**RAPPORT DE PROJET COMPIL**

| **Implémentation d’un compilateur pour le langage miniC** |
| --- |

**Réalisé par :**

* LAMDANI Wilem - Groupe : 2CS SIQ3
* BELKESSA Linda - Groupe : 2CS SIQ3

**Encadré par :**

* M. CHEBIEB Abdelkarim

**Introduction :**

Il est largement utilisé pour créer des applications, allant de programmes simples à de grands systèmes d'exploitation. En extrayant des règles de productions plus simples et limitées nous obtenons la grammaire suivante :

*Function→Type identif ier ( ArgList ) CompoundStmt*

*ArgList→Arg | ArgList , Arg | ϵ*

*Declaration→Type IdentList;*

*Type→int | float*

*IdentList→identifier , IdentList | identifier*

*Stmt→ForStmt | W hileStmt | Expr ; | IfStmt | CompoundStmt | Declaration| ϵ*

*ForStmt→for ( Expr ; OptExpr ; OptExpr ) Stmt*

*OptExpr→Expr | ϵ*

*WhileStmt→while ( Expr ) Stmt*

*IfStmt→if ( Expr ) Stmt ElsePart*

*ElseP art→elseStmt | ϵ*

*CompoundStmt→ StmtList*

*StmtList→StmtList Stmt | ϵ*

*Expr→identifier = Expr | Rvalue*

*Compare→ == | < | > | <= | >= | ! =*

*Mag→Mag + Term | Mag − Term | Term*

*Term→Term ∗ Factor | Term / Factor | Factor*

*Factor→( Expr ) | − Factor | + Factor | identifier | number*

A travers ce projet, nous allons construire un compilateur qui prend en charge cette grammaire et permet d’effectuer une analyse lexicale, syntaxique et sémantique sur les fichiers écrits dans ce langage à savoir MiniC.

Nous partons d’abord d’une conception de ces analyseurs, ensuite nous exposerons les étapes de la réalisation avec l’outil javacc, enfin nous prouverons l’efficacité du code par des tests qui tiendront compte de tous les cas pouvant être rencontrés.

**Présentation de l’outil javacc:**

JavaCC est un générateur de compilateur qui accepte les spécifications de langage au format de type BNF en entrée.

L'analyseur généré contient les composants de base du compilateur correspondant au fichier spécifié.

langage, qui comprend un analyseur lexical et un analyseur syntaxique. La figure 1 montre la structure d'un analyseur généré par JavaCC.

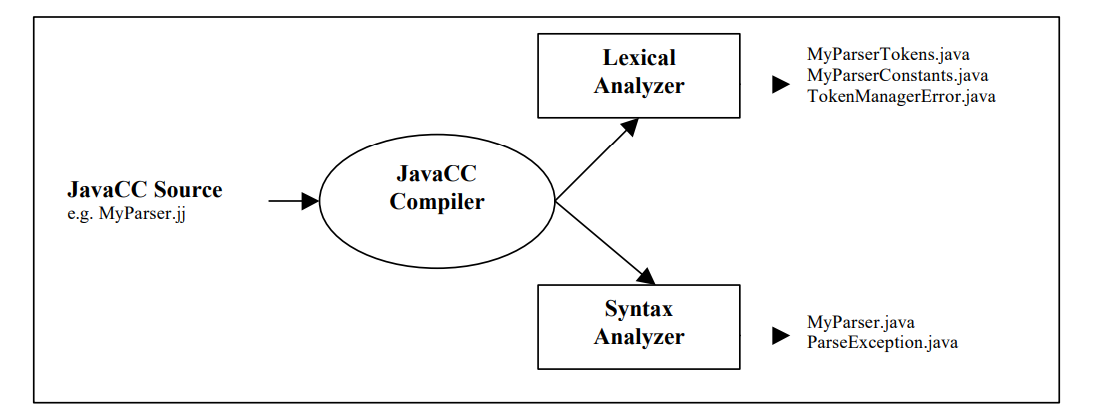


Figure 1 : Composants principaux d’un analyseur en javacc

**Analyse lexicale avec javacc :**

L'analyseur lexical de JavaCC s'appelle "TokenManager". Le TokenManager est utilisé pour regrouper les caractères d'un flux d'entrée dans des TOKENS selon des règles spécifiques. Chaque règle spécifiée dans TokenManager est associé à une d'expression :

