EGG : Guide de l'utilisateur

Marcel Gandriau [1]

15 avril 2005

[1]Institut de Recherche en Informatique de Toulouse CNRS (UMR 5055) - INPT - UPS

Table des matières

Listings

Chapitre 1

Préface

Dans ce chapitre
- Objectifs
- Public cible
- Prérequis
- Contacts
- L'IRIT

1.1 Objectifs

EGG est un générateur de compilateur en Java destiné à l'enseignement de la compilation. Il est utilisé à l'ENSEEIHT depuis le début de son développement en 1994.

1.2 Public cible

Ce manuel s'adresse aux utilisateurs de l'outil EGG, à savoir :
- les enseignants de la compilation ;
- les chercheurs spécialisés dans le développement de langage informatique ;
- les informaticiens ;
- les étudiants concernés par la compilation.

1.3 Prérequis

L'utilisation de EGG est simple, sous réserve de quelques notions de base en compilation, en particulier celle de grammaires attribuées.

1.4 Contacts

Contact scientifique
Marcel.Gandriau@enseeiht.fr

1.5 Transfert industriel

Daniel.Ventre@irit.fr - 33 (0)5 61 55 63 04

1.6 L'IRIT

L'Institut de Recherche en Informatique de Toulouse (IRIT) est une unité associée au Centre National de la Recherche Scientifique (CNRS), à l'Institut National Polytechnique de Toulouse (INPT) et à l'Université Paul Sabatier (UPS). Environ 335 personnes travaillent à l'IRIT dont 265 chercheurs et enseignants chercheurs (parmi lesquels 115 doctorants) et 78 ingénieurs, techniciens et administratifs.

Les recherches de l'IRIT couvrent l'ensemble des domaines où l'informatique se développe aujourd'hui, que ce soit dans son axe propre, de l'architecture des machines au génie logiciel et aux réseaux, comme dans son extension les plus contemporaines : intelligence artificielle et systèmes cognitifs, interaction multimedia homme-système, analyse et synthèses d'images.

L'IRIT à l'UPS

118 route de Narbonne - 31062 Toulouse Cedex 4 Tél. 05 61 55 67 65 / fax 05 61 55 62 58

L'IRIT à ENSEEIHT

2 rue Camichel - 31071 Toulouse Cedex Tél. 05 61 62 78 62 / fax 05 61 58 82 09

1.7 Participants

EGG est la suite de GEN6 qui a bénéficié du travail d'étudiants de l'ENSEEIHT. Parmi eux Gilles Gouaillardet, Mathieu Mollin, Raphael et David-Olivier Saban. Plus tous les testeurs plus ou moins volontaires que sont les étudiants de 2ième année informatique qui ont subi le projet de Traduction des Langages...

Chapitre 2

Présentation de EGG

Dans ce chapitre
- Généralités
- Principe de fonctionnement
- Concept - notion de grammaire attribuée

2.1 Généralités

2.1.1 Objectif

EGG est un générateur de compilateurs (en Java). Le développement de EGG en EGG/Java assure sa portabilité et son évolution sur diverses plates-formes. Ce type de développement a également constitué une autovalidation.

L'utilisation de EGG est simple sous réserve de quelques connaissances de base en compilation, en particulier la notion de grammaires attribuées. EGG est basé sur une analyse descendante (grammaires LL(k)).

Les actions sémantiques sont écrites dans un langage permettant de nombreux contrôles sur l'utilisation des attributs sémantiques.

2.1.2 Exemples de réalisation

EGG a été utilisé dans les contextes suivants :
- Minis compilateurs C, Java développés par les étudiants de l'ENSEEIHT en moins d'un mois ;
- Interface ml/prolog pour des travaux pratiques de sémantique ;
- IDL pour des travaux pratiques de système ;
- Logiciel de description de maisons à ossature bois nécessitant une représentation 3D avec Autocad. Un langage de description de la structure d'une maison (murs, planchers, toiture, ...) est écrit. Cette description est ensuite traduite automatiquement en commandes 3D Autolisp (langage interne d'Autocad), rendant aisée l'évolution de la description ;
- Analyseur XML ;
- EGG lui-même.

2.2 Principe de fonctionnement

EGG prend en entrée un fichier décrivant la grammaire du compilateur à engendrer pour créer un analyseur lexical et un analyseur syntaxique et sémantique descendant. Le fonctionnement de EGG s'inscrit dans le schéma suivant :

2.2.1 Schématisation

2.3 Concepts

2.3.1 Notion de grammaire attribuée

La sémantique du langage est décrite par une grammaire attribuée.

Il s'agit d'associer, si nécessaire à chaque symbole de la grammaire une ou plusieurs informations (attributs sémantiques) qui seront mises à jour par des instructions (actions sémantiques) insérées entre les symboles de la partie droite d'une règle de production.

Exemples d'attributs sémantiques

- une table des symboles
- le niveau d'imbrication d'une déclaration
- un descripteur de type, ou une liste de types
- un descripteur de fonction
- le code engendré

Exemples d'actions sémantiques

- l'insertion ou la recherche d'un identificateur dans une table des symboles
- l'affectation ou l'utilisation d'un niveau d'imbrication
- le contrôle du type d'un paramètre d'une fonction
- la gestion des variables locales d'une fonction
- la mise à jour du code à engendrer pour une expression

2.3.2 Attributs sémantiques

Suivant le sens de transmission dans l'arbre syntaxique associé à une phrase du source d'un attribut sémantique, on parlera d'attribut synthétisé ou hérité :

- vers le haut pour une synthèse : attribut synthétisé,
- vers le bas ou le coté pour une référence : attribut hérité.

Il faut noter qu'un terminal ne peut avoir que des attributs synthétisés.

2.3.3 Contraintes de construction

Les actions sémantiques doivent respecter quelques règles strictes pour que la mise à jour ou la référence à un attribut soit possible et correcte pendant le parcours de l'arbre. On peut voir ces règles comme des contraintes fortes, mais elles garantissent la correction de l'évaluation de la sémantique du langage.

Soit `A -> X1 X2 ... Xn` une règle de production pour le non-terminal A. Les actions sémantiques doivent respecter les contraintes suivantes :

- Tout attribut synthétisé de A, doit être initialisé fois avant la fin de l'analyse de la règle de production.
- Tout attribut hérité d'un symbole de la partie droite doit etre affecté dans une action située à gauche de ce symbole, de manière à garantir que sa valeur est connue au moment de son utilisation (dans un sous-arbre ou un arbre frère). Si l'initialisation du symbole est effectuée dans une branche d'un if, il devra également être initialisé dans toutes les autres branches et notamment dans la branche else.

En conséquence, On ne référencera jamais un attribut de symbole dans une action située à la gauche du symbole.

Chapitre 3

Exemple de réalisation

Dans ce chapitre
- Contexte de l'exemple
- Définition et test de la syntaxe du langage
- Traitement des identificateurs avec une Table des Symboles
- Contrôle de type
- Génération du code TAM

3.1 Contexte de l'exemple

3.1.1 Présentation de l'exemple

Dans ce chapitre vous allez développer un exemple complet de compilateur en passant par quatre étapes successives.

Pour chacune de ces étapes, plusieurs fichiers seront développés :
- le fichier POLTAM.m : fichier descripteur du langage
- le fichier POLTAM.fr des messages d'erreurs.
- le fichier makefile : fichier de gestion de la compilation complète du programme,
- le fichier INFO.java : un fichier de classe JAVA,
- les fichiers de test ex0 (correct) et ex1 (comportant des erreurs).

3.1.2 Présentation de la machine virtuelle TAM et du langage POLTAM

La machine virtuelle TAM est une machine à pile (un peu comme la JVM), sans registre de données, dont l'assembleur dispose de 16 instructions pour manipuler la pile, la mémoire et le contrôle de l'exécution. On trouvera en annexe une description plus précise de la machine TAM qui peut être utilisée pour implanter de vrais compilateurs.

