Université de Khenchela
ABBES LAGHROUR

3^{emme} licence Informatique – Système informatique

TP COMPILATION

REALISER UN COMPILATEUR ARAPASC

Etudiants :

Ahmed Fouad LAGHA

Mohamed Tarek LALAOUNA

Enseignant-responsable du mini-projet :

Dr Massoudi N.

2022/2023

Table des matières

Travail à Réaliser

Danc ce projet on va construire un compilateur appelé ARAPASC où l'utilisateur entre un programme en langue arabe avec l'alphabet latin et le compilateur traduira le programme arabe en langage Pascal et en langage C en créant deux fichiers exécutables ".pas" et ".c ".

Si un programme est correct (après les vérifications lexicale, syntaxique,..) deux fichiers de sortie sera génère contenants le code « arabe » en langage pascal et C.

Nous utilisons pour ce travail pratique l'outil JavaCC et la plate-forme Eclipse.

JavaCC (java Compiler Compiler) est un générateur de compilation utilisant la méthode LL(1) qui, a partir de la spécification d'une grammaire et d'un code adéquat. Génère (si pas d'erreurs), un fichier source écrit en Java .Ce fichier source devra a son tour être compile en utilisant éclipse (et plug-in adéquat) pour produire l'exécutable du compilateur que vous aurez conçu.

Notations et Acronymes

Javacc : java compiler compiler

DDS : La définition dirigée par la syntaxe

Intoduction

Pourquoi écrire un compilateur ?

Il est à peu près certain que la majorité des informaticiens n'auront jamais à écrire de compilateur.Cependant, mener à bien l'écriture d'un " gros " programme, le tester, documenter... a déjà un intérêt en soi. De plus, les algorithmes employés dans un compilateur sont repris dans la résolution d'autres problèmes : manipulation de listes de recherche (gestion de la table des symboles), vérification de la correction d'en-registrements (analyse syntaxique), traitement de caractères, de mots (grammaire BNF). Enfin, il semble intéressant pour un programmeur de connaître les dessous des choses; comprendre comment les programmes se compilent et s'exécutent aide à la compréhension des limitations, obligations... imposées par les langages.

Cette "transformation" du texte d'un programme vers une forme exécutable nécessite plusieurs étapes. Le regroupement de celles-ci constitue le processus dit de compilation. Dans ce rapport chacune de chapteries qu'on décrira amènera à la réalisation d'un compilateur. Afin de mener à bien ce développement, notre groupe a dû exploiter plusieurs modules, parallèlement les uns aux autres. Le résultat obtenu est, donc, la synergie de plusieurs modules dont il est question dans la suite de ce rapport.

Un compilateur est un programme qui permet de traduire un programme source écrit dans un certain langage en un programme cible dans un autre langage. Le plus souvent, le langage cible est le langage machine. Un compilateur fonctionne par analyse-synthèse, c'est-à-dire qu'au lieu de remplacer chaque construction du langage source par une suite équivalente de constructions du langage cible, il commence par analyser le texte source pour en construire une représentation intermédiaire qu'il traduit à son tour en langage cible.

JavaCC est un programme permettant de créer automatiquement un analyseur syntaxique (=parseur, parser en anglais) pour un langage décrit sous forme de grammaire hors contexte.

I. Analyse lexicale et syntaxique

1. JavaCC

Java Compiler Compiler (JavaCC) est le plus utilisé des générateurs de parer pour Java. Un générateur de parseur est un outil qui lit les spécifications d'une grammaire et qui la convertit en program Java. En plus d'être un générateur de parseur, Java CC fournit d'autre possibilité relative `à la génération de parer comme la construction d'arbre, le debugage, etc... JavaCC prend comme entrée un fichier MaGrammaire.jj qui contient entre autres les descriptions des règles de la grammaire et produit un parser descendant (dans le fichier MaGrammaire.java). Une classe MaGrammaire est définie dans le fichier java. Elle implémente l'interface MaGrammaireConstants, définie dans MaGrammaireConstants.java et qui contient les définitions des mots clés de la grammaire.

D'une façon plus générale l'analyse consistera à prendre un programme écrit en ARAPASC (´écrit dans un fichier .jj) et le découper en une séquence d'unités lexicales. L'analyse syntaxique va traiter ces unités lexicales afin de donner un arbre de syntaxe abstraite (ASA) qui représente la structure du programme. (fichier .java) Cet arbre est la base du compilateur, puisque toutes les autres tâches comme le contrôle de type, le compilateur et l'interpréteur dépendent directement de cet arbre.

2. Analyse lexicale

Le but de l'analyse lexicale est de transformer une suite de symboles en terminaux (un terminal peut être par exemple un nombre, un signe '+', un identifiant...). Une fois cette transformation effectuée, la main est repassée à l'analyseur syntaxique. Le but de l'analyseur lexical est de découper un fichier source en symboles terminaux et de les fournir à l'analyseur syntaxique. L'analyseur lexical peut séparer logiquement une séquence de symboles dans de sous-séquences appelés tokens en le classifiant par un ordre spécifique. Un possible exemple avec du langage ARAPASC pourra être :

```
Barnamedj EXEMPLE;
moutaghaire :
a : tabiaai;
bidaya
    a ← 3;
nihaya.
```

L'analyseur pourra donc aboutir a cette sequence :

```
< BARNAMEDJ > < IDENTIFIER > < POINTVERGULE > < MOUTAGHAIRE > < DEUXPOINT >
< IDENTIFIER > < DEUXPOINT > < INT > < POINTVERGULE > < BIDAYA > < IDENTIFIER >
< ASSIGN > < INTEGER_LITERAL > < POINTVERGULE > < NIHAYA > < POINT >
```

3. Créer un ficher *.jj*

La syntaxe du fichier passé en paramètre à javacc pour l'analyse lexicale a une "forme" type de ce genre :

– javac_options

– PARSER_BEGIN(MaGrammaire)

– java_grammaire_declaration

– PARSER_END(MaGrammaire)

