Rapport de Stage de Maîtrise Informatique
Le Langage Chocolat

Forax Rémi

1er juillet 1997

Je tiens à remercier tout particulièrement Mr Gilles Roussel et Mr Didier Parigot pour leur disponibilité et leurs réponses à mes nombreuses questions.
Mr Loïc Corenson et Etienne Duris pour leur infinie patience face à mon incompréhension et mon ignorance.
Ainsi que tous les membres du projet OSCAR de l'INRIA pour leur accueil chaleureux.

Table des matières

Chapitre 1

Introduction

Le but de ce stage est d'implémenter les bases d'un nouveau langage permettant de manipuler des grammaires attribuées. Celui-ci devra s'intégrer harmonieusement à un environnement de développement Java. De plus, dans la perspective d'effectuer l'année prochaine un DEA, je voulais connaître les conditions dans lesquelles la conduite d'un projet de recherche s'effectuait.
Ce stage s'est décomposé en deux grandes phases. J'ai dû dans un premier temps développer les classes Java pour permettre de lire et d'écrire le langage intermédiaire Cacao (le *package* Cacao). Langage dont les spécifications ont été faites en concertation avec l'équipe de recherche de l'INRIA. Dans un second temps, j'ai créé le package *package* Chocolat qui contient principalement le compilateur de Cacao en Chocolat.
Dans ce rapport, je vais commencer par décrire brièvement les grammaires attribuées et leur spécification en Chocolat. Ensuite je décrirai de façon plus technique mon implantation.

Chapitre 2

Les Grammaires Attribuées et Chocolat

Je vais dans un premier temps rappeler ce qu'est une grammaire attribuée puis dans un second temps introduire un nouveau langage de manipulation des grammaires attribuées nommé Chocolat.

2.1 Les grammaires

Une grammaire est un ensemble de règles de substitutions organisées sous forme d'arbre permettant initialement de reconnaître un langage. Prenons l'exemple du langage permettant de reconnaître l'expression suivante

$$1 + 2 + 3 \tag{2.1}$$

et regardons sa spécification :

```
Expr = Expr '+' Expr
    |  VALUE
```

On peut traduire l'expression 2.1 sous forme d'un arbre :

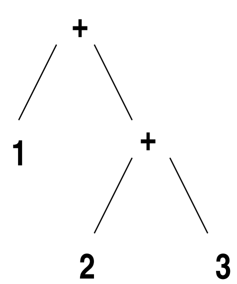


Fig. 2.1 – *Exemple1*

2.2 Les attributs et les arbres d'évaluation

On désire maintenant donner une sémantique permettant de calculer la valeur de l'expression associée à l'arbre pécédent, pour cela on utilise des attributs, d'où le nom de grammaire

attribuée.
A chaque noeud de l'arbre on associe un attribut stockant le calcul intermédiaire permettant d'obtenir le résultat.

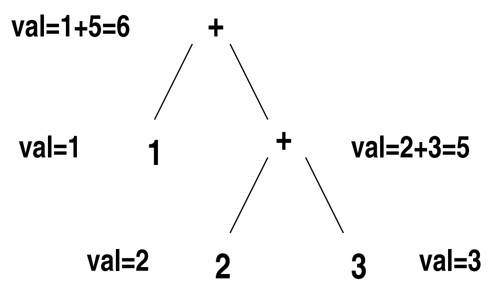


FIG. 2.2 – *Exemple2*

Il faut donc écrire les règles qui permettent de calculer la valeur des attributs des noeuds en fonction de son noeud père et des noeuds fils. On écrira cette règle directement dans la grammaire de cette façon :

```
Expr = Expr1 '+' Expr2       val=Expr1.val+Expr2.val
    |  VALUE                 val=VALUE.val
```

On vient donc de créer une grammaire possédant des attributs, ici val, et des règles de calcul entre les differentes occurences d'attributs appelées règles sémantiques.

2.3 La grammaire en Chocolat

Nous allons maintenant décrire un nouveau langage travaillant sur les grammaires attribuées, le langage Chocolat. Ce langage devra contenir une partie de déclaration permettant une spécification de la grammaire et d'une partie de codage des règles sémantiques.
Prenons maintenant, un exemple un peut plus complexe :

```
List = Cons VALUE List
    |  Nil
```

Cette grammaire permet de construire une liste de valeurs, un mot de ce langage est par exemple :

$$Cons(1, Cons(2, Cons(3, Nil))). \tag{2.2}$$

La partie déclaration du langage Chocolat permet de définir ce genre de construction, voici l'exemple 2.2 écrit en chocolat :

```
class List
{
  case Cons(int, List);
  case Nil();
  ...
}
```

Note : Cons et Nil sont appelés les constructeurs de la classe List.