Le langage POLTAM permet de déclarer et d'utiliser des variables de type entier et caractère.

La syntaxe de POLTAM (non LL(1)) est la suivante :

Listing 3.1 – La syntaxe de POLTAM

```
PROG -> DS IS
DS ->
DS -> D DS
TYPE -> int
TYPE -> char
D -> var ident deuxpts TYPE pv
IS ->
IS -> I IS
I -> ident aff E pv
E -> T OPADD E
E -> T
OPADD -> opplus
OPADD -> opmoins
T -> T OPMUL F
T -> F
OPMUL -> opmult ;
OPMUL -> opdiv ;
F -> parouv E parfer
F -> ident
F -> entier
F -> caractere
```

comme dans l'exemple ;

Listing 3.2 – Un exemple de programme en POLTAM

```
var x : int ;
var y : int ;
x := 9;
x := 3 + x ;
y := x + 1;
```

La conception du compilateur avec EGG comporte deux phases :

- définition de la syntaxe du langage ;
- élaboration de l'analyse sémantique (gestion des tables de symboles, contrôle de type, génération du code).

3.1.3 Développement du compilateur

Ainsi, le développement du compilateur avec EGG/java sera décrit en 4 temps :

- la définition et test de la syntaxe
- le traitement des identificateurs avec une Table des Symboles (TDS)
- le contrôle de type
- la génération de code TAM

3.2 Définition et test de la syntaxe du langage

3.2.1 Introduction

EGG n'accepte que des grammaires LL(k) qui en particulier sont non récursives à gauche. Le calcul des symboles directeurs permet de s'assurer que la grammaire est bien LL(k), en fait fortement LL(k). Après transformations on obtient une nouvelle grammaire utilisable par EGG que l'on utilisera dans toute la suite.

3.2.2 Les fichiers

Grammaire

Le fichier POLTAM.m décrivant la syntaxe est le suivant :

Listing 3.3 – Version LL(1) de la grammaire de POLTAM

```
-- Syntaxe de POLTAM : LL(1)
space     separateur     is        "[\n\t ]+";
sugar     opplus         is        "\+";
sugar     opmoins        is        "\-";
sugar     opmult         is        "\*";
sugar     opdiv          is        "\/";
sugar     parouv         is        "\(";
sugar     parfer         is        "\)";
sugar     deuxpts        is        ":";
sugar     pv             is        ";";
sugar     aff            is        ":=";
sugar     var            is        "var";
sugar     int            is        "int";
sugar     char           is        "char";
term      entier         is        "[0-9]+";
term      caractere      is        "\'([^\']|\\\')\'";
term      ident          is        "[a-z_]+";

-- REGLES DE PRODUCTION
----------programme----------
PROG -> DS IS ;
----------declarations----------
DS -> ;
DS -> D DS ;
TYPE -> int ;
TYPE -> char ;
D -> var ident deuxpts TYPE pv ;
----------instructions----------
IS -> ;
IS -> I IS ;
I -> ident aff E pv ;
----------expressions----------
```

```
E -> T TX ;
T -> F FX ;
TX -> OPADD T TX ;
TX -> ;
OPADD -> opplus ;
OPADD -> opmoins ;
F -> parouv E parfer ;
F -> ident ;
F -> entier ;
F -> caractere ;
FX -> OPMUL F FX ;
FX -> ;
OPMUL -> opmult ;
OPMUL -> opdiv ;
end
```

Les lignes 4 à 19 décrivent les terminaux de la grammaire (mots-cle, identificateurs, entiers, ...). Un terminal est référencé dans les règles de production par son nom et associé à un automate construit à partir de son expression régulière.

Les lignes 24 à 49 décrivent les non-terminaux par leurs règles de production. L'ordre des règles n'est pas important mais il est conseillé de regrouper les règles décrivant un même non-terminal.

Messages

Le fichier des messages d'erreurs

Listing 3.4 – Messages d'erreur pour la syntaxe

```
// messages obligatoires
A_00, "Symbole inattendu ^1 au lieu de ^2." ,2 ;
A_01, "Fin de source attendue pres de ^1.",1;
// messages specifiques a partir d'ici
```

Ce fichier contient les différents messages d'erreur du compilateur engendré. Chaque ligne correspond à un message avec un numero de message, un texte avec des 'trous' pour personnaliser le message (de la forme $\hat{x}$) et le nombre de trous. Il est possible d'avoir plusieurs fichiers de messages dans des langues différentes.

Génération

Le fichier makefile pour automatiser la génération est :

Listing 3.5 – Fichier de génération

```
#-------------------------------------------------------------
# la grammaire (voir src)
XLANG=POLTAM
#-------------------------------------------------------------
# le package ou est cree le compilateur (voir src et class)
```

```
PACKAGE=poltam
#-----------------------------------------------------------------
# la langue du fichier de message (voir pmess)
MLANG=fr
#-----------------------------------------------------------------
# repertoires contenant egg et mess
EDIR=$(HOME)/EGG/egg/lib
MDIR=$(HOME)/EGG/mess/lib
#EDIR=/usr/local/gen6/lib
#MDIR=/usr/local/gen6/lib
# les jars associes
GJAR=$(EDIR)/egg.jar:.
MJAR=$(MDIR)/mess.jar:$(GJAR)
#-----------------------------------------------------------------
# java, javac, jar
JDIR=/usr/bin
#-----------------------------------------------------------------
all : src javalex pmess class

src :
        $(JDIR)/java -cp $(GJAR) egg.java.EGGC $(XLANG).m -l
    java -s jlex -p $(PACKAGE).java -mkdir -vs "`date`"

javalex :
        $(JDIR)/java -cp $(GJAR) JL.Main $(PACKAGE)/java/JLEX_$(
    XLANG)

pmess:
        $(JDIR)/java -cp $(MJAR) mess.java.MESSC $(XLANG).$(
    MLANG) -l java -d poltam/java

class :
        $(JDIR)/javac -classpath $(GJAR) $(PACKAGE)/java/*.java

clean :
        rm -rf $(PACKAGE)
```

Tests

Deux fichiers d'exemples :
Fichier EX0

Listing 3.6 – Exemple de syntaxe : ex0

```
var x : int ;
var y : int ;
x := 9;
x := 3 + x ;
```

```
y := x + 1;
```

Fichier EX1

Listing 3.7 – Exemple de syntaxe : ex1

```
var x : int;
var y ;
x := 9;
x := 3 + x ;
y:= x + 1;
```

Script d'exécution

Pour exécuter le compilateur généré sur l'exemple ex0, le script 'poltamc' permet de lancer plus facilement l'appel de java sur la classe principale du compilateur engendré.

Listing 3.8 – Script d'exécution

```
#!/bin/sh
#-----------------------------------------------------------------
# repertoires contenant egg et mess
EDIR=$HOME/EGG/egg_auto/lib
# les jars associes
GJAR=$EDIR/egg.jar:.
#-----------------------------------------------------------------
# java, javac, jar
JDIR=/usr/bin
#-----------------------------------------------------------------
$JDIR/java -cp $GJAR poltam.java.POLTAMC $*
```

3.2.3 Exécution du compilateur

`poltamc ex0`
L'absence de message signifie qu'il n'y a pas d'erreur de syntaxe.
`poltamc ex1`

Listing 3.9 – exécution de ex1

```
version lun avr 11 13:10:33 CEST 2005
2 : Symbole inattendu ; au lieu de deuxpts.
```

Le message d'erreur signale une erreur de syntaxe à la ligne 2 car il manque le ' :'.
Vous pouvez passer à l'étape suivante : la gestion de la table des symboles.