– (production)*

Dans notre cas sp´ecifique le fichier contiendra :

```
options
{
  static = true;
}

PARSER_BEGIN(ARAC)
package parser;

import java.util.*;
import java.io.*;

public class ARAC
{
  static ArrayList<String> Pascal = new ArrayList<String>();
  static ArrayList<String> C = new ArrayList<String>();
  public static void main(String args []) throws IOException,ParseException
  {
    //Saisir le programme ARAPASC
       FileInputStream file = new FileInputStream("src/test/if_else.txt");
       ARAC parser = new ARAC(file);
       parser.Start();
       //programme pascal
       File file_PASCAL = new File("src/resultat/PASCAL_programme.pas");

    try {
      FileWriter Writer_PASCAL = new FileWriter(file_PASCAL);
        for(String P:Pascal)
             Writer_PASCAL.write(P);

      Writer_PASCAL.close();
      System.out.println("Le programme a été converti en Pascal.");
    } catch (IOException e) {
      System.out.println("Une erreur s'est produite.");
      e.printStackTrace();
    }

    //programme c
    File file_C = new File("src/resultat/C_programme.c");

    try {
      FileWriter Writer_C = new FileWriter(file_C);
      for(String c:C)
             Writer_C.write(c);

      Writer_C.close();
      System.out.println("Le programme a été converti en C.");
    } catch (IOException e) {
      System.out.println("Une erreur s'est produite.");
      e.printStackTrace();
    }
       file.close();
  }
}

   PARSER_END(ARAC)
```

Le fichier .jj doit contenir la définition des tokens pour ainsi pouvoir les identifier dans le code source que l'on passera au parser. En spécifiant tous les termnes que l'on peut rencontrer dans un langage précis on parviendra au but de l'identification dans le langage. En premier lieu il est nécessaire de définir tous les caractères « invisibles » du langage ceux qui vont être lus par le parser mais qui n'ont pas d'influence sur la structure du programme et qui doivent donc être passés. On sp´ecifiera ainsi notre analyseur lexical en utilisant les mots clés skip et tokens :

```
SKIP : {
  " "
| "\t"
| "\n"
| "\r"
| < "//" (~[ "\n", "\r" ])* ( "\n" | "\r" | "\r\n" ) >
| < "/*" (~[ "*" ])* "*" ( ~[ "/" ] (~[ "*" ])* "*" )* "/" >
}

TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL : < DECIMAL_LITERAL >   >
| < FLOAT_LITERAL : (< INTEGER_LITERAL >)+( "." (< INTEGER_LITERAL >)+)? >
| < #DECIMAL_LITERAL : ["1"-"9"] (["0"-"9"])* | "0" >
}

TOKEN : /* KEYWORDS */
{
    < BARNAMEDJ : "Barnamedj">
  | < MOUTAGHAIRE : "moutaghaire" >
  | < BIDAYA : "bidaya" >
  | < NIHAYA : "nihaya" >
  | < INT : "tabiaai">
  | < BOOLEAN :"mantiki" >
  | < FLOAT :"sahih" >
  | < CHAR:"harref" >
  | < IF:"ida">
  | < ELSE:"waila" >
  | < BREAK:"kherouj" >
  | < CONTINUE:"akmil" >
  | < ENDIF:"nihaya_ida" >
  | < DO:"ifaal" >
  | < WHILE:"madama" >
  | < ENDWHILE:"nihaya_madama" >
  | < FOR:"halaka" >
  | < ENDFOR:"nihaya_halaka" >
  | < PROCEDURE:"dala" >
  | < RETURN:"irjaa" >
  | < VOID : "lachaye" >
  | < OUKTOUB : "ouktoub" >
  | < IKRAA : "ikraa" >
}

TOKEN : /* OPERATOR */
{
      < ADD:"+">
  | < MIN:"-">
  | < MUL:"*">
  | < DIV:"/">
  | < MODULO:"%" >
  | < ASSIGN:"<-">
  | < ADDONE:"++" >
  | < MINUTEONE:"--" >
}
```

```
TOKEN : /* RELATION */
{
       < LT:"<"  >
  | < LE:"<=" >
  | < GT:">"  >
  | < GE:">=" >
  | < EQ:"tousaoui" >
  | < NE:"latousaoui" >
   /* LOGICAL */
  | < AND:"wa" >
  | < OR:"aw" >
  | < NOT:"la" >
}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER : < LETTER > ( < LETTER > | < DIGIT > )* >
  | < #LETTER : [ "_", "a"-"z", "A"-"Z" ] >
  | < #DIGIT : [ "0"-"9" ] >
}

TOKEN:/* SEPARATER */
{
    <LC:"(">
  | <RC:")">
  | <LM:"[">
  | <RM:"]">
  | <LB:"{">
  | <RB:"}">
  | < COMMA:"," >
  | < POINTVERGULE:";" >
  | < DEUXPOINT:":" >
  | < POINT: "." >
  | < GUILLEMET : "\"" >
}

TOKEN : /* MESSAGES */
{
  < MESSAGE :  < GUILLEMET > < STRING > ( < STRING > )* < GUILLEMET > >
| < #STRING : ~[ ] >

}
```

4. Analyse syntaxique

Si la définition des tokens permet d'identifier les symboles d'un langage, la spécification du parser permet de définir la structure du langage et donc de vérifier la validité d'un programme par rapport à la grammaire requise. Il faut donc définir une séquence de token qui sera déterminée comme valide par le parser pour ensuite passer à la phase d'analyse syntaxique. La phase successive à l'analyse lexicale est la vérification du bon enchainement des tokens (terminaux) reconnus par JavaCC. Cette tâche est accomplie en gérant l'analyseur syntaxique qui reconnait une grammaire du type BNF. Il est aussi possible, en plus de la vérification de la syntaxe de la grammaire, de lui faire exécuter des actions lorsqu'une régle est reconnue.