2.4 Un exemple complet en Chocolat

Essayons maintenant d'écrire la fonction reverse qui permet de retourner la liste, c-à-d obtenir à partir du mot 2.2

$$Cons(3, Cons(2, Cons(1, Nil))). \tag{2.3}$$

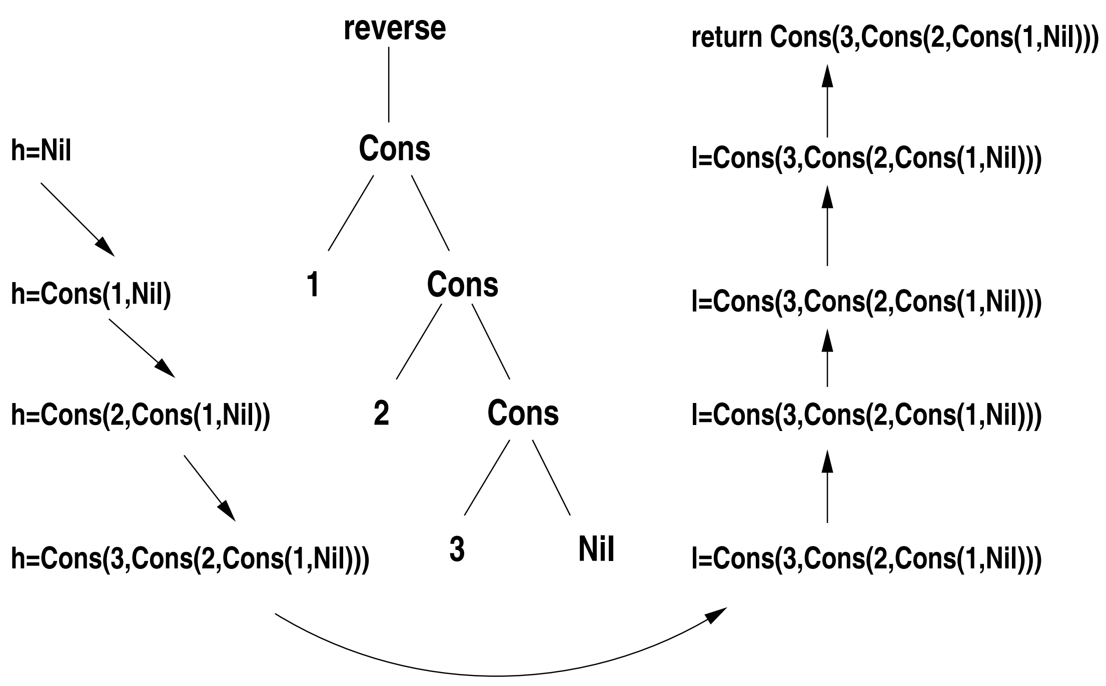


FIG. 2.3 – *Exemple3*

On écrira en chocolat :

```
class List
{
  case Cons(int, List);
  case Nil();

  List reverse()
  {
     List l,h;
     case Cons(value,tail):
       tail.h=Cons(value,this.h);
       this.l=tail.l;
     case Nil():
       this.l=this.h;
     case reverse:
       this.h=Nil;
       return this.l;
  }
}
```

Comme on le voit sur cette exemple, il y a plusieurs règles sémantiques associées à un constructeur, ce qui pose un certain nombre de problèmes, dans quel ordre doit-on exécuter les règles, existe-t-il au moins un ordre d'exécution possible?
Pour répondre à ces questions, il existe une série de tests[1] et autres usines à gaz dont le but final est de déterminer de façon statique, si possible, l'ordre dans lequel on doit exécuter chaque règle sémantique dynamiquement.

1. pour de plus amples informations, voir FNC, DNC et OAG.

Chapitre 3

Le langage Chocolat

3.1 Pourquoi un nouveau langage ?

Il existait déjà des outils de manipulation de grammaire attribuée mais ceux-ci ne sont pas integrés dans un langage. Chocolat permet de spécifier plusieurs grammaires attribuées et il est totalement intégré au langage puissant et objet Java.

3.2 La syntaxe du langage Chocolat

Le langage Chocolat est un langage objet, il est donc constitué d'un ensemble de classes. Chaque classe est composée de deux parties distinctes.

1. la déclaration des constructeurs.
2. la déclaration des fonctions [1].