3.3 Traitement des identificateurs

3.3.1 Introduction

Il s'agit ici de ranger et référencer les identificateurs et les informations les concernant dans une Table Des Symboles (TDS).

Dans cette première version on se contente de conserver le nom de la variable dans la table, car on ne prend pas encore en compte la notion de type.

On associe donc aux règles de production utilisant 'ident' un traitement pour ranger ou contrôler les noms de variables.

Des actions sémantiques mettant à jour l'attribut sémantique 'table' sont donc associées aux règles directement concernées, puis aux règles nécessitant une transmission de la table.

L'attribut `table` est donc déclaré (lignes 6 à 9) pour les symboles
`DS, IS, D, I, E, T, TX, F, FX`

Les terminaux ont par défaut un attribut `txt` qui est la chaine de caractères décodée par l'analyseur lexical.

L'acces à un attribut `a` d'un symbole `X` se fait par `X^a`.

Le nom d'une action sémantique commence par un #.

Le code de l'action est donné après la règle de production concernée.

3.3.2 Les fichiers

Grammaire

Fichier POLTAM.m

Listing 3.10 – Grammaire attribuée pour la TDS

```
1  ------------------------------------------------------
   -- Traduction d'expressions en TAM
   -- avec gestion de table des symboles
   ------------------------------------------------------
5
   inh     table : TDS
       for
           DS, IS, D, I,
           E, T, TX, F, FX;
10
   space    separateur     is        "[\n\t ]+";
   sugar    opplus         is        "\+";
   sugar    opmoins        is        "\-";
   sugar    opmult         is        "\*";
15 sugar    opdiv          is        "\/";
   sugar    parouv         is        "\(";
   sugar    parfer         is        "\)";
   sugar    deuxpts        is        ":";
   sugar    pv             is        ";";
20 sugar    aff            is        ":=";
   sugar    var            is        "var";
   sugar    int            is        "int";
   sugar    char           is        "char";
   term     entier         is        "[0-9]+";
25 term     caractere      is        "\'([^\']|\\\')\'";
```

```
term     ident          is        "[a-z_]+";

PROG -> #table DS IS ;
#table {
    local
        t : TDS;
    do
        -- creation de la table des symboles
        --new t.make();
        t := new TDS();
        DS^table := t;
        IS^table := t;
    end
}

DS -> #fin ;
#fin {
    local
    do
        write DS^table;
    end
}


DS -> #table D DS ;
#table {
    local
    do
        D^table := DS^table;
        DS1^table := DS^table;
    end
}

TYPE -> int ;

TYPE -> char ;

D -> var ident deuxpts TYPE pv #inserer ;
#inserer {
    local
        i : INFO;
    do
        -- rechercher l'ident
        i := D^table.chercher(ident^txt);
        if (i /= nil) then
            error(P_00, ident^txt);
        end
```

```
            new i.make();
            call D^table.inserer(ident^txt, i);
            -- inserer ident^txt dans D^table
        end
    }

    IS -> ;

    IS -> #table I IS ;
    #table {
        local
        do
            I^table := IS^table;
            IS1^table := IS^table;
        end
    }

    I -> ident aff #table E pv ;
    #table {
        local
            i : INFO;
        do
            -- rechercher l'ident
            i := I^table.chercher(ident^txt);
            if (i = nil) then
                error(P_01, ident^txt);
            end
            -- transmettre la table
            E^table := I^table;
        end
    }

    E -> #table T #trans TX ;
    #table {
        local
        do
            T^table := E^table;
        end
    }

    #trans {
        local
        do
            TX^table := E^table;
        end
    }
```

```
T -> #table F #trans FX ;
#table {
    local
    do
        F^table := T^table;
    end
}

#trans {
    local
    do
        FX^table := T^table;
    end
}

TX -> OPADD #table T #trans TX ;
#table {
    local
    do
        T^table := TX^table;
    end
}

#trans {
    local
    do
        TX1^table := TX^table;
    end
}

TX -> ;

OPADD -> opplus ;

OPADD -> opmoins ;

F -> parouv #table E parfer ;
#table {
    local
    do
        E^table := F^table;
    end

}

F -> ident #table ;
#table {
```

```
        local
            i:INFO;
        do
            i := F^table.chercher(ident^txt);
            if (i = nil) then
                error(P_01, ident^txt);
            end
        end
}

F -> entier ;

F -> caractere ;

FX -> OPMUL #table F #trans FX ;
#table {
        local
        do
            F^table := FX^table;
        end
}
#trans {
        local
        do
            FX1^table := FX^table;
        end
}

FX -> ;

OPMUL -> opmult ;

OPMUL -> opdiv ;

end
```

Messages

Le fichier des messages d'erreurs

Listing 3.11 – Les messages d'erreur pour la TDS

```
// messages obligatoires
A_00, "Symbole inattendu ^1 au lieu de ^2." ,2 ;
A_01, "Fin de source attendue pres de ^1.",1;
// messages specifiques a partir d'ici
P_00, "La variable ^1 est àdej édfinie.",1;
P_01, "La variable ^1 est inconnue.",1;
```

Génération

Fichier makefile modifié pour prendre en compte l'utilisation des classes implantant la table des symboles (TDS et INFO).

Listing 3.12 – Fichier de génération

```
#-----------------------------------------------------------------
# la grammaire (voir src)
XLANG=POLTAM
#-----------------------------------------------------------------
# le package ou est cree le compilateur (voir src et class)
PACKAGE=poltam
#-----------------------------------------------------------------
# la langue du fichier de message (voir pmess)
MLANG=fr
#-----------------------------------------------------------------
# repertoire contenant les bibliotheques
ATT=att_java
# repertoires contenant egg et mess
EDIR=$(HOME)/EGG/egg/lib
MDIR=$(HOME)/EGG/mess/lib
#EDIR=/usr/local/gen6/lib
#MDIR=/usr/local/gen6/lib
# les jars associes
GJAR=$(EDIR)/egg.jar:.
MJAR=$(MDIR)/mess.jar:$(GJAR)
#-----------------------------------------------------------------
# java, javac, jar
JDIR=/usr/bin
#-----------------------------------------------------------------
all : src javalex pmess att class

src :
        $(JDIR)/java -cp $(GJAR) egg.java.EGGC $(XLANG).m -l
    java -s jlex -p $(PACKAGE).java -mkdir -a att.cfg -vs "`date`
    " -auto

javalex :
        $(JDIR)/java -cp $(GJAR) JL.Main $(PACKAGE)/java/JLEX_$(
    XLANG)

pmess:
        $(JDIR)/java -cp $(MJAR) mess.java.MESSC $(XLANG).$(
    MLANG) -l java -d poltam/java

class :
        $(JDIR)/javac -classpath $(GJAR) $(PACKAGE)/java/*.java
```

```
att :
40          $(JDIR)/javac -classpath $(GJAR) $(ATT)/*.java

clean :
            rm -rf $(PACKAGE)
```

Classes

Fichier de classe : INFO.java

Listing 3.13 – La classe INFO.java pour la TDS

```
//-------------------------------------------------------
// INFO la classe representant une variable
//-------------------------------------------------------
package att_java ;
5

public class INFO {
  // constructeur
  public INFO (){
    }
10  // affichage
  public String toString ( ) {
    return "VAR ";
    }
  }
```

3.3.3 Exécution du compilateur

Pour exécuter le compilateur généré sur l'exemple ex0 tapez la commande suivante : `poltamc ex0`

Listing 3.14 – tds : exécution de ex0

```
version lun avr 11 13:12:39 CEST 2005
y : VAR
x : VAR
```

Il n'y a pas d'erreur sur l'utilisation des variables.
`poltamc ex1`

Listing 3.15 – tds : exécution de ex1

```
version lun avr 11 13:12:39 CEST 2005
x : VAR
4 : La variable y est inconnue.
```

Le message indique qu'à la ligne 4 la variable 'y' n'a pas été déclarée.
Vous pouvez passer à l'étape suivante : le contrôle de type.