La spécification du parseur consiste à ce qu'on appelle une production BMF et apparaitra comme une définition de méthode java :

```
//analyseur syntaxique

void Start() :
{ }
{
  program() <EOF>
}
```

```
void program() : {Token t;}
{
  < BARNAMEDJ > t = < IDENTIFIER > < POINTVERGULE >
  { Pascal.add("program  "+t.toString()+";\n"); } { C.add("#include
<iostream>\nusing namespace std;\nint main(){\n "); }  declaration()
  < BIDAYA > { Pascal.add("begin\n"); }
  (
    Statement()
  )*
  < NIHAYA > < POINT > { Pascal.add("\nend."); } { C.add("return 0; \n }"); }
}

void declaration() : {Token t1,t2; ArrayList<String> C_ident = new
ArrayList<String>();}
{
  < MOUTAGHAIRE > < DEUXPOINT > { Pascal.add("Var :\n"); }
  (
    t1 = < IDENTIFIER > { Pascal.add(t1.toString()); C_ident.add(t1.toString()); }
    (
      < COMMA > t2 = < IDENTIFIER > { Pascal.add(","+t2.toString());
C_ident.add(","+t2.toString());}
    )*
    < DEUXPOINT > { Pascal.add(" : "); } type ()
    < POINTVERGULE > { Pascal.add(";\n"); C.addAll(C_ident); C.add(";\n"); }
  )*
}

void type() :
{}
{
    < INT > { Pascal.add("integer"); C.add("int "); }
  | < BOOLEAN > { Pascal.add("boolean"); C.add("bool "); }
  | < FLOAT > { Pascal.add("real"); C.add("float "); }
  | < CHAR > { Pascal.add("char"); C.add("char "); }
}

void Statement() :
{}
{
  SequenceStatement() | ConditionalStatements() | LoopStatement() |
ReadStatement() | WriteStatement()
}

void ReadStatement() : { Token t15,t16; }
{
  < IKRAA > <LC>  t15 = <IDENTIFIER> { Pascal.add("Read ("+t15.toString());
C.add("cin >> "+t15.toString()); }
  (
    < COMMA > t16 =<IDENTIFIER> { Pascal.add(","+t16.toString()); C.add(" >>
"+t16.toString()); }
  )*
   <RC> < POINTVERGULE >  { Pascal.add(");"); C.add(" >> endl;\n"); }
  (
    Statement()
  )*
}

void WriteStatement() : { Token t18,t19;}
```

```
{
  < OUKTOUB > <LC>
  (
    t18 = <  MESSAGE > { Pascal.add("Write ("+t18.toString()); C.add("cout <<
"+t18.toString()); }
  | t19 = <IDENTIFIER> { Pascal.add("Write ("+t19.toString()); C.add("cout <<
"+t19.toString()); }

  )

  (
    < COMMA >
    (
      t18 = <  MESSAGE > { Pascal.add(","+t18.toString()); C.add(" <<
"+t18.toString()); }
      | t19 = <IDENTIFIER> { Pascal.add(","+t19.toString()); C.add(" <<
"+t19.toString()); }

      )

  )*

  <RC> < POINTVERGULE > { Pascal.add(");\n"); C.add(" << endl ;\n"); }
  (
    Statement()
  )*

}

void SequenceStatement() : {Token t3;}
{
   t3 = <IDENTIFIER> { Pascal.add(t3.toString()); C.add(t3.toString());}
   (
     statement() | assignment()
   )
   < POINTVERGULE > { Pascal.add(";\n"); C.add(";\n"); }
}

void statement() : { Token t4,t5,t6,t7,t8;}
{
   (
     <ASSIGN> { Pascal.add(" := "); C.add(" = "); }
     (
               t4  = <INTEGER_LITERAL> { Pascal.add(t4.toString());
C.add(t4.toString());}
             | t5 = <FLOAT_LITERAL> { Pascal.add(t5.toString());
C.add(t5.toString()); }
       )

   )?
   (
     <COMMA> t6 = <IDENTIFIER> { Pascal.add("," + t6.toString()); C.add("," +
t6.toString()); }
     (
       <ASSIGN> { Pascal.add(":="); Pascal.add("="); }
       (
          t7 =  <INTEGER_LITERAL> { Pascal.add(t7.toString());
C.add(t7.toString()); }
```

```
            | t8 = <FLOAT_LITERAL> { Pascal.add(t8.toString());
C.add(t8.toString()); }

          )
        )?
    )*
}
void assignment() :
{ }
{
   <ASSIGN> { Pascal.add(":="); Pascal.add(" = "); } Expression()
}

void Expression() :
{ }
{
  AdditiveExpression()
}

void AdditiveExpression() :
{ }
{
  MultiplicativeExpression()
  (
    (
      <ADD> { Pascal.add("+"); C.add("+"); } | <MIN> { Pascal.add("-"); C.add("-
"); }
    )
    MultiplicativeExpression()
  )*
}

void MultiplicativeExpression() : { }
{
  UnaryExpression()
  (
    (
      <MUL> { Pascal.add("*"); C.add("*"); } | <DIV> { Pascal.add("/");C.add("/");
} | <MODULO> { Pascal.add("%"); C.add("%"); }
    )
    UnaryExpression()
  )*
}
void UnaryExpression() : { Token t9,t10,t11; }
{
      t9 = <INTEGER_LITERAL> { Pascal.add(t9.toString()); C.add(t9.toString()); }
|     t10 = <FLOAT_LITERAL> { Pascal.add(t10.toString()); C.add(t10.toString()); }
|     t11 = <IDENTIFIER> { Pascal.add(t11.toString()); C.add(t11.toString()); }
}
void ConditionalStatements() : { }
{
  <IF> <LC> { Pascal.add("if ("); C.add("if ("); } Logiccondition() <RC> < DO > {
Pascal.add(") do \n "); C.add(") { \n "); }
  Statement()
  (
    <ELSE> { Pascal.add("else\n"); C.add("\n} else{ \n"); } Statement()
  )?
  < ENDIF > { Pascal.add("end;\n"); C.add(" }\n"); }
}
```

```
void Logiccondition() : { }
{
  Condition() ( Logic() Logiccondition() )?
}
void Logic() : { }
{
  <AND> { Pascal.add(" and "); C.add(" && "); } | <OR> { Pascal.add(" or
");C.add(" || "); }
}
void Condition() : { }
{
  Expression() ( Relationship() Expression() )?
}
void Relationship() : { }
{
  (
    <GT> { Pascal.add(" > "); C.add(" > "); }
  | <LT> { Pascal.add(" < "); C.add(" < "); }
  | <GE> { Pascal.add(" >= "); C.add(" >= "); }
  | <LE> { Pascal.add(" <= "); C.add(" <= ");}
  | <NE> { Pascal.add(" <> "); C.add(" != "); }
  | <EQ> { Pascal.add(" = "); C.add(" == ");}

  )
}
void LoopStatement() : { }
{
    (
      <WHILE> <LC> { Pascal.add("while ( "); C.add(" while ( "); }
Logiccondition() <RC> { Pascal.add(" ) do\nbegin"); C.add(" ) { \n"); } < DO >
      Statement()
      <ENDWHILE > { Pascal.add("end;\n"); C.add(" } ");}
    )
|
      (
        <DO> { Pascal.add("repeat\n"); C.add(" do { \n "); }
        Statement()
        <WHILE> <LC> { Pascal.add("\nuntil ( "); C.add(" } while ( "); }
Logiccondition() <RC> <POINTVERGULE> { Pascal.add(" );\n"); C.add(" );\n"); }
      )
|
      (
        <FOR> <LC> { Pascal.add("for ( "); C.add("for ( "); } statement()
<POINTVERGULE> { Pascal.add(";"); C.add(";"); }
        Logiccondition() <POINTVERGULE> { Pascal.add(";"); C.add(";"); }
        crease() <RC> <DO> { Pascal.add(")\n"); C.add(") { \n"); }
        Statement()
        < ENDFOR > { Pascal.add("end;\n"); C.add(" }\n"); }
      )
}
void crease() : { Token t12;}
{
  t12 = <IDENTIFIER> { Pascal.add(t12.toString()); C.add(t12.toString()); }
  (
    <ADDONE> { Pascal.add("++"); C.add("++"); }
  | <MINUTEONE> { Pascal.add("--"); C.add("--"); }

  )?
}
```

