*;

function =
 [name]:ident [params]:ident* [stmts]:stmt* [return]:exp;

stmt =
  {assign}        [var]:ident exp |
  {global_assign} [var]:ident exp |
  {foreach}       [var]:ident reversed? exp [stmts]:stmt*|
  {if}            exp [then_stmts]:stmt* [else_stmts]:stmt* |
  {call}          [proc_name]:ident [args]:exp* |
```

```
  {switch}        exp [cases]:P.case+ P.default?;

case =
  [string_literals]:string_literal+ [stmts]:stmt*;

default =
  [stmts]:stmt*;

exp =
  {or}     [l_exp]:exp [r_exp]:exp |
  {and}    [l_exp]:exp [r_exp]:exp |
  {gt}     [l_exp]:exp [r_exp]:exp |
  {ge}     [l_exp]:exp [r_exp]:exp |
  {eq}     [l_exp]:exp [r_exp]:exp |
  {ne}     [l_exp]:exp [r_exp]:exp |
  {le}     [l_exp]:exp [r_exp]:exp |
  {lt}     [l_exp]:exp [r_exp]:exp |
  {plus}   [l_exp]:exp [r_exp]:exp |
  {minus}  [l_exp]:exp [r_exp]:exp |
  {mult}   [l_exp]:exp [r_exp]:exp |
  {div}    [l_exp]:exp [r_exp]:exp |
  {mod}    [l_exp]:exp [r_exp]:exp |
  {not}    exp |
  {dot}    exp ident |
  {string} string_literal |
  {number} number |
  {ident}  ident |
  {call}   [func_name]:ident [args]:exp*;
```

APPENDICE B

SCRIPT POUR LA GENERATION EN JAVA

Cet appendice représente la procédure `parser` d'un script pour la génération en *Java*.

```
procedure parser(){
  macroFile = loadmacro(folder + "parser.txt");

  if (PackageName == ""){
       nodePackageName = "parser";
       importLexer = "lexer";
       importNode = "node";
       importAnalysis = "analysis";
  }
  else{
       nodePackageName = PackageName + ".parser";
       importLexer = PackageName + ".lexer";
       importNode = PackageName + ".node";
       importAnalysis = PackageName + ".analysis";
  }

  dir = DirName + "/parser";

  fileParser = createfile(dir, "Parser.java");
  macro(fileParser, macroFile,  "ParserHeader", nodePackageName,
      importLexer, importNode, importAnalysis);

  if(ActivateFilter && !GrammarHasTransformations){
```

```
        macro(fileParser, macroFile,
             "ParserNoInliningPushHeader");
        macro(fileParser, macroFile,  "ParserCommon", ",
             true", ", false");
 }
 else{
        macro(fileParser, macroFile,
             "ParserNoInliningPushHeader");
        macro(fileParser, macroFile, "ParserCommon", "", "");
 }

 foreach reduce in Reduces{
      if(ActivateFilter && !GrammarHasTransformations){
      macro(fileParser, macroFile,  "ParserNoInliningReduce",
           reduce.index, reduce.productionNameIndex,
           reduce.startsWith, reduce.productionName);
 }
 else{
      macro(fileParser, macroFile,  "ParserInliningReduce",
           reduce.index, reduce.productionNameIndex,
           reduce.productionName);
 }
}

 macro(fileParser, macroFile,  "ParserParseTail",
      Productions.first.name);

 foreach reduce in Reduces{
        macro(fileParser, macroFile,  "ParserNewHeader",
             reduce.index, reduce.productionName);

 foreach node in reduce.nodes{
   //macro(fileParser, macroFile,  node.macro, node.arguments);
   switch(node.nodeType){
     case "inAAltTransform", "inAParams":{
       if(node.typeNameStartsWith == "L"){
            macro(fileParser, macroFile,
                 "ParserListVariableDeclaration",
                 node.position);
       }
```

```
  else{
    if(node.typeNameStartsWith == ""){
        macro(fileParser, macroFile,
             "ParserNullVariableDeclaration",
             node.position);
    }
    else{
        macro(fileParser, macroFile,
             "ParserSimpleVariableDeclaration",
             node.typeName, node.typeNameLowerCase,
             node.position);
    }
  }
}

case "outAAltTransform":{
  foreach term in node.terms{
        macro(fileParser, macroFile,
             "ParserNewBodyListAdd", term.typeName,
             term.position);
  }
        macro(fileParser, macroFile,  "ParserNewTail");
}

case "inASimpleTerm", "inASimpleListTerm":{
        macro(fileParser, macroFile,  "ParserSimpleTerm",
             node.typeNameLowerCase, node.position,
             node.type, node.elementPosition,
             node.positionMap);
}

case "inANewTerm", "inANewListTerm":{
        macro(fileParser, macroFile,
             "ParserBraceOpening");
}

case "inAListTerm":{
        macro(fileParser, macroFile,
             "ParserBraceOpening");
  foreach term in node.terms{
```