3.4 Contrôle de type

3.4.1 Introduction

Dans cette étape, vous allez décrire le contrôle de type :
Il faut ajouter au fichier précédent (gestion de la TDS) tout ce qui concerne les types.

- Création des types (associée au règles décrivant le non-terminal `TYPE`.
- Modification de la classe INFO pour ajouter le type dans le descripteur de variable.
- Modification de la gestion de la TDS : déclaration d'une variable avec son type.
- Réalisation des contrôles : au niveau de l'instruction d'affectation, et au niveau des opérations.

3.4.2 Les fichiers

Grammaire

Fichier POLTAM.m

Listing 3.16 – La grammaire attribuée pour le contrôle de type

```
-------------------------------------------------------
-- Traduction d'expressions en TAM
-- avec gestion de table des symboles
--  et controle de types
-------------------------------------------------------

inh     table : TDS for
        DS, IS, D, I,
        E, T, TX, F, FX;

syn     type : INTEGER for
        E, T, TX, F, FX;

inh     htype : INTEGER for
        TX, FX;

syn     taille : INTEGER for
        TYPE;

space   separateur      is      "[\n\t ]+";
sugar   opplus          is      "\+";
sugar   opmoins         is      "\-";
sugar   opmult          is      "\*";
sugar   opdiv           is      "\/";
sugar   parouv          is      "\(";
sugar   parfer          is      "\)";
sugar   deuxpts         is      ":";
sugar   pv              is      ";";
sugar   aff             is      ":=";
```

```
sugar     var             is          "var";
sugar     int             is          "int";
sugar     char            is          "char";
term      entier          is         "[0-9]+";
term      caractere       is         "\'([^\']|\\\')\'";
term      ident           is          "[a-z_]+";

PROG -> #table DS IS ;
#table {
    local
        t : TDS;
    do
        -- creation de la table des symboles globale
        t := new TDS();
        DS^table := t;
        IS^table := t;
    end
}

DS -> #fin;
#fin {
    local
    do
        write DS^table;
    end
}

DS -> #table D DS ;
#table {
    local
    do
        D^table := DS^table;
        DS1^table := DS^table;
    end
}

TYPE -> int #type ;
#type{
  local
  do
    TYPE^taille := 4;
  end
  }

TYPE -> char #type ;
#type{
  local
```

```
    do
      TYPE^taille := 1;
    end
80  }

  D -> var ident deuxpts TYPE pv #inserer ;

  #inserer {
85      local
            i : INFO;
        do
            -- rechercher l'ident
            i := D^table.chercher(ident^txt);
90          if i /= nil then
               error(P_00, ident^txt);
            end
            -- creer une info
            i := new INFO(TYPE^taille);
95          -- inserer ident^txt dans D^table
            call D^table.inserer(ident^txt, i);
        end
  }

100 IS -> ;

  IS -> #table I IS ;
  #table {
        local
105     do
            I^table := IS^table;
            IS1^table := IS^table;
        end
  }
110
  I -> ident aff #table E pv #type ;
  #table {
        local
        do
115         -- transmettre la table
            E^table := I^table;
        end
  }

120 #type {
        local
            i : INFO;
        do
```

```
            -- rechercher l'ident
125         i := I^table.chercher(ident^txt);
            if i = nil then
               error(P_01, ident^txt);
            end
            if i.getType() /= E^type then
130            error(P_02, i.getType(), E^type);
            end
        end
    }

135 E -> #table T #trans TX #type ;
    #table {
        local
        do
            T^table := E^table;
140     end
    }

    #trans {
        local
145     do
            TX^table := E^table;
            TX^htype := T^type;
        end
    }
150
    #type {
        local
        do
            E^type := TX^type;
155     end
    }

    T -> #table F #trans FX #type ;
    #table {
160     local
        do
            F^table := T^table;
        end
    }
165
    #trans {
        local
        do
            FX^table := T^table;
170         FX^htype := F^type;
```

```
    end
}

#type {
    local
    do
        T^type := FX^type;
    end
}

TX -> OPADD #table T #trans TX #type ;
#table {
    local
    do
        T^table := TX^table;
    end
}

#trans {
    local
      t : INTEGER;
    do
        TX1^table := TX^table;
        if TX^htype = T^type then
          TX1^htype := T^type;
        else
          error(P_02, TX^htype, T^type);
        end
    end
}

#type {
    local
    do
        TX^type := TX1^type;
    end
}

TX -> #type ;
#type {
    local
    do
        TX^type := TX^htype;
    end
}

OPADD -> opplus ;
```

```
OPADD -> opmoins ;

F -> parouv #table E parfer #type ;
#table {
    local
    do
        E^table := F^table;
    end

}
#type {
    local
    do
        F^type := E^type;
    end
}

F -> ident #gen ;
#gen {
    local
        i:INFO;
    do
        i := F^table.chercher(ident^txt);
        if i = nil then
           error(P_01, ident^txt);
        end
        F^type :=  i.getType();
    end
}

F -> entier #type ;
#type {
    local
    do
        F^type := 4;
    end
}

F -> caractere #type ;
#type {
    local
    do
        F^type := 1;
    end
}
```

```
265 FX -> OPMUL #table F #trans FX #type ;
    #table {
        local
        do
            F^table := FX^table;
270     end
    }
    #trans {
        local
        do
275         FX1^table := FX^table;
            if F^type = FX^htype then
              FX1^htype := F^type;
            else
              error(P_02, FX^htype, F^type);
280         end
        end
    }

    #type {
285     local
        do
            FX^type := FX1^type;
        end
    }
290
    FX -> #type ;
    #type {
        local
        do
295         FX^type := FX^htype;
        end
    }

    OPMUL -> opmult ;
300
    OPMUL -> opdiv ;

    end
```

On y déclare de nouveaux attributs sémantiques (lignes 11 à 18) associés aux symboles concernés, ainsi que des actions pour le contrôle (lignes 129-131, 194-198, 276-280).

Messages

Le fichier des messages d'erreurs

Listing 3.17 – Messages d'erreur pour le contrôle de type

```
// messages obligatoires
A_00, "Symbole inattendu ^1 au lieu de ^2." ,2 ;
A_01, "Fin de source attendue pres de ^1.",1;
// messages specifiques a partir d'ici
P_00, "La variable ^1 est àdej édfinie.",1;
P_01, "La variable ^1 est inconnue.",1;
P_02, "Types ^1 et ^2 incompatibles.",2;
```

On a ajouté le message concernant l'incompatibilité de types.

Génération

Le fichier makefile reste identique à celui de l'étape précédente.

Classes

Fichier de classe : INFO.java

Listing 3.18 – Classe INFO.javapour le contrôle de type

```
//-----------------------------------------------------
// INFO la classe representant une variable
//-----------------------------------------------------
package att_java ;

public class INFO {
  private int type ;
  public int getType(){
    return type;
    }

  // constructeur
  public INFO (int t){
    type = t;
    }
  // affichage
  public String toString ( ) {
    return "VAR : " + type;
    }
  }
```

3.4.3 Exécution du compilateur

```
poltamc ex0
```

Listing 3.19 – types : exécution de ex0

```
version lun avr 11 14:12:16 CEST 2005
y : VAR : 4
x : VAR : 4
```

Il n'y a pas d'erreur sur l'utilisation des types.

```
poltamc ex1
```

Listing 3.20 – types : exécution de ex1

```
version lun avr 11 14:12:16 CEST 2005
y : VAR : 1
x : VAR : 4
5 : Types 1 et 4 incompatibles.
```

Le message indique à la ligne 4 une incompatibilité de types `int / char` (la variable `y` est de type char).

Vous pouvez passer à l'étape suivante : la génération du code TAM.