3.2.1 La déclaration des Constructeurs

On déclare les constructeurs de la classe chocolat. Les constructeurs correspondent aux éléments permettant de créer une expression de la grammaire attribuée. On déclare de la façon suivante :

$$\textbf{case } Constructeur(arg_1, arg_2, \cdots, arg_n); \tag{3.1}$$

ou $Constructeur$ est le nom du constructeur. arg_n correspond au type du n^ieme argument du constructeur.
Le nombre d'argument peut être nul auquel cas le constructeur devra être déclaré de la façon suivante :

$$\textbf{case } Nil(); \tag{3.2}$$

Pour écrire par exemple le constructeur `Fork` qui prend un entier et deux arbres en paramètre, on écrira :

```
case Fork(int,Tree,Tree);
```

1. On pourrait ici parler de méthode car chaque fonction n'existe que dans l'espace de nommage de la classe.

3.2.2 La déclaration des Fonctions

Une fonction Chocolat contient en elle-même une serie de blocs constitués d'un pattern et de règles sémantiques associées au pattern. Les règles sémantiques décrivent comment calculer sur les attributs lors de la rencontre du pattern sur l'arbre d'analyse.
Prenons l'exemple d'un arbre binaire de recherche :

```
public class Tree2
{
  case Val(int);
  case Deux(int,Tree2_3,Tree2_3);

  public int find(int val)
  {
    boolean f;
    case Val(x):
      f=(val==x);
    case Deux(x,t1,t2):
      if (val==x)
        f=true;
      else
         if (val<x)
           f=t1.f;
         else
           f=t2.f;
    case find:
      return f;
  }
}

public class Main
{
  public final static main(String argv[])
  {
    Tree2 t=new Deux(4,new Deux(2,new Val(1),new Val(3)),new Val(5));
    System.err.println("find:"+t.find(2));
    System.err.println("find:"+t.find(6));
  }
}
```

Pendant l'analyse, si le constructeur trouvé est val alors on demande de tester la valeur de l'attribut du constructeur (x) avec la valeur recherchée (val). Si le constructeur trouvé est $Deux$, on teste si la valeur du constructeur (x) est égal à la valeur recherchée (val). Sinon on demande l'analyse soit sur l'arbre de gauche si val est inférieur, soit sur l'arbre de droite, si val est supérieur à x.

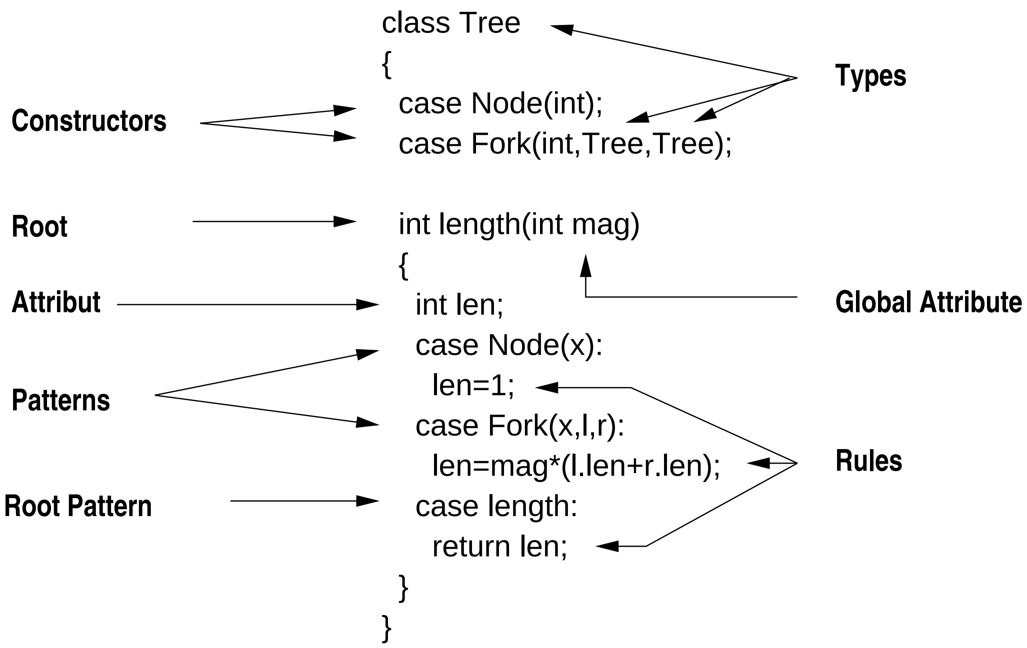


FIG. 3.1 – *La syntaxe de Chocolat*

3.3 Vue d'ensemble des différents éléments de Chocolat

Le langage Chocolat permet de spécifier des grammaires attribuées. Pour permettre des modifications et analyses ultérieures de la grammaire attribuée, le langage Chocolat est converti en plusieurs fichiers ASCII dans un langage intermédiaire, Cacao, qui lui représente une seule grammaire attribuée (voir figure 3.2 page 11).