5. Grammaire sous forme BNF :

Start ::= program <EOF>

program ::= <BARNAMEDJ> <IDENTIFIER> <POINTVERGULE> declaration <BIDAYA> (Statement)* < NIHAYA > < POINT >

declaration ::= < MOUTAGHAIRE > < DEUXPOINT > (< IDENTIFIER > (< COMMA > < IDENTIFIER >)* < DEUXPOINT > < POINTVERGULE >)*

type ::= (<INT> | <BOOLEAN> | <FLOAT> | <CHAR>)

Statement ::= (SequenceStatement | Conditionalstatements | loopstatement | ReadStatement | WriteStatement)

ReadStatement ::= < IKRAA > <LC> <IDENTIFIER> (< COMMA > <IDENTIFIER>)* <RC> < POINTVERGULE > (Statement)*

WriteStatement ::= < OUKTOUB > <LC> (< MESSAGE >|<IDENTIFIER>) (< COMMA > (< MESSAGE >|<IDENTIFIER>))* <RC> < POINTVERGULE > (Statement)*

SequenceStatement ::= (statement | assignment) < POINTVERGULE >

statement ::= (<ASSIGN> (<INTEGER_LITERAL> | <FLOAT_LITERAL>)) (<COMMA> <IDENTIFIER> (<ASSIGN> (<INTEGER_LITERAL> | <FLOAT_LITERAL>))?)*

assignment ::= <ASSIGN> Expression

Expression ::= AdditiveExpression

AdditiveExpression ::= MultiplicativeExpression ((<ADD> | <MIN>) MultiplicativeExpression)*

MultiplicativeExpression ::= UnaryExpression ((<MUL> | <DIV> | <QUEUE>) UnaryExpression)*

UnaryExpression ::= <INTEGER_LITERAL> | <FLOAT_LITERAL> | <IDENTIFIER>

Conditionalstatements ::= <IF> <LC> Logiccondition <RC> <DO> Statement (<ELSE> Statement)? < ENDIF >

Logiccondition ::= Condition (Logic Logiccondition)?

Logic ::= <AND> | <OR>

Condition ::= Expression (Relationship Expression)?

Relationship ::= (<GT> | <LT> | <GE> | <LE> | <NE> | <EQ>)

loopstatement ::= (<WHILE> <LC> Logiccondition <RC> Statement <ENDWHILE >)

| (<DO> Statement <WHILE> <LC> Logiccondition <RC> <POINTVERGULE>)

| (<FOR> <LC> statement <POINTVERGULE> Logiccondition <POINTVERGULE>crease <RC> Statement) < ENDFOR >

crease ::= <IDENTIFIER> (<ADDONE> | <MINUTEONE>)?

6. Génération du parseur

Ayant con¸cu le fichier parse.jj, on appellera JavaCC sur ce document.

D:

\home\\parser>javacc parser.jj

Java Compiler Compiler Version 2.1 (Parser Generator)

Copyright (c) 1996-2001 Sun Microsystems, Inc.

Copyright (c) 1997-2001 WebGain, Inc.

(type "javacc" with no arguments for help)

Reading from file parser.jj . . .

File "TokenMgrError.java" does not exist. Will create one.

File "ParseException.java" does not exist. Will create one.

File "Token.java" does not exist. Will create one.

File "SimpleCharStream.java" does not exist. Will create one.

Parser generated successfully.

Cette opération va générer 7 classes java, chacune avec son propre fichier

– TokenMgrError - c'est une simple classe d'erreur. Elle est utilisée pour les erreurs d´etect´es par l'analyseur lexicale et c'est aussi une sous-classe de Throwable.

– ParseException - c'est une autre classe d'erreur. Sous-classe de Exception et Throwable.

– Token - c'est une classe qui représente les tokens.

