```
/**************************************************
 * Analyseur lexical pour Init version simple
 * fichier de description pour JFlex
 * produit anLexInit/ScannerInit.java
 * M. Nebut
 * 03/04 revu 09/09
 **************************************************/

/*****************************************************************
 * Première partie : code utilisateur inclus tel quel dans le fichier
 * .java généré. On met typiquement ici les déclarations de paquetage
 * (package) et les importations de classes (import).
 *****************************************************************/

// déclaration du paquetage auquel appartient la classe générée
package init.analyseurs;

%%

/*****************************************************************
 * Seconde partie : options et déclarations de macros.
 *****************************************************************/

/************************  Options  ************************/

// ATTENTION : le % doit toujours être en 1ère colonne

// la classe générée implantant l'analyseur s'appelle ScannerInit.java
%class ScannerInit
// et est publique
%public
// la cl. générée implante l'itf java_cup.runtime.Scanner fournie par Cup
%implements java_cup.runtime.Scanner
// pour utiliser les caractères unicode
%unicode
// pour garder trace du numéro de ligne du caractère traité
%line
// pour garder trace du numéro de colonne du caractère traité
%column
// l'an. lex. retourne des symboles de type java_cup.runtime.Symbol
%type java_cup.runtime.Symbol
// la fonction de l'analyseur fournissant le prochain Symbol s'appelle
// next_token...
%function next_token
// ... et lève une exception ScannerException en cas d'erreur lexicale
%yylexthrow{
ScannerException
%yylexthrow}
// action effectuée qd la fin du fichier à analyser est rencontrée
// le type EOF est généré automatiquement par Cup
%eofval{
  return new Symbole(TypeSymboles.EOF);
%eofval}
// code recopié dans la classe générée
%{
  private Symbole creerSymbole(String representation, int type) {
    return new Symbole(representation,type,yyline,yycolumn);
  }

  private Symbole creerSymbole(String representation, int type, Object valeur) {
    return new Symbole(representation,type,valeur,yyline,yycolumn);
  }
%}

/************************  définitions macros  ************************/

// une macro est une abbréviation pour une expression régulière

// syntaxe : <nom_macro> = <expr_reg>
// une macro peut être utilisée pour en définir une autre (de
// manière non récursive !) : il faut alors entourer son
// identificateur d'accolades.

finLigne = \r|\n|\r\n // convention Java
// | est le choix des expr reg de JFlex
// \n = retour-chariot sous Unix, \r = rc sous Windows
```



```
blancs = {finLigne} | [ \t\f]
// \t = tabulation, \f = form-feed
// [ \t\f] est une classe de caractères qui dénote soit " ", soit \t,
// soit \f

prog = "program" // une simple chaîne

identificateur = [:jletter:] [:jletterdigit:]*
// * est l'étoile des expressions régulières standard
// [:jletter:] représente n'importe quel caractère qui peut débuter un
// identificateur Java
// [:jletterdigit:] représente n'importe quel caractère qui peut
// suivre le 1er caractère d'un identificateur Java (donc une lettre
// ou un chiffre)

entier = [:digit:]+
// x+ signifie classiquement xx*
// [:digit:]  représente n'importe quel chiffre

affect = := // ou ":="
%%

/*****************************************************************/
/* Troisième partie : règles lexicales et actions. */
/*****************************************************************/

// syntaxe : { <nom_macro> | <expr_reg> } { <code_java> } un return ds
// le code Java correspond au retour d'un symbole (ici de type Symbol)
// résultat de la méthode d'analyse (ici la fonction next_token).
// S'il n'y a pas de return, on passe au symbole suivant.

{blancs} { /* on ignore les blancs */ }
{prog} { // on a reconnu le mot-clé program
  return creerSymbole("PROG",TypeSymboles.PROG);
}
"int" { // on a reconnu le mot-clé int
  return creerSymbole("INT",TypeSymboles.INT);
}
"read" {// on a reconnu un read
  return creerSymbole("READ",TypeSymboles.READ);
}

// on a défini tous les mot-clés qui pourraient préfixer un indentificateur :
// on définit donc maintenant seulement les identificateurs

{identificateur} {// on a reconnu un identificateur, par la suite il
                  // faudra lui associer par ex son nom : on utilise
                  // yytext() qui représente la portion du texte
                  // d'entrée reconnue
  return creerSymbole("IDENT",TypeSymboles.IDENT);
  //return creerSymbole("IDENT",TypeSymboles.IDENT,yytext());
}
{entier} {// on a reconnu un entier, par la suite il faudra lui
          // associer par ex sa valeur
  //return creerSymbole(TypeSymboles.ENTIER);
  return creerSymbole("ENTIER",TypeSymboles.ENTIER, new Integer(yytext()));
}
{affect} {// on a reconnu un opérateur d'affectation
  return creerSymbole("AFF",TypeSymboles.AFF);
}
";" {// on a reconnu un ";"
  return creerSymbole("FININSTR",TypeSymboles.FININSTR);
}
"," {// on a reconnu un ","
  return creerSymbole("SEPVAR",TypeSymboles.SEPVAR);
}
.|\n {// erreur : .|\n désigne n'importe quel caractère non reconnu
      // par une des règles précédentes
  throw new ScannerException("symbole inconnu, caractère " + yytext() +
                             " ligne " + yyline + " colonne " + yycolumn);
}
```

```
/*****************************************************
 * Description pour Cup d'une grammaire vide
 * Sert uniquement à définir les symboles de INIT
 *
 * 16/12/04 - revu 09/09
 * M. Nebut
 *
 *****************************************************/

package init.analyseurs;

/* terminaux (symboles retournés par l'analyseur lexical) */

terminal PROG,INT,READ,IDENT,ENTIER,AFF,FININSTR;
terminal SEPVAR;

/********* Non utilisé pour le TP1 *************/

/* non-terminaux de la grammaire */

non terminal texte;

/* la grammaire */

texte ::= ;
```



```
//----------------------------------------------------
// The following code was generated by CUP v0.11a beta 20060608
// Mon Sep 14 14:17:55 CEST 2009
//----------------------------------------------------

package init.analyseurs;

/** CUP generated class containing symbol constants. */
public class TypeSymboles {
  /* terminals */
  public static final int READ = 4;
  public static final int INT = 3;
  public static final int IDENT = 5;
  public static final int SEPVAR = 9;
  public static final int PROG = 2;
  public static final int EOF = 0;
  public static final int FININSTR = 8;
  public static final int ENTIER = 6;
  public static final int error = 1;
  public static final int AFF = 7;
}
```

init.analyseurs

Class Symbole

java.lang.Object
  └ java_cup.runtime.Symbol
    └ **init.analyseurs.Symbole**

public class **Symbole**
extends java_cup.runtime.Symbol

Un Symbole qui possède une représentation textuelle.

Version:
12/12/04 revu 09/09

Author:
M. Nebut

Field Summary

Fields inherited from class java_cup.runtime.Symbol

left, parse_state, right, sym, value

Constructor Summary

Symbole(java.lang.String representationTextuelle, int type, int line, int column)

Symbole(java.lang.String representationTextuelle, int type, java.lang.Object valeur, int line, int column)