Le langage intermédiaire Cacao permet de travailler séparemment sur chaque grammaire attribuée. Il existe une définition ASCII du langage Cacao pour permettre à des modules écrits dans n'importe quel langage de pouvoir effectuer des traitements sur la grammaire attribuée.

Il existe une librairie implementée en Java[2] dans le package cacao permettant de charger de décharger du code cacao en mémoire. De plus, ce package contient aussi une librairie permettant de générer du code Chocolat à partir de son équivalent Cacao.

Pour générer du Cacao à partir du Chocolat, un compilateur écrit en Java avec l'outil JavaCC est utilisé[3]. Le compilateur s'appuie sur le package Chocolat qui est grossièrement un sous-type du package Cacao. En effet, un objet du package Chocolat est un objet du package Cacao qui n'aurait pas subit de test de cohérence (*type checking*). Dans une version future, il est prévu d'écrire la partie analyse syntaxique du compilateur en Chocolat (*bootstrap*).

2. Java copyright Sun 1995-1997
3. JavaCC copyright Sun 1996-1997. JavaCC est un générateur de parseur LL(n).Il permet de générer un lexeur et un parseur en Java.

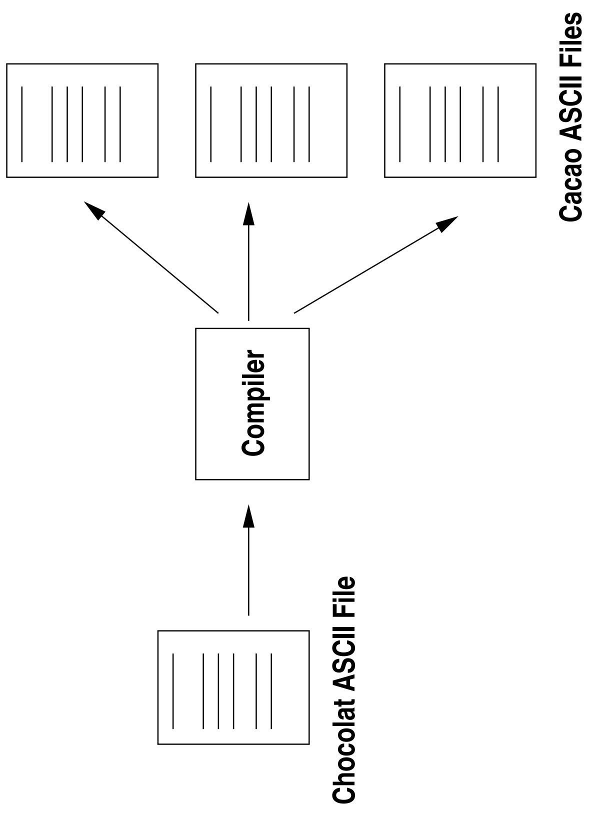


FIG. 3.2 – *Processus de création de fichier Cacao*

3.4 Le langage intermédiaire Cacao

3.4.1 Description de la structure statique

Un fichier ASCII au format Cacao contient un ensemble d'objets, briques de base de la grammaire attribuée :

- les objets `Type` qui representent aussi bien les types de base comme `int` ou `char` que les classes construites.
- les objets `Racine`, ils correspondent à une grammaire attribuée particulière, ils constituent aussi le point d'accès à l'arbre d'analyse.
- les objets `Globales`, sont des attributs qui sont associés à la grammaire en elle-même, leur valeur est globale à toute la grammaire attribuée.
- les objets `Constructeur`, ils permettent de construire les expressions reconnues par la grammaire attribuée.
- les objets `Attributs`, ils représentent les attributs de chaque noeud de l'arbre d'analyse (voir exemple 2.2).