– SimpleCharStream

– ParserConstants - nombre de classes utilisés par l'analyseur lexical et parseur.

– ParserTokenManager - c'est l'analyseur lexical

– Parser - est le parseur en soi même.

7. Problèmes : élimination de la récursivité gauche

La grammaire proposée pour le langage ARAPASC est, dans certains cas, ambigüe parce qu'elle produit deux arbres d'analyse différentes pour la même entrée. Donc on pourrait avoir deux dérivations les plus à gauche différentes pour la même entrée.

La plupart des générateurs d'analyseurs syntaxiques (voir Flex/Bison) permettent la spécification de règles de "suppression de l'ambiguïté" qui suppriment les dérivations non désirées afin de s'assurer que la chaîne d'entrée a une seule interprétation possible.

Toutefois, dans notre cas, en utilisant JavaCC , on n'a pas d'autres choix que de r´e´ecrire la grammaire en une grammaire ´equivalente acceptable pour une analyse LL(1) (ou, pour plus tard, LR(1))

Pour résoudre ce problème, nous avons appliqué simplement la règle d'élimination de la récursivité du cours : Production « A --> A alpha | beta » peut être remplacée par : A -> {beta}A' A -> {alpha}A' | {epsilon}

8. La définition dirigée par la syntaxe (DDS)

Une DDS d'un langage de programmation est constitué par :

- la grammaire qui spécifie la syntaxe du texte d'entrée.

- les règles sémantiques qui calculent les valeurs des attributs associés aux symboles d'une construction du langage.

Production	Action sémantique
START→PROGRAM	Pascal_file = PROGRAM.pas C_file = PROGRAM.c
PROGRAM→Barnamedj IDENTIFIER ; DECLARATION bidaya STATEMENT nihaya ;	PROGRAM.pas = "*program*" + IDENTIFIER.pas + ";" + DECLARATION.pas + "*begin*" + STATEMENT.pas + "*end.*" PROGRAM.c = "*#include* < *iostream* >\n*using namespace std;*\n*int main*(){"+DECLARATION.c +STATEMENT.c+"*return* 0; }"
IDENTIFIER → CHIFFRE IDENTIFIER	IDENTIFIER.pas = CHIFFRE.pas + IDENTIFIER.pas IDENTIFIER.c = CHIFFRE.c + IDENTIFIER.c
IDENTIFIER → LETTRE IDENTIFIER	IDENTIFIER.pas = LETTRE.pas + IDENTIFIER.pas IDENTIFIER.c = LETTRE.c + IDENTIFIER.c
IDENTIFIER → $\boldsymbol{\varepsilon}$	IDENTIFIER.pas = "" IDENTIFIER.c = ""
CHIFFRE → 1	CHIFFRE.pas = "1" CHIFFRE.c = "1"
CHIFFRE → 2	CHIFFRE.pas = "2" CHIFFRE.c = "2"
…	…
CHIFFRE → 9	CHIFFRE.pas = "9" CHIFFRE.c = "9"
LETTRE → a	LETTRE.pas = "*a*" LETTRE.c = "*a*"
LETTRE → b	LETTRE.pas = "*b*" LETTRE.c = "*b*"
…	…
LETTRE → z	LETTRE.pas = "*z*" LETTRE.c = "*z*"
LETTRE → A	LETTRE.pas = "*A*" LETTRE.c = "*A*"
LETTRE → B	LETTRE.pas = "*B*" LETTRE.c = "*B*"
…	…
LETTRE → Z	LETTRE.pas = "*Z*"