SKIP : jeter l'expression correspondante

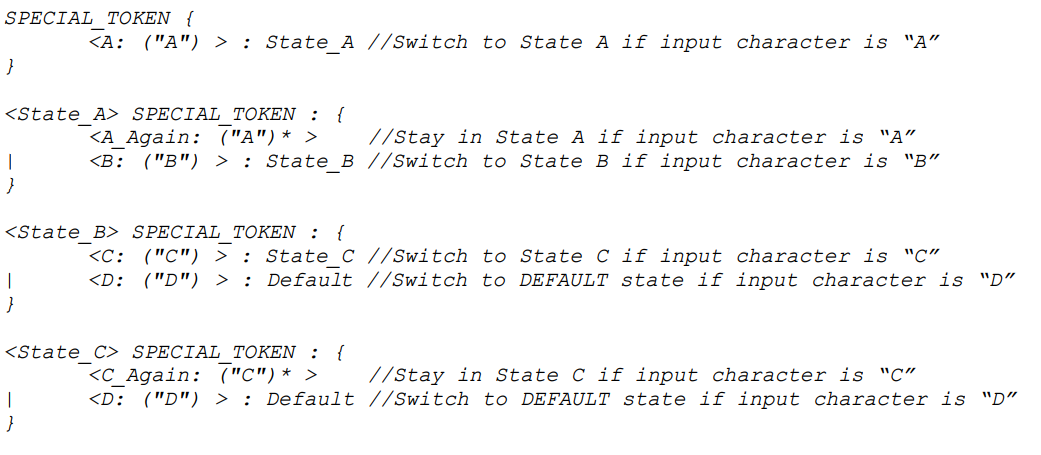
PLUS : continuez à prendre l'expression correspondante suivante pour créer une expression plus longue

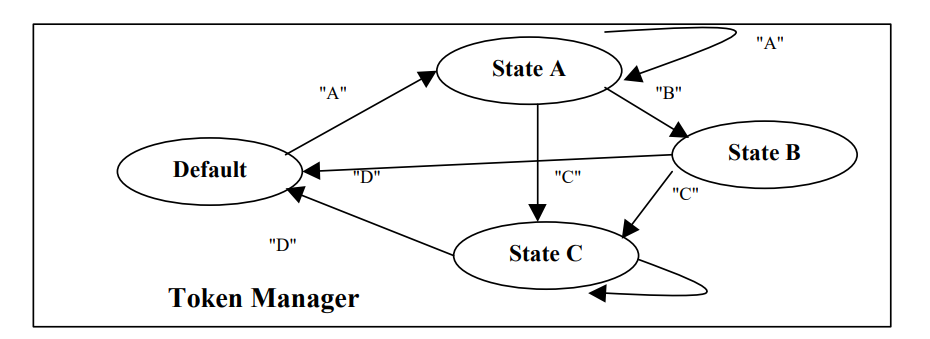
TOKEN : crée un jeton à l'aide de l'expression correspondante et l'envoie à l'analyseur syntaxique

SPECIAL\_TOKEN : crée un jeton avec l'expression de correspondance et l'envoie éventuellement à la syntaxe

Le TokenManager est une machine à états qui se déplace entre différents états lexicaux pour classer

jetons. La figure 2 illustre un exemple de fonction de machine d'état de l'analyseur lexical. Quand l'analyseur démarre, il est dans l'état par défaut qui attend les entrées. Si l'entrée est un caractère A, il passe à l'état A. Puis à partir de l'état A, si l'entrée est le caractère "A", il reste dans le même état. Cependant, lorsque l'entrée est le caractère "B" ou "C", le système passe aux états correspondants. Si la machine d'état est confrontée à une situation non spécifiée, telle que frapper un caractère "E" dans l'état C, il génère des erreurs lexicales. Le segment de code suivant implémente une partie du TokenManager pour l'analyseur lexical décrit dans JavaCC :



**

*Figure 2 : Fonctionnement de l’analyseur lexicale de javacc*

Appliquons ces principes sur la grammaire de MiniC :