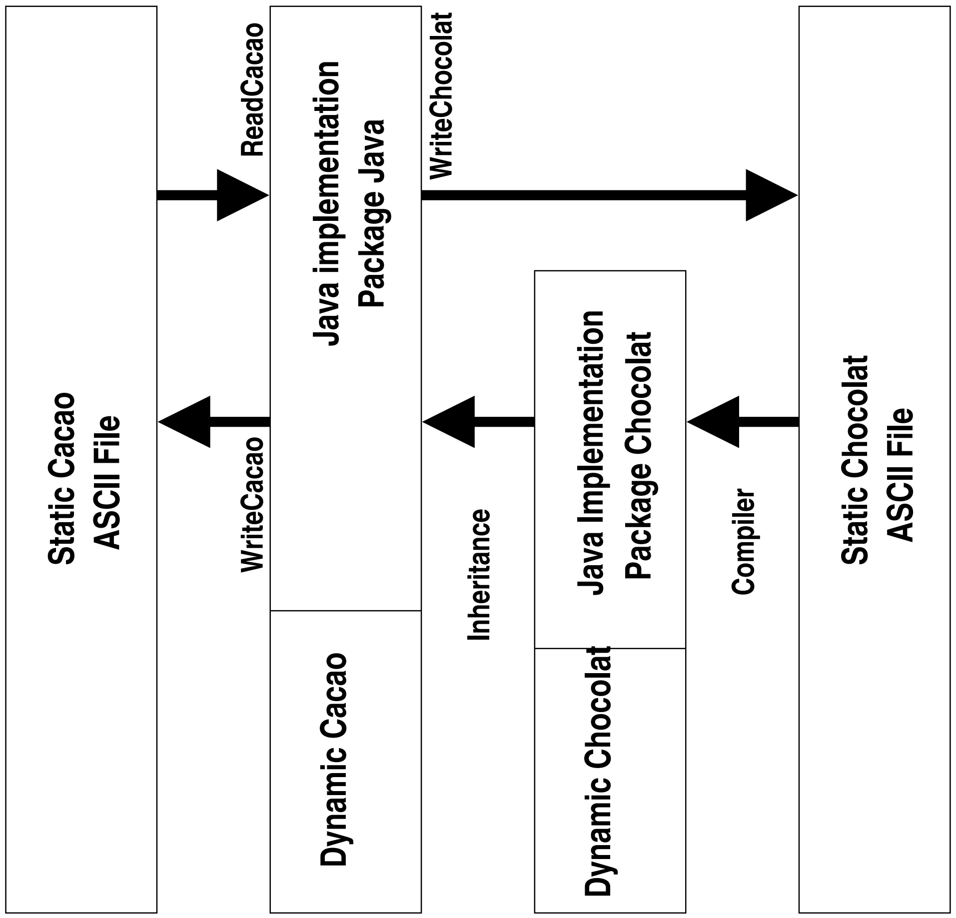


FIG. 3.3 – *Packages et relations entre-eux*

- les objets `Locales`, ce sont des attributs temporaires, qui ne sont déclarés que pour un même bloc de règle.
- les objets `Patterns`, ils sont composés de constructeurs et associent à ceux-ci une ou plusieurs règles.
- les objets `Règles`, ils sont composés d'objets sémantiques qui reflètent les opérations à effectuer sur les attributs.

En plus des objets de base, il existe les objets sémantiques qui permettent de décrire les actions à effectuer, les opérations entre les différents attributs et constructeurs.

- le `Champ`, il correspond à une occurence d'attribut.

- le `Constructeur`[4], il correspond à la construction d'un nouvel objet, d'une partie d'un mot du langage de la grammaire.
- la `Fonction`, il représente un appel de fonction, qui peut être implicite[5] ou explicite.
- le `IfThenElse` qui correspond à une expression différente suivant une variable booléenne.
- le `Attribut` qui correspond à un attribut.
- le `Locale` qui correspond à un attribut local.
- le `Globale` qui correspond à un attribut global.

3.4.2 La description de la structure dynamique

La structure dynamique est composée des mêmes objets que la structure statique. Ceux-ci ont été implémentés sous forme d'objets selon les concepts de la programmation orientée objet.

3.5 Le *package* Cacao

3.5.1 Le chargement des objets Cacao

Le chargement des objets Cacao s'effectue en plusieurs étapes :

1. On détermine tous les objets Cacao disponibles et on charge une instance de chacun.
2. On utilise un Tokenizer[6]
3. On crée une nouvelle instance de l'objet en fonction du code de l'objet Cacao correspondant.
4. On demande à l'instance obtenue de lire sur le Tokenizer.

Cette méthode quoiqu'un peu barbare garantit l'évolutivité de la procédure. En effet, chaque objet connaît sa représentation ASCII. Un changement de spécification du langage Cacao statique n'entraîne de modification que dans les objets concernés. De plus, on peut facilement rajouter des objets car la première étape de l'algorithme consiste à regarder tous les objets présents. Rajouter un objet Cacao ne nécessite aucune modification du code déjà écrit.