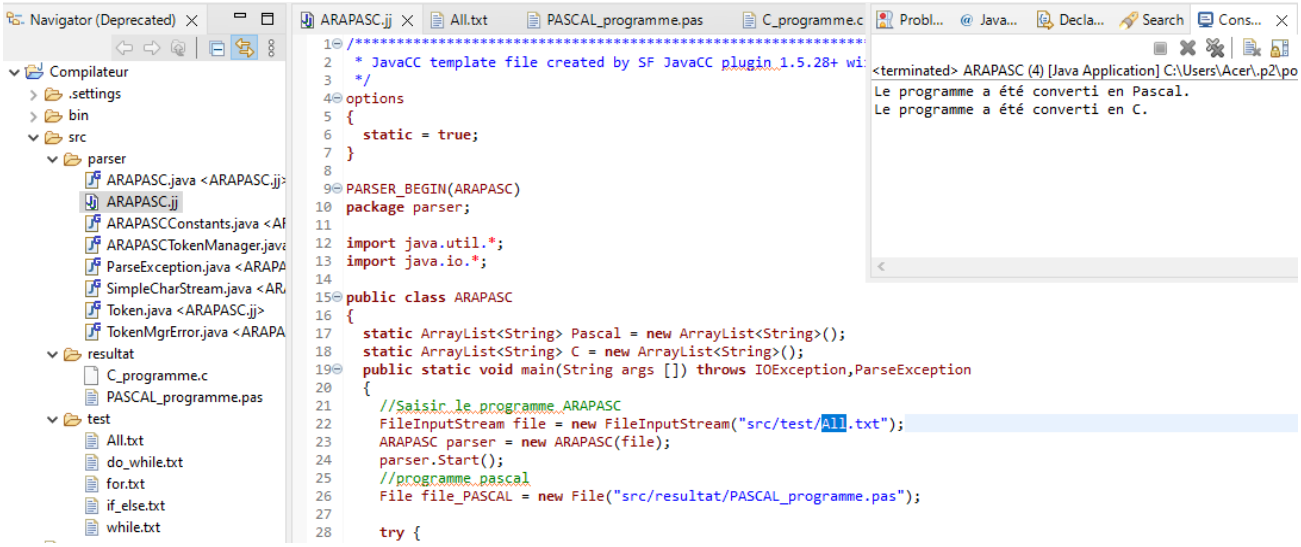
	LETTRE.c = "Z"
DECLARATION → moutaghaire : IDENTIFIERS : TYPE ;	DECLARATION.pas = "var : " + IDENTIFIERS.pas + ": "+TYPES.pas +"; " DECLARATION.c = TYPES.c + IDENTIFIERS.c + "; "
IDENTIFIERS → IDENTIFIER	IDENTIFIERS.pas = IDENTIFIER.pas IDENTIFIERS.c = IDENTIFIER.c
IDENTIFIERS → IDENTIFIER , IDENTIFIERS	IDENTIFIERS.pas = IDENTIFIER.pas +", "+ IDENTIFIERS.pas IDENTIFIERS.c = IDENTIFIER.c + ", " + IDENTIFIERS.c
TYPE → tabiaai	TYPE.pas = "$integer$" TYPE.c = "int"
TYPE → hakiki	TYPE.pas = "$float$" TYPE.c = "$float$"
TYPE → mantiki	TYPE.pas = "$boolean$" TYPE.c = "$bool$"
TYPE → harref	TYPE.pas = "$char$" TYPE.c = "$char$"
STATEMENT → SEQUENCE STATEMENT	STATEMENT.pas = SEQUENCE.pas + STATEMENT.pas STATEMENT.c = SEQUENCE.c + STATEMENT.c
STATEMENT → CONDITIONAL STATEMENT	STATEMENT.pas = CONDITIONAL.pas + STATEMENT.pas STATEMENT.c = CONDITIONAL.c + STATEMENT.c
STATEMENT → LOOP STATEMENT	STATEMENT.pas = LOOP.pas + STATEMENT.pas STATEMENT.c = LOOP.c + STATEMENT.c
STATEMENT → READ STATEMENT	STATEMENT.pas = READ.pas + STATEMENT.pas STATEMENT.c = READ.c + STATEMENT.c
STATEMENT → WRITE STATEMENT	STATEMENT.pas = WRITE.pas + STATEMENT.pas STATEMENT.c = WRITE.c + STATEMENT.c
STATEMENT → $\boldsymbol{\varepsilon}$	STATEMENT.pas = "" STATEMENT.c = ""
READ→ ikraa(IDENTIFIERS) ;	READ.pas = "$readln$("+IDENTIFIERS.pas + "); " READ.c = " $cin \gg$ " + IDENTIFIERS.c +" $\gg endl;$ "
WRITE → ouktoub(AFFICHAGE);	WRITE.pas = "$writeln$("+AFFICHAGE.pas + "); " WRITE.c = " $cout \ll$ " + AFFICHAGE.c +" $\ll endl;$ "
AFFICHAGE→MESSAGE	AFFICHAGE.pas = MESSAGE.pas AFFICHAGE.c = MESSAGE.c
AFFICHAGE→IDENTIFIERS	AFFICHAGE.pas = IDENTIFIERS.pas AFFICHAGE.c = IDENTIFIERS.c

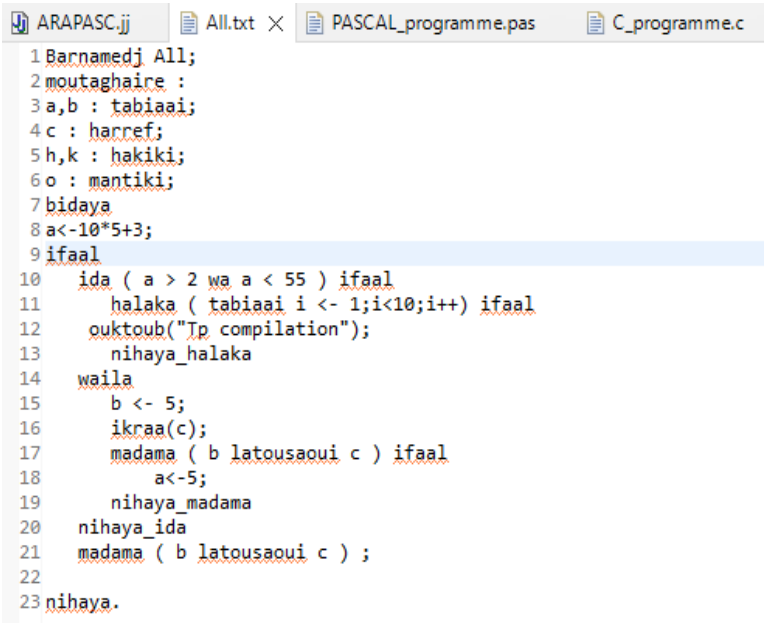
SEQUENCE→ STATEMENTS	SEQUENCE.pas = STATEMENTS.pas SEQUENCE.c = STATEMENTS.c
SEQUENCE→ IDENTIFIER AFFICTATION ;	SEQUENCE.pas IDENTIFIER.pas AFFICTATION.pas+ « ; » SEQUENCE.c IDENTIFIER.c AFFICTATION.c+ « ; »
STATEMENTS→TYPE IDENTIFIER <- INTEGER_LITERAL	STATEMENTS.pas = TYPE.pas +IDENTIFIER.pas + ":=" + INTEGER_LITERAL.pas STATEMENTS.c = TYPE.c +IDENTIFIER.c + " = " + INTEGER_LITERAL.c
STATEMENTS→TYPE IDENTIFIER <- FLOAT_LITERAL	STATEMENTS.pas = TYPE.pas +IDENTIFIER.pas + ":=" + FLOAT _LITERAL.pas STATEMENTS.c = TYPE.c +IDENTIFIER.c + " = " + FLOAT _LITERAL.c
STATEMENTS→TYPE IDENTIFIER	STATEMENTS.pas = TYPE.pas +IDENTIFIER.pas STATEMENTS.c = TYPE.c +IDENTIFIER.c
AFFICTATION → <- EXPRESSION	AFFICTATION.pas = ":=" + EXPRESSION.pas AFFICTATION.c = "=" + EXPRESSION.c
EXPRESSION → ADDITIVE	EXPRESSION.pas = ADDITIVE.pas EXPRESSION.c = ADDITIVE.c
ADDITIVE→ MULTIPLICATIVE + MULTIPLICATIVE	ADDITIVE.pas = MULTIPLICATIVE.pas + « + » + MULTIPLICATIVE.pas ADDITIVE.c = MULTIPLICATIVE.c + « + » + MULTIPLICATIVE.c
ADDITIVE→ MULTIPLICATIVE - MULTIPLICATIVE	ADDITIVE.pas = MULTIPLICATIVE.pas + « - » + MULTIPLICATIVE.pas ADDITIVE.c = MULTIPLICATIVE.c + « - » + MULTIPLICATIVE.c
ADDITIVE→ MULTIPLICATIVE	ADDITIVE.pas = MULTIPLICATIVE.pas ADDITIVE.c = MULTIPLICATIVE.c
MULTIPLICATIVE→ UNARY * UNARY	MULTIPLICATIVE.pas = UNARY.pas + "*" + UNARY.pas MULTIPLICATIVE.c = UNARY.c + "*" + UNARY.c
MULTIPLICATIVE→ UNARY / UNARY	MULTIPLICATIVE.pas = UNARY.pas + "/" + UNARY.pas MULTIPLICATIVE.c = UNARY.c + "/" + UNARY.c
MULTIPLICATIVE→ UNARY	MULTIPLICATIVE.pas = UNARY.pas MULTIPLICATIVE.c = UNARY.c