```
            if(term.typeNameStartsWith == "L"){
                macro(fileParser, macroFile,
                      "ParserListVariableDeclaration",
                      term.position);
    }
    else{
      if(term.typeNameStartsWith == ""){
                macro(fileParser, macroFile,
                      "ParserNullVariableDeclaration",
                      term.position);
      }
      else{
        macro(fileParser, macroFile,
             "ParserSimpleVariableDeclaration",
             term.typeName, term.typeNameLowerCase,
             term.position);
      }
    }
  }
}

case "outAListTerm":{
  foreach term in node.terms{
    if(term.typeNameStartsWith == "L"){
         macro(fileParser, macroFile,
               "ParserTypedLinkedListAddAll", "list",
               term.listPosition, "list", term.position);
    }
    else{
         macro(fileParser, macroFile,
               "ParserTypedLinkedListAdd", "list",
               term.listPosition, term.typeNameLowerCase,
               term.position);
    }
  }
  macro(fileParser, macroFile,  "ParserBraceClosing");
}

case "outANewTerm", "outANewListTerm":{
```

```
      macro(fileParser, macroFile,  "ParserNewBodyNew",
            node.typeName, node.position, node.newAltName);
      i=0;
      foreach param in node.params{
      if(param.paramTypeName == ""){
        if(i == 0){
            macro(fileParser, macroFile,
                  "ParserNew&ListBodyParamsNull", "null");
        }
        else{
            macro(fileParser, macroFile,
                  "ParserNew&ListBodyParamsNull", ",
                  null");
        }
      }
      else{
          if(i == 0){
              macro(fileParser, macroFile,
                    "ParserNew&ListBodyParams",
                    param.paramTypeName,
                    param.paramPosition);
          }
      else{
          name = ", " + param.paramTypeName;
          macro(fileParser, macroFile,
                "ParserNew&ListBodyParams", name,
                param.paramPosition);
      }
    }
    i = i + 1;
    }
    macro(fileParser, macroFile,  "ParserNewBodyNewTail");
    macro(fileParser, macroFile,  "ParserBraceClosing");
}

case "caseAElem":{
     macro(fileParser, macroFile,  "ParserNewBodyDecl",
           node.current);
}
```

```
    }
  }
}

  macro(fileParser, macroFile,  "ParserActionHeader");

  foreach actionTable in ParsingTable.ActionTable{

    macro(file, macroFile,  "ParserActionLine");

    foreach action in actionTable.listActionTable{
      macro(file, macroFile,  "ParserActionGoto", action.from,
            action.actionNbr, action.to);
    }

    macro(file, macroFile,  "ParserActionLineEnd");

  }

  macro(fileParser, macroFile,  "ParserActionTail");


  macro(fileParser, macroFile,  "ParserGotoHeader");
  macro(fileParser, macroFile,  "ParserGotoTail");
  macro(fileParser, macroFile,  "ParserErrorsHeader");
  macro(fileParser, macroFile,  "ParserErrorsTail");
  macro(fileParser, macroFile,  "ParserErrorIndexHeader");
  macro(fileParser, macroFile,  "ParserErrorIndexTail");
  macro(fileParser, macroFile,  "ParserTail");

  close(fileParser);

  fileToken = createfile(dir, "TokenIndex.java");
  macro(fileToken, macroFile,  "TokenIndexHeader",
      nodePackageName, importNode, importAnalysis);

  i = 0;
```

```
  foreach terminal in Tokens{
    if(terminal.isFixed){
      macro(fileToken, macroFile,  "TokenIndexBody",
            terminal.name, i);
      i = i + 1;
    }
  }

  macro(fileToken, macroFile,  "TokenIndexTail", i);

  close(fileToken);

  fileParserException = createfile(dir, "ParserException.java");
  macro(fileParserException, macroFile,  "ParserException",
        nodePackageName, importNode);
  close(fileParserException);

  fileState = createfile(dir, "State.java");
  macro(fileState, macroFile,  "State", nodePackageName);
  close(fileState);

}
```

APPENDICE C

GRAMMAIRE SABLECC 3

Cette grammaire est tirée du site de *SableCC* (SableCC).

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * 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 =
        {parsed} alt_name? [elems]:elem* alt_transform?
         {-> New alt.parsed(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 =
    {parsed} [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*;
```

BIBLIOGRAPHIE

(Aho, Sethi et Ullman, 1991) Aho, Alfred, Ravi Sethi et Jeffrey Ullman. 1991. *Compilateurs principes, techniques et outils*. Paris, InterEditions, 875 p.