3.5.2 Le déchargement des objets Cacao

Chaque objet dynamique Cacao possède une méthode d'écriture de lui-même au format ASCII. L'algorithme est ici trivial, on prend chaque objet et lui demande de s'écrire sur un flux de sortie.

4. Attention, il ne faut pas confondre l'objet de base constructeur avec l'objet sémantique constructeur, l'objet de base déclare un constructeur tandis que l'objet sémantique représente l'appel un appel de fonction de création d'un objet.
5. L'expression sémantique $a+b$ s'écrit comme l'appel de fonction $operator\_plus(a,b)$.
6. Objet Java qui fait office de Lex, il transforme un flux de caractères en token.

3.5.3 La génération de Chocolat à partir de la structure dynamique

De même que chaque objet Cacao sait se coder sur un flux, celui-ci possède les méthodes pour créer le code ASCII de l'objet Chocolat correspondant. L'algorithme est ici un peu moins trivial car le format de Chocolat est un format objet, chaque objet Chocolat possède des attributs, il faut donc regrouper les objets en classe.

3.6 Le *package* Chocolat

3.6.1 Le compilateur de Chocolat vers Cacao

Le compilateur utilise comme squelette les fonctions générées par JavaCC (voir 3.3 page 10) à partir de la spécification de la grammaire de Chocolat. Lors de la création des objets Chocolat, il n'y a pas de vérification des types, la vérification est effectuée lors d'une seconde passe qui permet de transformer les objets Chocolat en leur équivalent Cacao.

Chapitre 4

Conclusion

A ce stade, mon implantation du compilateur de Chocolat n'est pas terminée. Elle est pour l'instant capable de reconnaître une spécification écrite en Chocolat et de générer l'arbre abstrait Cacao correspondant. De plus, j'ai implanté toute la bibliothèque de manipulation de Cacao : lecture, écriture et regénération du Chocolat. Le compilateur Chocolat est donc pour l'instant assez sommaire et ne permet pas, entre autre, l'interdépendance entre deux grammaires attribuées.

Il reste du travail mais je pense que la conception et les parties déjà implantées forme une base solide pour les développements futurs.

Le stage m'a apporté une façon différente de voir un domaine informatique et il m'a permis d'entrevoir comment un projet de recherche évolue au cours du temps. Je pense poursuivre mes études dans ce domaine qui me semble offrir plusieurs sujets de réflexion possibles.