3.5 Génération de code TAM

3.5.1 Introduction

Dans cette étape, vous allez décrire la génération de code

Une variable doit être conservée dans la pile de la machine TAM à une adresse donnée (différente pour chaque variable). Il faut donc encore une fois compléter la description d'une variable en ajoutant ici le déplacement de la variable par rapport à la base de la pile.

Il faut également associer du code à une déclaration, à chaque instruction et à chaque expression.

3.5.2 Les fichiers

Grammaire

Fichier POLTAM.m

Listing 3.21 – Grammaire attribuée pour la génération de code

```
------------------------------------------------------
-- Traduction d'expressions en TAM
-- avec gestion de table des symboles
--  et controle de types
------------------------------------------------------

inh  table : TDS for
     DS, IS, D, I,
     E, T, TX, F, FX;

syn  type : INTEGER for
     E, T, TX, F, FX;

inh  htype : INTEGER for
     TX, FX;
```

```
  syn  taille : INTEGER for
      TYPE;

20 inh hdep : INTEGER for
      D, DS ;

  syn dep : INTEGER for
      D ;
25
  syn  code : STRING for
      DS, D, IS, I,
      E, T, TX, F, FX;

30 inh  hcode : STRING for
      TX, FX;

  syn  cop : STRING for
      OPADD, OPMUL;
35
  space  separateur  is    "[\n\t ]+";
  sugar  opplus      is    "\+";
  sugar  opmoins     is    "\-";
  sugar  opmult      is    "\*";
40 sugar  opdiv       is    "\/";
  sugar  parouv      is    "\(";
  sugar  parfer      is    "\)";
  sugar  deuxpts     is    ":";
  sugar  pv          is    ";";
45 sugar  aff         is    ":=";
  sugar  var         is    "var";
  sugar  int         is    "int";
  sugar  char        is    "char";
  term   entier      is    "[0-9]+";
50 term   caractere   is    "\'([^\']|\\\')\'";
  term   ident       is    "[a-z_]+";

  PROG -> #table DS IS #gen;
  #table {
55   local
       t : TDS;
     do
       -- creation de la table des symboles globale
       t := new TDS();
60     DS^table := t;
       IS^table := t;
       DS^hdep := 0;
```

```
    end
  }

#gen {
    local
    do
      write "xxx.out%N" @ DS^code @ IS^code ;
    end
  }

DS -> #fin;
#fin {
    local
    do
      write DS^table;
      DS^code := "";
    end
}


DS -> #table D #dep DS #gen;
#table {
    local
    do
      D^table := DS^table;
      DS1^table := DS^table;
      D^hdep := DS^hdep;
    end
  }

#dep {
    local
    do
      DS1^hdep := D^dep;
    end
  }

#gen {
    local
    do
      DS^code := D^code @ DS1^code;
    end
}

TYPE -> int #gen ;
#gen{
    local
```

```
    do
      TYPE^taille := 4;
    end
    }

TYPE -> char #gen ;
#gen{
    local
    do
      TYPE^taille := 1;
    end
    }

D -> var ident deuxpts TYPE pv #inserer ;
#inserer {
    local
      i : INFO;
    do
      i := D^table.chercher(ident^txt);
      if i /= nil then
         error(P_00, ident^txt);
      end
      i := new INFO(TYPE^taille , D^hdep);
      call D^table.inserer(ident^txt , i);
      D^dep := D^hdep + TYPE^taille;
      D^code := "%TPUSH " @ TYPE^taille @
                "%T%I%T; reservation " @ ident^txt @ "%N";
    end
}

IS -> #gen ;
#gen {
    local
    do
      IS^code := "%THALT" @ "%T%T%T; fin du programme%N";
    end
}

IS -> #table I IS #gen;
#table {
    local
    do
      I^table := IS^table;
      IS1^table := IS^table;
    end
}
```

```
#gen {
  local
  do
    IS^code := I^code @ IS1^code;
  end
}

I -> ident aff #table E pv #typecode ;
#table {
  local
  do
    -- transmettre la table
    E^table := I^table;
  end
}

#typecode {
  local
    i : INFO;
  do
    -- rechercher l'ident
    i := I^table.chercher(ident^txt);
    if i = nil then
      error(P_01, ident^txt);
    end
    if i.getType() /= E^type then
      error(P_02, i.getType(), E^type);
    end
    I^code := E^code @"%TSTORE " @ i.genadr() @
          "%T%T; affectation " @ ident^txt @"%N";
  end
}

E -> #table T #trans TX #typecode ;
#table {
  local
  do
    T^table := E^table;
  end
}

#trans {
  local
  do
    TX^table := E^table;
    TX^hcode := T^code;
    TX^htype := T^type;
```

```
    end
}

#typecode {
    local
    do
        E^type := TX^type;
        E^code := TX^code;
    end
}

T -> #table F #trans FX #typecode ;
#table {
    local
    do
        F^table := T^table;
    end
}

#trans {
    local
    do
        FX^table := T^table;
        FX^htype := F^type;
        FX^hcode := F^code;
    end
}

#typecode {
    local
    do
        T^type := FX^type;
        T^code := FX^code;
    end
}

TX -> OPADD #table T #trans TX #typecode ;
#table {
    local
    do
        T^table := TX^table;
    end
}

#trans {
    local
        t : INTEGER;
```

```
    do
      TX1^table := TX^table;
      if TX^htype = T^type then
        TX1^htype := T^type;
        TX1^hcode := TX^hcode @ T^code @
          "%TSUBR " @ OPADD^cop @ "%N";
      else
                              error(P_02, TX^htype, T^type);
      end
    end
  }

  #typecode {
    local
    do
      TX^type := TX1^type;
      TX^code := TX1^code;
    end
  }

  TX -> #typecode ;
  #typecode {
    local
    do
      TX^type := TX^htype;
      TX^code := TX^hcode;
    end
  }


  OPADD -> opplus #gen ;
  #gen {
    local
    do
      OPADD^cop := "Iadd";
    end
  }

  OPADD -> opmoins #gen ;
  #gen {
    local
    do
      OPADD^cop := "Isub";
    end
  }

  FX -> OPMUL #table F #trans FX #typecode ;
```

```
  #table {
    local
    do
      F^table := FX^table;
    end
  }
  #trans {
    local
    do
      FX1^table := FX^table;
      if F^type = FX^htype then
        FX1^htype := F^type;
        FX1^hcode := FX^hcode @ F^code @
                "%TSUBR " @ OPMUL^cop @ "%N";
      else
                          error(P_02, FX^htype, F^type);
      end
    end
  }

  #typecode {
    local
    do
      FX^type := FX1^type;
      FX^code := FX1^code;
    end
  }

  FX -> #typecode ;
  #typecode {
    local
    do
      FX^type := FX^htype;
      FX^code := FX^hcode;
    end
  }

  F -> parouv #table E parfer #typecode ;
  #table {
    local
    do
      E^table := F^table;
    end

  }
  #typecode {
    local
```

```
345   do
        F^type := E^type;
        F^code := E^code;
      end
    }

350
    F -> ident #gen ;
    #gen {
      local
        i:INFO;
355   do
        i := F^table.chercher(ident^txt);
        if i = nil then
          error(P_01, ident^txt);
        end
360     F^type :=  i.getType();
        F^code := "%TLOAD " @ i.genadr() @
            "%T%T; referenceà  " @ ident^txt @"%N";
      end
    }

365
    F -> entier #typecode ;
    #typecode {
      local
      do
370     F^type := 4;
        F^code := "%TLOADL " @ entier^txt @
            "%T%T%T; chargement constante%N";
      end
    }

375
    F -> caractere #typecode ;
    #typecode {
      local
      do
380     F^type := 1;
        F^code := "%TLOADL " @ caractere^txt @
            "%T%T%T; chargement constante%N";
      end
    }

385
    OPMUL -> opmult #gen ;
    #gen {
      local
      do
390     OPMUL^cop := "Imul";
      end
```

```
    }

    OPMUL -> opdiv #gen ;
395 #gen {
      local
      do
        OPMUL^cop := "Idiv";
      end
400 }

    end
```

On associe donc
- un attribut synthétisé `dep` au symbole `D` (lignes 14-15),
- un attribut hérité `hdep` aux symboles `D, DS` (lignes 14-15),
- un attribut synthétisé `code` aux symboles `I, IS, E, T, F, TX, FX` (lignes 26-28),
- un attribut hérité `hcode` aux symboles `TX, FX` (lignes 30-31)
- un attribut synthétisé `cop` aux symboles `OPADD et OPMUL` (lignes 33-34).