(Akers, Baxter, Mehlich, Ellis et Luecke, 2005) Akers, Robert, Ira Baxter, Michael Mehlich, Brian Ellis et Kenn Luecke. 2005. *Reengineering C++ Component Models via Automatic Program Transformation*, 12th Working Conference on Reverse Engineering, IEEE Computer Society, p. 13-22

(Alternative Output) Alternative Output, URL: http://www.mare.ee/indrek/sablecc/#altgen

(Appel, 2002) Appel, Andrew W. 2002. *Modern Compiler Implementation in Java, second edition.* Cambridge (Angleterre) : Cambridge University Press, 501 p.

(Autoconf) Autoconf, URL: http://www.gnu.org/software/autoconf/

(AutoGen) AutoGen, URL: http://www.gnu.org/software/autogen/

(Baxter, 2002) Baxter, Ira. 2002. *Parallel Support for Source Code Analysis and Modification*, 13 p. URL: http://www.semdesigns.com/Company/Publications/parallelism-in-symbolic-computation.pdf

(Baxter, Pidgeon et Mehlich, 2004) Baxter, Ira, Christopher Pidgeon et Michael Mehlich. 2004. *DMS: Program Transformations for Practical Scalable Software Evolution*, In Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, p. 625-634

(Baxter, Yahin, Moura, Sant'Anna et Bier, 1998) Baxter, Ira, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna et Lorraine Bier. 1998. *Clone Detection Using Abstract Syntax Trees*, In International Conference On Softwate Maintenance, IEEE Computer Society, p. 368

(BYACC/J, 2005) BYACC/J, 2005, URL: http://byaccj.sourceforge.net/

(Corbett, 1993) Corbett, Robert. 1993. byacc, version 1.9, URL: http://www.isc.org/sources/devel/tools/byacc-1.9.txt

(Deitel et Deitel, 1994) Deitel, M. Harvey et Paul J. Deitel. 1994. *C++ How to Program, Englewood Cliffs*, New Jersey, Prentice Hall, 950 p.

(Donnelly et Stallman, 2005) Donnelly, Charles et Richard Stallman. 2005. *Bison*, Boston, Free Software Foundation, 142 p.

(Friedman et Koffman, 1997) Friedman, L. Frank et Elliot B. Koffman. 1997. *Problem solving, abstraction, and design using C++, second edition*, Addison-Wesley, 713 p.

(Gagnon, 1998) Gagnon, Etienne. 1998. *SableCC, an Object-Oriented Compiler Framework.* Thèse de maîtrise, Montréal, Université McGill, 107 p.

(Grand dictionnaire terminologique) Grand dictionnaire terminologique, URL: http://www.granddictionnaire.com

(Java Bison) Java Bison, URL: http://serl.cs.colorado.edu/~arcadia/Software/jb.html

(Agbakpem, 2006) Agbakpem, Komivi Kevin. 2006. *Transformation automatique d'arbres syntaxiques avec SableCC.* Thèse de maîtrise, Montréal, Université du Québec à Montréal, 204 p.

Lemaire, Cédric, 2005. *CodeWorker Parsing Tool and Code Generator – User's Guide & Reference Manual*, Release 4.0.2, Free Software Foundation, 321 p.

(Martelli, 2004) Martelli, Alex. 2004. *Python en concentré*, Paris, O'Reilly, 644 p.

(Mertz, 2003) Mertz, David. 2003. *Text Processing in Python*, Addison-Wesley, 520 p.

(Parr, 2005) Parr, Terence. 2005. *An Introduction To ANTLR.* University of San Francisco, http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html

(SableCC) SableCC, URL : http://sablecc.org

(Semantic) Semantic Design, URL : http:// www.semdesigns.com

(StringTemplate) StringTemplate, URL: http://www.stringtemplate.org/

(Sudkamp, 1997) Sudkamp, Thomas A. 1997. *Language and Machines, An Introduction to the Theory of Computer Science*, second Edition. Addison-Wesley, 569 p.

(Visual Parse++)Visual Parse++, URL: http://www.sand-stone.com/

(World Wide Web Consortium) World Wide Web Consortium (W3C), URL: http://www.w3c.org