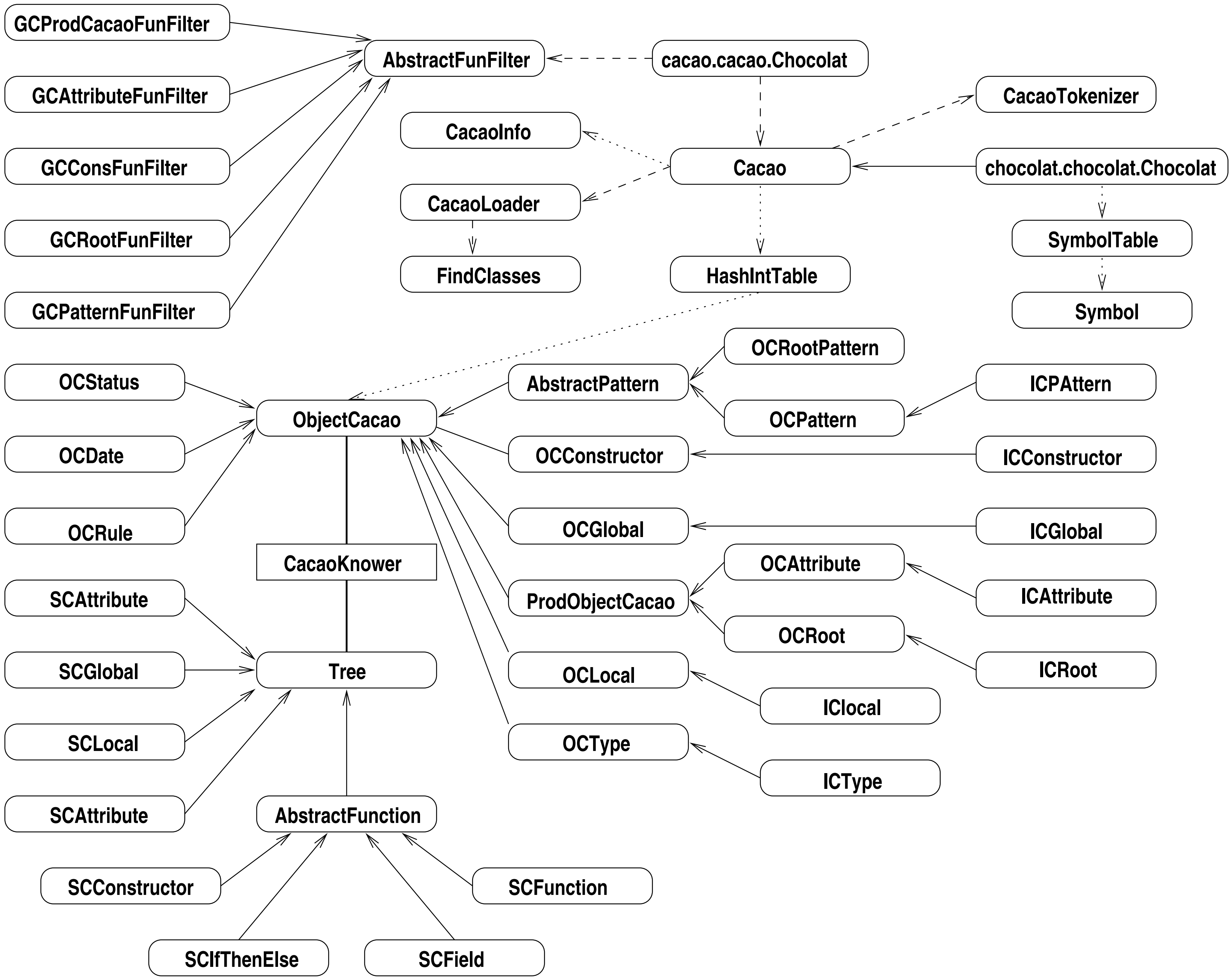
Bibliographie

[1] David FLANAGAN *Java in a Nutshell* O'Reilly & Associates, Inc. Feb 1996.

[2] Patrick NIEMEYER & Joshua PECK *Java par la pratique* O'Reilly & Associates, Inc. 1996.

[3] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel *Attribute grammars and functional programming deforestation* ftp://ftp.inria.fr/INRIA/Projects/oscar/FNC-2/publications/sas97.ps.gz, INRIA, April 1997.

[4] Didier Parigot. *Transformations, Évaluation Incrémentale et Optimisations des Grammaires Attribués : Le Système FNC-2* PhD thesis, Université de Paris-Sud, Orsay. 1988.


GCProdCacaoFunFilter
AbstractFunFilter
cacao.cacao.Chocolat
GCAttributeFunFilter
CacaoTokenizer
CacaoInfo
GCConsFunFilter
Cacao
chocolat.chocolat.Chocolat
CacaoLoader
GCRootFunFilter
SymbolTable
FindClasses
HashIntTable
GCPatternFunFilter
Symbol
OCRootPattern
OCStatus
AbstractPattern
ICPAttern
ObjectCacao
OCPattern
OCDate
OCConstructor
ICConstructor
OCRule
OCGlobal
ICGlobal
CacaoKnower
OCAttribute
SCAttribute
ProdObjectCacao
ICAttribute
OCRoot
SCGlobal
Tree
ICRoot
OCLocal
IClocal
SCLocal
OCType
ICType
SCAttribute
AbstractFunction
SCConstructor
SCFunction
SCIfThenElse
SCField