Messages

Le fichier des messages d'erreurs est inchangé car il n'y a pas de message d'erreur associé à la génération de code.

Génération

Le fichier makefile reste identique à celui de l'étape précédente.

Classes

Fichier de classe : INFO.java

Listing 3.22 – Classe INFO.java pour la génération de code

```
    //-----------------------------------------------------
    // INFO la classe representant une variable
    //-----------------------------------------------------
    package att_java ;
 5
    public class INFO {
      // le type
      private int type ;
      public int getType(){
10      return type;
        }

      // le deplacement
      private int dep ;
```

```
15    public String genadr(){
        return "(" + type + ")␣" + dep + "[SB]";
        }

      // constructeur
20    public INFO (int t, int d){
        type = t;
        dep = d;
        }
            //
25    // affichage
      public String toString ( ) {
        return ";␣VAR␣:␣" + "type=" + type + ",␣dep=" + dep;
        }
      }
```

3.5.3 Exécution du compilateur

```
poltamc ex0
```

Listing 3.23 – code : exécution de ex0

```
   version lun avr 11 14:04:20 CEST 2005
   y : VAR : type=4dep=4
   x : VAR : type=4dep=0
   xxx.out
5          PUSH 4                    ; reservation x
           PUSH 4                    ; reservation y
           LOADL 9                   ; chargement constante
           STORE (4) 0[SB]           ; affectation x
           LOADL 3                   ; chargement constante
10         LOAD (4) 0[SB]            ; referenceà  x
           SUBR Iadd
           STORE (4) 0[SB]           ; affectation x
           LOAD (4) 0[SB]            ; referenceà  x
           LOADL 1                   ; chargement constante
15         SUBR Iadd
           STORE (4) 4[SB]           ; affectation y
           HALT                      ; fin du programme
```

Le code engendré est affiché.

Chapitre 4

Mise en œuvre d'un compilateur

Dans ce chapitre
- Syntaxe de EGG
- Les messages d'erreur.
- Contrôles effectués.
- Options de génération.
- Options du générateur de messages.

4.1 Syntaxe de EGG

4.1.1 Grammaire

Les commentaires commencent par -- jusqu'à la fin de ligne.
La déclaration des terminaux précède celle des non-terminaux.

Terminaux

Un terminal est décrit par son nom et une expression régulière (à la lex).
On peut déclarer 4 sortes de terminaux :
- space : Un séparateur, qui est consommé par l'analyseur lexical sans plus de traitement. Un commentaire en est un bon exemple.
  ```
  space comm is "//.*$";
  ```
- sugar : Du sucre syntaxique, par exemple un mot-clé.
  ```
  sugar debut is "begin";
  ```
- term : Un terminal qui a du 'sens', comme un identificateur ou un nom. Un attribut sémantique `txt` de type STRING lui est automatiquement associé avec pour valeur la chaine reconnue par l'analyseur lexical.
  ```
  term ident is "[a-z]+";
  ```
- macro : Une macro-definition est une expression régulière qui peut être utilisée pour décrire d'autres expressions régulières.
  ```
  macro lettre is "[a-z]";
  term ident is "{lettre}+";
  ```

Non-terminaux

Les non-terminaux sont déclarés après les terminaux.
On peut déclarer 2 sortes de non-terminaux :

- `A -> X ident Z;`
  `A -> ident U;`
  A est déclaré comme non-terminal décrit par 2 règles de production.
- `compil EXT;`
  EXT est déclaré comme compilateur externe. EXT doit avoir été généré par EGG comme `module` (option -m).

4.1.2 Attributs sémantiques

Les attributs sémantiques sont déclarés avant les terminaux (donc en tête de fichier).

Attributs sémantiques par défaut

Les terminaux (de la sorte `term`) ont par défaut un attribut synthétisé `txt` de type STRING.

Tous les symboles ont par défaut un attribut synthétisé `scanner` qui permet d'accéder au descripteur du fichier d'entrée.

L'axiome possède un attribut `options` de type Options qui permet de traiter les arguments de la ligne de commande.

Attributs sémantiques à déclarer

Il y a deux sortes d'attributs hérité (mot-clef `inh`) et synthétisé (mot-clef `syn`) :

- `syn code : STRING for A ;`
  code (de type STRING) est déclaré comme attribut synthétisé de A.
- `inh table : TDS for A, B;`
  table (de type TDS) est déclaré comme attribut hérité de A et B.

Il faut noter que tous les attributs sémantiques doivent avoir un nom différent (et en particulier différent des attributs sémantiques par défaut `txt, scanner, options`).

4.1.3 Actions sémantiques

Généralités

Une action sémantique est représentée par un nom et un code.

Une action sémantique peut être référencée par son nom (qui commence par un #) dans la partie droite d'une règle de production. Son code peut être donné JUSTE APRÈS la règle de production. Le code contient la manipulation

- des attributs sémantiques des symboles apparaissant dans la règle de production,
- des variables locales déclarées en début d'action,
- des variables globales déclarées avant l'action.

La syntaxe précise du code d'une action est donnée plus bas.

Remarque 1 : Si une action est référencée mais pas utilisée, une erreur fatale est signalée, mais si une action est définie sans être référencée un message sans gravité est émis, il est ainsi possible d'activer ou désactiver des actions (pour la mise au point par exemple).

Remarque 2 : Il est possible de référencer plusieurs actions successivement dans une règle de production.

```
A -> #init B #check #debug C #gen
#init { ... }
global
   g : TRUC;
#debug { ... }
#check {
   ...
   g := ...
   ...
   }
#gen { ... }
```

A la ligne 4, g est déclarée globale aux actions qui suivent (uniquement pour la règle de production en cours).

A la ligne 8, le code de l'action `#check` affecte cette variable.

Il suffit de mettre la ligne 5 en commentaire pour désactiver l'appel à `#debug` dans la règle de production.

Syntaxe du code des actions sémantiques.

Le code d'une action est donné entre { et }.