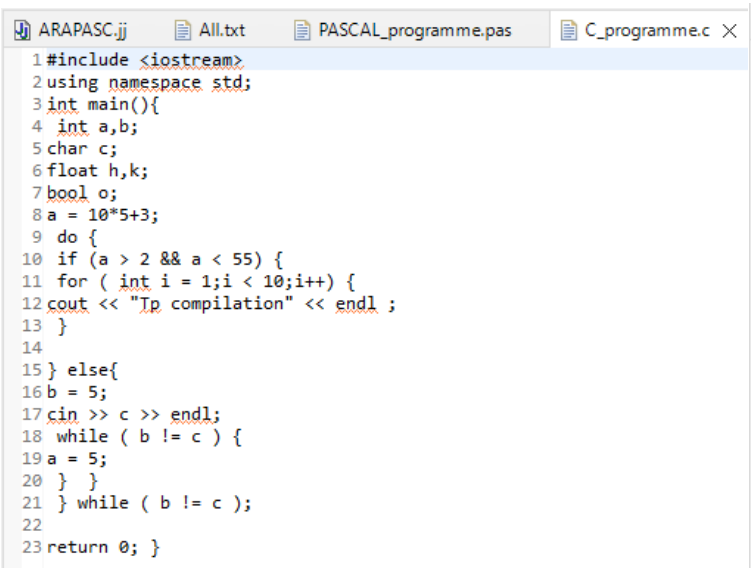
UNARY → INTEGER_LITERAL IDENTIFIER	UNARY.pas = INTEGER_LITERAL.pas + IDENTIFIER.pas UNARY.c = INTEGER_LITERAL.c + IDENTIFIER.c
UNARY → FLOAT_LITERAL IDENTIFIER	UNARY.pas = FLOAT _LITERAL.pas + IDENTIFIER.pas UNARY.c = FLOAT_LITERAL.c + IDENTIFIER.c
CONDITIONAL→ida(LOGICCONDITION)ifaa l STATEMENT nihaya_ida	CONDITIONAL.pas = "if(" + LOGICCONDITION.pas + ")do"+ STATEMENT.pas + « end ; » CONDITIONAL.c = "if(" + LOGICCONDITION.c + "){"+ STATEMENT.c + « } »
CONDITIONAL→ida(LOGICCONDITION)ifaa l STATEMENT waila STATEMENT nihaya_ida	CONDITIONAL.pas = "if(" + LOGICCONDITION.pas + ")do"+ STATEMENT.pas + « else »+ STATEMENT.pas + « end ; » CONDITIONAL.c = "if(" + LOGICCONDITION.c + "){"+ STATEMENT.c +« else »+ STATEMENT.pas + « } »
LOGICCONDITION→CONDITION LOGIC LOGICCONDITION	LOGICCONDITION.pas = CONDITION.pas + LOGIC.pas + LOGICCONDITION.pas LOGICCONDITION.c = CONDITION.c + LOGIC.c + LOGICCONDITION.c
LOGICCONDITION→CONDITION	LOGICCONDITION.pas = CONDITION.pas LOGICCONDITION.c = CONDITION.c
Logic→wa	Logic.pas = « and » Logic.c = « && »
Logic→aw	Logic.pas = « or » Logic.c = « || »
Logic→ ***la***	Logic.pas = « not » Logic.c = « ! »
CONDITION→EXPRESSION RELATIONSHIP EXPRESSION	CONDITION.pas = EXPRESSION.pas + RELATIONSHIP.pas + EXPRESSION.pas CONDITION.c = EXPRESSION.c + RELATIONSHIP.c + EXPRESSION.c
CONDITION→EXPRESSION	CONDITION.pas = EXPRESSION.pas CONDITION.c = EXPRESSION.c
RELATIONSHIP→ tousaoui	RELATIONSHIP.pas= « = » RELATIONSHIP.c= « = = »
RELATIONSHIP→ <	RELATIONSHIP.pas= « < » RELATIONSHIP.c= « < »
RELATIONSHIP→ >	RELATIONSHIP.pas= « > » RELATIONSHIP.c= « > »
RELATIONSHIP→ <=	RELATIONSHIP.pas= « <= » RELATIONSHIP.c= « <= »
RELATIONSHIP→ >=	RELATIONSHIP.pas= « >= » RELATIONSHIP.c= « >= »