Annexe A

File Format for Attribute Grammar The Cacao Syntax

```
Main File

   One Cacao File contains one Attribute Grammar

   % _ means one space.
   % Spaces ans Newlines must be respected.
   % Extra spaces or newlines are forbidden.
   % \n means an unreprented newline.

   %date
   D_ddmmyyyy

   % status of cacao file
   S_[Status]

   % type definition
   {T_[Type Name]_[Type Nb]_{X|M}
   }*

   % root definition
   {Z_[Root Name]_[Root Nb]_[Result Type Nb]_[Tree Type Nb]_
   {G_[Global Name]_[Global Attribute Nb]_[Type Nb]_
   }*
   }*

   % constructor definition
   {C_[Cons Name]_[Cons Nb]_[LHS Type Nb]_[Size RHS]_{[RHS Type Nb]}^Size
   }*

   % attribute definition
```

```
{A_[Attr Name]_[Attr Nb]_[Attr Type]_[Non-Term Type]_{S|H|U}{P|N}
}*

% pattern definition, local attributes and rules
{<pattern>
{L_[Attr Local Name]_[Attr Nb]_[Attr Type]_<exp>
}*

<system>
}*

<end-of-file>


where <system> is as follows :

<system> ::= {R_<rule>\n}*


where <rule> is as follows :

<rule> ::= C_[Position]_[Attr_Nb]_<exp>     % constructor pattern
        |  R_<exp>                          % root pattern
        |  D_<exp>\n<view>\n<system>\n<view>\n<system>


where <view> is as follows :

 <view> ::= {V_{+_[New Local Position]_<exp>\n}|{-_[Position]}\n}*


where <exp> is as follows :

<exp> ::= M_<exp>_[Attribute Nb]             % field
       | C_[Cons Nb]_[Size]_{<exp>}^Size     % new Constructor
       | F_[Function Nb]_[Size]_{<exp>}^Size % Function
       | I_<exp>_<exp>_<exp>                 % If Then Else
       | X_[Position Nb]                     % Non-Terminal
       | L_[Attr Nb]                         % Local Attribute
       | G_[Attr Nb]                         % Global Attribute


where <pattern> is as follows :

%pattern matching
<pattern> ::= P_[Cons Nb]_[Size]_{<pattern>}^Size
            | Q_[Root Nb]                         % Root Pattern Definition
```

```
              | X

   % usefull properties :
   % <exp> always terminates with _
   % [identifier] or [integer] are always followed by _
   % <system> and <view> always terminates with \n
```

Functions File

```
   % external function code
   {[Function NB]_[Language]_[Size]_{char}^Size
   }*
```

Annexe B

The BNF of Chocolat

```
Description BNF de la grammaire de chocolat

 Introduction

        Le but de chocolat est de pouvoir specifier une grammaire attribuee orientee
        objet. Pour cela, la grammaire de chocolat est un "super-set" de java.

 Description d'une Classe Chocolat

        Globalement, une classe chocolat est constituee de deux parties :
              La declaration des constructeurs ;
              la description des fonctions utilisants les constructeurs.

    Description BNF

        Chocolat = Classes

        Classes = Classes Classe
               |  Classe

        Classe = "class" ID '{' Body '}'

        Body = Declare Functions

 La declarations des Constructeurs

        Un constructeur chocolat joue le meme role qu'un constructeur dans un langage objet, il peut de construire une instance de la
        classe.
        Contrairement aux langages objets, un constructeur ne possede pas le meme nom que la classe et il existe souvent plusieurs
        constructeurs pour une classe doneee.

        Exemple :
```

```
class List
{
  case Nil();
  case Cons(int a,List tail);
  ...
}
```

Cette exemple construit une liste d'entier, on pourra construire la liste [1,3,2] avec la sequence suivante : Cons(1,Cons(3,Cons(2,Nil))).

Note : On peut nommer les arguments d'un constructeur (Cons(int a,List tail)) ou ne pas les nommer (Cons(int,List)), il faudra alors nommer le constructeur lors du "case" a l'interieur d'une fonction.

Description BNF

```
Declare = Declare Constructeur
       |

Constructeur = "case" ID '(' Params ')' ';'

Params = Fields
     |   Types
     |

Fields = Fields ',' Field
     | Fields

Field = ID ID

Types = Types Type
     |  Type

Type = ID
```

La declaration des Fonctions

Une fonction est constitue de parametres d'appels, d'une valeur de retour, d'attributs locaux et d'un code pour chaque constructeurs.

Description BNF

```
Functions = Functions Function
        |   Function

Function = Type ID '(' Fields ')'
           '{' Attributs Matches '}'
```

```
Attributs = Attributs Attribut
        |   Attribut

Attribut = Type Ids ';'

Ids = Ids ',' Id
   |  Id

Id = ID
```

La specification du code associe aux constructeurs

Description BNF

```
Matches = Matches Match
       |  Match

Match = "case" Pattern ':' Exprs

Pattern = ID '(' Vars ')'
       |  ID '(' ')'
       |  ID

Vars = Vars ',' Var
   |   Var

Var = ID (' Vars ')'
   |  ID '(' ')'
   |  ID
```

Declaration d'un code associe a un constructeurs

Decription BNF

```
Value = Object
    |  VALUE

Object = Object '.' Identif
      |  Identif

SimpleObject = SimpleObject '.' ID
          |   ID

Identif = ID '(' Variables ')'
      |   ID '(' ')'
      |   ID
```

```
Variables = Variables ',' Variable
        | Variable

Variable = ID

Exprs = Exprs Expr
    |

Expr = Object '=' Mult ';'

Mult = Mult '*' Add
   |   Mult '/' Add
   |   Mult '%' Add
   |   Add

Add = Add '+' Value
   |  Add '-' Value
   |  Value
```