Il commence par la déclaration (éventuelle) des variables locales après le mot-clef `local`.

```
#init {
  local
    x : T ;
    ...
  }
```

Les instructions sont encadrées par les mots-clef `do` et `end`. Elles peuvent prendre plusieurs formes :

- affectation `x := ... ;`
- création d'objet

```
...
local
  x: T;
do
  ...
  new x.make(...);
end
```

Il s'agit de l'appel au constructeur de la classe de x (ici T).
- appel de procedure (à la java) `call o.p(..., ..., ...) ;`
- conditionnelle

```
if ... then
  ...
end
```

ou

```
if ... then
  ...
else
  ...
end
```

ou

```
if ... then
  ...
elseif
  ...
elseif
  ...
else
  ...
end
```

- test de type (genre instanceof)

```
match ...
 with  ...
 with  ...
 else  ...
end
```

- un appel à la procedure `write` pour afficher une expression. `write ... ;`
- un appel aux procedures `error` et `warning` pour arreter le programme en affichant un message d'erreur, ou simplement signaler sans arrêter

Les instructions manipulent des expressions qui peuvent être :
- une variable (locale ou globale) par exemple `o`
- un attribut sémantique d'un symbole `A^x` (pour l'attribut x du symbole A)
- un appel de fonction `o.f(...)` (pour une variable) ou `A^x.f(...)` (pour un attribut)
- l'application d'un opérateur `A^x + o.f(B^y.g( ...), 3)`
- la création d'un objet `new TDS(20)`
- le pointeur null `nil`
- les constantes habituelles

Action sémantique spéciale

L'action de nom `#>` permet de transmettre automatiquement les attributs hérités du symbole de gauche au symbole situé après, évitant ainsi d'avoir à écrire explicitement ces transmissions. Ainsi, si h1 et h2 sont des attributs hérités de A, h1 de CC et h2 de D alors

```
A -> B #> C #> D
```

est équivalent à

```
A -> B #x C #y D
#x {
do
  C^h1 := A^h1 ;
end}
#y {
do
  D^h2 := A^h2 ;
end}
```

Il faut noter que les transmissions explicites sont plus prioritaires que les implicites. Ainsi dans l'exemple suivant

```
A -> B #x #> C #> D
#x {
do
  C^h1 := 12 ;
end}
```

c'est l'affectation de l'action `#x` qui prime même si l'action implicite est située après.

Il existe une option de génération qui permet d'affecter automatiquement tous les attributs hérités (sauf ceux explicitement affectés dans une action sémantique). On peut ainsi se dispenser non seulement de l'écriture explicite de ces transmissions mais également des `#>`. On y perd cependant en lisibilité.

4.2 Les messages d'erreur

Pour permettre une internationalisation des messages d'erreur, il faut les décrire dans une classe Java (xxxMessages où xxx est le nom du langage). Pour faciliter la génération de cette classe il suffit d'utiliser mess.MESSC (voir le makefile de l'exemple développé) sur un fichier dont la syntaxe est très simple (voir également le fichier 'POLTAM.fr' de l'exemple développé). Il suffit de fournir un fichier similaire avec les messages dans une autre langue et de regénérer.

4.3 Contrôles effectués

Les contrôles portent essentiellement sur la vérifiaction des contraintes d'utilisation et de mise à jour des attributs sémantiques.

En cas d'erreur la génération s'arrête avec un message explicatif. Certains contrôles ne pouvant avoir lieu qu'en fin d'action ou même en fin de règle de production, le numéro de la ligne affiché peut être après la ligne réelle de l'erreur.

En plus des erreurs classiques (non déclaration d'un attribut ou d'une variable), la non initialisation d'une variable utilisée est également une erreur fatale.

4.4 Options de génération

java -cp egg.jar egg.java.EGGC <file>

- -m : pour engendrer un compilateur sans 'main' destiné à être utilisé par un autre compilateur. Par exemple EGG lui-même est décrit par 3 langages (règles de production, expressions régulières, code des actions sémantiques). Il y a donc un compilateur principal (EGG) et deux modules (EXPREG et LACTION).
- -k <number> : pour engendrer un analyseur LL(k). Attention si k est trop important (supérieur à 3) la génération peut faire exploser la pile Java.
- -l java : pour fixer le langage de génération. Pour l'instant uniquement Java.
- -s internal|jlex|<lexical analyzer class> : il est possible de ne pas utiliser JavaLex (le défaut) pour l'analyse lexicale. On peut utiliser un analyseur spécifique (dans le langage de génération) si les expressions régulières ne suffisent pas à décrire les terminaux.
- -a <config file> : le nom du ficher qui contient les bibliothèques utilisées pour compléter le compilateur (en général les classes des attributs sémantiques).
- -p <generation package> : le nom du paquetage java engendré.
- -mkdir : pour créer automatiquement le répertoire du paquetage (ne pas utiliser cette option si le répertoire existe déjà comme dans le cas des compilateurs à modules).
- -i : Pour rendre plus clair le code des actions sémantiques il peut être bon d'interdire l'imbrication des structures de contrôle. Utile pour l'enseignement des grammaires attribuées par exemple.
- -b : pour engendrer une version sérialisée du compilateur.
- -vs : pour donner un numero de version qui est affiché automatiquement à l'exécution du compilateur engendré.
- -auto : pour la transmission automatique des attributs hérités.
- -nsargs : pour choisir comment les arguments de la ligne de commande sont décodés. Par défaut le premier argument est le nom du fichier à traiter. Il suffit de définir une classe xxxOptions (où xxx est le nom du compilateur) implementant l'interface libjava.Options (voir l'API de libjava).
- -? : fournit la liste des options.

4.5 Options du générateur de classes messages

java -cp mess.jar :egg.jar mess.java.MESSC <file>

- -l java : pour fixer le langage de génération. Pour l'instant uniquement Java.

- -d <répertoire> : pour fixer le répertoire dans lequel engendrer la classe. qui implémente libjava.Messages (voir l'API de libjava).
- -? : fournit la liste des options.

Chapitre 5

Annexes

5.1 Structure du compilateur généré

Différents fichiers sont créés dans le répertoire de génération :

- le fichier contenant la classe racine.
  La classe racine contient le nécessaire à la prise en compte du fichier à compiler avec le nouveau compilateur et elle lance l'analyse de l'axiome de la grammaire.
- un fichier par classe symbole
  Chaque classe associée à un terminal non sucre (`term`) possède une méthode permettant de reconnaître l'expression régulière associée au terminal. Son nom est de la forme T_SYMB_LANG.java.
  Chaque classe associée à un non-terminal possède une méthode d' analyse correspondant aux différentes règles de production décrivant le non-terminal. Son nom est de la forme S_SYMB_LANG.java.

5.2 Glossaire

Générateur de compilateur

Un programme qui, à partir d'une grammaire d'un langage, fabrique automatiquement un compilateur (plus ou moins complet) pour ce langage.

Symbole Terminal

Représente un mot du vocabulaire de base du langage.

Symbole Non-terminal

Représente une sous-phrase du langage.

Règle de production

Décrit un non-terminal

Grammaire LL(1)

Une grammaire est LL(1) si elle n'est pas récursive à gauche et si les symboles directeurs des règles de production décrivant un même symbole non-terminal sont des ensembles disjoints deux à deux. Une grammaire est LL(k) si elle n'est pas récursive à gauche et si les k-symbole directeurs des règles de production décrivant un même symbole non-terminal sont des ensembles disjoints deux à deux. L'analyse et la traduction d'un texte source en un texte cible font appel à trois types d'analyse :

Analyse lexicale

C'est la partie du compilateur qui reconnaît les terminaux du langage à partir des caractères du texte source.

Analyse syntaxique

C'est la partie du compilateur qui vérifie que les phrases du texte source sont en accord avec la grammaire du langage. Elle utilise les terminaux reconnus par l'analyse lexicale et les règles de production de la grammaire. L'analyse peut être ascendante (LR) ou descendante (LL).

Analyse sémantique

C'est la partie du compilateur qui implante les différents contrôles et la génération du code cible. Parmi les contrôles : insertion des identificateurs et des informations associées dans la table des symboles, contrôle (ou inférence) de type si le langage à traduire est type, génération du code cible pour une machine réelle ou virtuelle. Action sémantique Dans une grammaire attribuée, la traduction du langage est effectuée par des actions sémantiques insérées entre les symboles de la partie droite des règles de production. Ces actions sont soit du code (C, Eiffel, Java, etc.) soit un langage spécifique de manipulation d'attributs sémantiques. Dans une grammaire attribuée les informations nécessaires a la traduction sont associées directement aux différents symboles de la grammaire. Ces informations sont appelées attributs sémantiques. Suivant l'utilisation des attributs, on les qualifie de synthétisés ou d'hérités.

Attribut sémantique hérité

Un attribut sémantique est hérité si sa valeur doit être transmise à un fils ou à un frère dans l'arbre syntaxique.