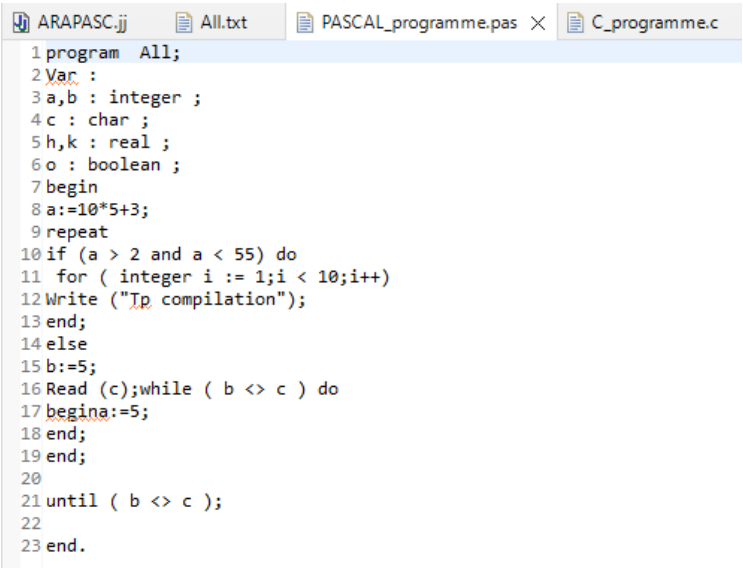
RELATIONSHIP→ latousaoui	RELATIONSHIP.pas= « <> » RELATIONSHIP.c= « != »
CREASE→IDENTIFIER ++	CREASE.pas = IDENTIFIER.pas + “++” CREASE.c = IDENTIFIER.c + “++”
CREASE→IDENTIFIER --	CREASE.pas = IDENTIFIER.pas + “--” CREASE.c = IDENTIFIER.c + “--”
CREASE→IDENTIFIER	CREASE.pas = IDENTIFIER.pas CREASE.c = IDENTIFIER.c
LOOP→madama(LOGICCONDITION)ifaal STATEMENT nihaya_madama	LOOP.pas = “while(”+LOGICCONDITION.pas+”)do” + STATEMENT.pas+ “end;” LOOP.c = “while(”+LOGICCONDITION.c+”){”+ STATEMENT.c+ “}”
LOOP→ifaal STATEMENT madama(LOGICCONDITION);	LOOP.pas = “do” + STATEMENT.pas+ “while(”+LOGICCONDITION.pas+”);” LOOP.c = “do” + STATEMENT.c+ “while(”+LOGICCONDITION.c+”);”
LOOP→halaka(STATEMENTS; LOGICCONDITION; CREASE)ifaal STATEMENT nihaya_halaka	LOOP.pas = “for(“+ STATEMENT.pas+”;”+ LOGICCONDITION.pas+”;”+ CREASE.pas+”)”+ STATEMENT.pas+” end;” LOOP.c = “for(“ + STATEMENT.c+”;”+ LOGICCONDITION.c+”;”+ CREASE.c+”) {”+ STATEMENT.c+” }”

9. Tests


Navigator (Deprecated)
Compilateur
.settings
bin
src
parser
ARAPASC.java <ARAPASC.jj>
ARAPASC.jj
ARAPASCConstants.java <AR
ARAPASCTokenManager.java
ParseException.java <ARAPA
SimpleCharStream.java <AR
Token.java <ARAPASC.jj>
TokenMgrError.java <ARAPA
resultat
C_programme.c
PASCAL_programme.pas
test
All.txt
do_while.txt
for.txt
if_else.txt
while.txt
ARAPASC.jj
All.txt
PASCAL_programme.pas
C_programme.c
Probl...
Java...
Decla...
Search
Cons...
<terminated> ARAPASC (4) [Java Application] C:\Users\Acer\.p2\po
Le programme a été converti en Pascal.
Le programme a été converti en C.
/* JavaCC template file created by SF JavaCC plugin 1.5.28+ wi
*/
options
{
static = true;
}
PARSER_BEGIN(ARAPASC)
package parser;
import java.util.*;
import java.io.*;
public class ARAPASC
{
static ArrayList<String> Pascal = new ArrayList<String>();
static ArrayList<String> C = new ArrayList<String>();
public static void main(String args []) throws IOException,ParseException
{
//Saisir le programme ARAPASC
FileInputStream file = new FileInputStream("src/test/All.txt");
ARAPASC parser = new ARAPASC(file);
parser.Start();
//programme pascal
File file_PASCAL = new File("src/resultat/PASCAL_programme.pas");
try {


ARAPASC.jj
All.txt
PASCAL_programme.pas
C_programme.c
Barnamedj All;
moutaghaire :
a,b : tabiaai;
c : harref;
h,k : hakiki;
o : mantiki;
bidaya
a<-10*5+3;
ifaal
ida (a > 2 wa a < 55) ifaal
halaka (tabiaai i <- 1;i<10;i++) ifaal
ouktoub("Tp compilation");
nihaya_halaka
waila
b <- 5;
ikraa(c);
madama (b latousaoui c) ifaal
a<-5;
nihaya_madama
nihaya_ida
madama (b latousaoui c) ;
nihaya.


ARAPASC.jj
All.txt
PASCAL_programme.pas
C_programme.c
#include <iostream>
using namespace std;
int main(){
int a,b;
char c;
float h,k;
bool o;
a = 10*5+3;
do {
if (a > 2 && a < 55) {
for (int i = 1;i < 10;i++) {
cout << "Tp compilation" << endl ;
}
} else{
b = 5;
cin >> c >> endl;
while (b != c) {
a = 5;
} }
} while (b != c);
return 0; }


ARAPASC.jj
All.txt
PASCAL_programme.pas
C_programme.c
program All;
Var :
a,b : integer ;
c : char ;
h,k : real ;
o : boolean ;
begin
a:=10*5+3;
repeat
if (a > 2 and a < 55) do
for (integer i := 1;i < 10;i++)
Write ("Tp compilation");
end;
else
b:=5;
Read (c);while (b <> c) do
begina:=5;
end;
end;
until (b <> c);
end.

Conclusion

Ce projet était une occasion pour mettre en oeuvre les concepts élémentaires de compilation de langage de programmation moderne. Nous nous sommes familiaris´ees avec des outils d'analyse lexicale et d'analyse syntaxique tel JAVACC. Le compilateur développé peut certainnement étre amélioré.