Attribut sémantique synthétisé

Un attribut sémantique est synthétisé si sa valeur doit être transmise au père dans l'arbre syntaxique.

Grammaire attribuée

Une grammaire attribuée est une extension des grammaires dans laquelle on associe : des informations aux différents symboles (attributs sémantiques) ; du code de mise à jour des attributs sémantiques aux règles de production. On distingue différentes grammaires attribuées suivant les contraintes que l'on impose aux attributs ou aux actions sémantiques.

Grammaire L-attribuée

Si on impose que la mise a jour des attributs puisse se faire par un parcours en profondeur/droite-gauche, la grammaire est dite L-attribuée.

Grammaire S-attribuée

Si une grammaire attribuée ne possède que des attributs sémantiques synthétisés, elle est dite S-attribuée. Ce genre de grammaire attribuée est facile à analyser par un analyseur ascendant : la synthèse des attributs se fait naturellement 'en montant'.

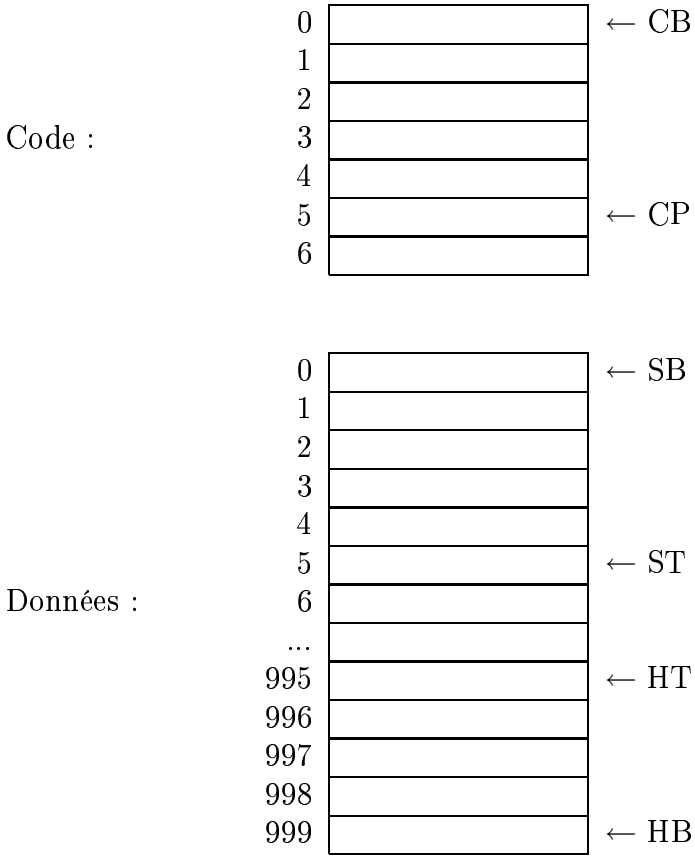
Compilateur modulaire

Si, dans la partie droite d'une règle de production, on rencontre un compilateur externe, le programme passe la main à un autre compilateur généré par EGG et compilé avec l'option - m. Notons que ces deux compilateurs n'ont pas le même analyseur lexical. Par ailleurs, aucun contrôle n'est effectué pour assurer la similitude des attributs du compilateur externe.

5.3 La machine TAM

5.3.1 Structure

Machine à pile. Pas de registre de donnée.



5.3.2 Instructions

Instructions (16) dont :

```
    LABEL etiq          Déclaration d'une étiquette
    PUSH n              ST = ST+n
    POP (d) n           a = ST -d;  ST = ST - d -n;
                        Pour i de d à 0 Donnees(ST++) = Donnees[a++] fin pour
    LOADL n             Donnees(ST)= n; ST = ST+1
    LOAD (n) d[r]       Pour i de 0 a n-1
                          Donnees(ST+i) = Donnees(val(r)+d+i)
                        fin pour;
                        ST = ST+n
    STORE (n) d[r]      Pour i de 0 a n-1
                          Donnees(val(r)+d+i) = Donnees(ST+i-n);
                        fin pour;
```

```
                    ST = ST-n
JUMP etiq           CP = val(etiq)
JUMP d[r]           CP = val(r) + d
JUMPIF (n) etiq     si Donnees(ST -1) = n alors CP = val(etiq) fin si;
                    ST = ST -1
JUMPIF (n) d[r]     si Donnees(ST -1) = n alors CP = val(r) + d fin si;
                    ST = ST -1
SUBR op             Appel de op, consommation des arguments
                    laissés en sommet de pile
HALT                Arret
```

5.3.3 Fonctions de bibliothèque

Fonctions sur les Booléens

Nom	Paramètres	Résultat	
BNeg	1	1	Négation logique
BOr	2	1	Ou logique
BAnd	2	1	Et logique
BOut	1	0	Affiche sur `stdout` un booléen (`true` ou `false`)
BIn	0	1	Lit sur `stdin` un booléen (`true` ou `false`)
B2C	0	1	Conversion vers un caractère (`true` = `'1'`, `false` = `'0'`)
B2I	0	1	Conversion vers un entier (`true` = `1`, `false` = `0`)
B2S	0	1	Conversion vers une chaîne (`"true"`, `"false"`)

Fonctions sur les Caractères

Nom	Paramètres	Résultat	
COut	1	0	Affiche sur `stdout` un caractère
CIn	0	1	Lit sur `stdin` un caractère
C2B	1	1	Conversion vers un booléen (`'1'` = `true`, `'0'` = `false`)
C2I	1	1	Conversion vers un entier (le code ASCII)
C2S	1	1	Conversion vers la chaîne contenant seulement ce caractère

Fonctions sur les Entiers

Nom	Paramètres	Résultat	
INeg	1	1	Négation entière
IAdd	2	1	Addition entière
ISub	2	1	Soustraction entière
IMul	2	1	Multiplication entière
IDiv	2	1	Diviseur dans division entière
IMod	2	1	Reste dans division entière
IEq	2	1	Test égalité entre 2 entiers
INeq	2	1	Test différence entre 2 entiers
ILss	2	1	Test inférieur strictement entre 2 entiers
ILeq	2	1	Test inférieur ou égal entre 2 entiers
IGtr	2	1	Test supérieur strictement entre 2 entiers
IGeq	2	1	Test supérieur ou égal entre 2 entiers
IOut	1	0	Affiche sur `stdout` un entier
IIn	0	1	Lit sur `stdin` un entier
I2B	1	1	Conversion vers un booléen (1 = `true`, 0 = `false`)
I2C	1	1	Conversion vers un caractère (le code ASCII)
I2S	1	1	Conversion vers la chaîne représentant cet entier

Fonctions de gestion de la Mémoire

Nom	Paramètres	Résultat	
MVoid	0	1	Renvoie la valeur « adresse non initialisée »
MAlloc	1	1	Alloue un bloc mémoire et renvoie son adresse
MFree	1	0	Libère un bloc mémoire
MCompare	2	1	Test égalité entre le contenu de 2 blocs mémoire
MCopy	2	0	Copie le contenu d'un bloc mémoire dans le second bloc mémoire

Fonctions sur les Chaînes

Nom	Paramètres	Résultat	
SAlloc	1	1	Création d'une nouvelle chaîne
SCopy	1	1	Création d'une copie de la chaîne passée en paramètre
SConcat	2	1	Création d'une nouvelle chaîne contenant la juxtaposition de deux paramètres
SOut	1	0	Affiche sur `stdout` une chaîne
SIn	0	1	Lit sur `stdin` une chaîne
S2B	1	1	Conversion vers un booléen (`"true"` = `true`, `"false"` = `false`)
S2C	1	1	Extraction du premier caractère de la chaîne
S2I	1	1	Conversion vers l'entier représenté par la chaîne