1- Recensement des caractères SKIP, TOKEN

| Type | label / étiquette | Expression régulière |
| --- | --- | --- |
| SKIP | / | "\t"  "\n"  "\r"  <"//" (~["\n","\r"])\* ("\n" | "\r" "\r\n")>  <"/\*" (~["\*"])\* "\*" ("\*"  ~["\*","/"] (~["\*"])\* "\*")\* "/"> |
| TOKEN | TYPE  IF  #INT  #FLOAT  WHILE  FOR  ELSE | <TYPE: <INT> <FLOAT> >  <IF: "if" > <#INT: "int" > <#FLOAT: "float" > <WHILE: "while" > <FOR: "for" > |
| TOKEN | ID  NUMBER  #LETTER  #DIGIT  #ZERO | <ID: <LETTER> (<LETTER> | <DIGIT> | <ZERO>)\* >  <NUMBER: <DIGIT> (<DIGIT> | <ZERO>)\* | <ZERO> >  <#LETTER: ["a"-"z","A"-"Z"] >  <#DIGIT: ["1"-"9"] > <#ZERO: "0" > |
| TOKEN | GPAR  DPAR  VIRGULE GACC  DACC  POINTVIRGULE  AFFECT  EQ  LT  LTE  GT  GTE  NEQ  ADD  SUB  MUL  DIV | <GPAR: "(" > |  <DPAR: ")" > |  <VIRGULE: "," > |  <GACC: "{" > |  <DACC: "}" > |  <POINTVIRGULE: ";" > |  <AFFECT: "=" > |  <EQ: "==" > |  <LT: "<" > |  <LTE: "<=" > |  <GT: ">" > |  <GTE: ">=" > |  <NEQ: "!=" > |  <ADD: "+" > |  <SUB: "-" > |  <MUL: "\*" > |  <DIV: "/" > |

**Analyse syntaxique avec javacc :**

L'analyseur de syntaxe de JavaCC est un analyseur LL(k) à descente récursive. Ce type d'analyseur utilise k nombre de jetons de lookahead pour générer un ensemble de productions mutuellement exclusives, qui reconnaissent la langue étant analysée par l'analyseur.

Par défaut, l'analyseur de syntaxe de JavaCC définit k sur 1, mais les développeurs peuvent remplacer le nombre de jetons d'anticipation par n'importe quel nombre arbitraire pour correspondre correctement les productions.

Les analyseurs LL(k) ne permettent que la récursivité à droite dans la production.

Notre grammaire doit d’abord etre transformée :

* Rendre la grammaire propre :
  + Eliminer les productions inutiles
  + Eliminer les productions unitaires
  + Eliminer les epsilon-productions
* Eliminer la récursivité à gauche
* Eliminer la factorisation à gauche

1- Rendre la grammaire propre ( Eliminer les productions unitaires et epsilon-productions)

Function -> Type identifier ( ArgList ) CompoundStmt

ArgList -> Arg | ArgList , Arg | ϵ

Declaration -> Type IdentList;

Type -> int | float

IdentList -> identifier , IdentList | identifier

Stmt -> ForStmt | WhileStmt | Expr ; | IfStmt | CompoundStmt | Declaration| ϵ

ForStmt -> for ( Expr ; OptExpr ; OptExpr ) Stmt

OptExpr -> Expr | ϵ

WhileStmt -> while ( Expr ) Stmt

IfStmt -> if ( Expr ) Stmt ElsePart

ElsePart -> elseStmt | ϵ

CompoundStmt -> StmtList

StmtList -> StmtList Stmt | ϵ

Expr -> identifier = Expr | Rvalue

Compare -> == | < | > | <= | >= | ! =

Mag -> Mag + Term | Mag − Term | Term

Term -> Term ∗ Factor | Term / Factor | Factor

Factor -> ( Expr ) | − Factor | + Factor | identifier | number

Après transformation :



2- Elimination de la récursivité gauche ainsi que les production gauches ( transformation en LL(1))

Function -> Type identifier ( ArgList ) CompoundStmt

ArgList -> Type identifier ArgList1

Arg -> Type identifier

Declaration -> Type IdentList ;

Type -> int

| float

IdentList -> identifier IdentList1

Stmt -> for ( Expr ; OptExpr ; OptExpr ) Stmt

| while ( Expr ) Stmt

| Expr ;

| if ( Expr ) Stmt ElsePart

| { StmtList }

| int IdentList ;

| float IdentList ;

| ;

ForStmt -> for ( Expr ; OptExpr ; OptExpr ) Stmt

OptExpr -> Term OptExpr3

| Mag OptExpr2

| identifier OptExpr1

| Rvalue Compare Mag

| ( Expr )

| - Factor

| + Factor

| number

| ϵ

WhileStmt -> while ( Expr ) Stmt

IfStmt -> if ( Expr ) Stmt ElsePart

ElsePart -> else Stmt

| ϵ

CompoundStmt -> { StmtList }

StmtList -> StmtList1

Expr -> Term Expr3

| Mag Expr2

| identifier Expr1

| Rvalue Compare Mag

| ( Expr )

| - Factor

| + Factor

| number

Rvalue -> Term Rvalue2 Rvalue3

| Mag Rvalue1 Rvalue3

| ( Expr ) Rvalue3

| - Factor Rvalue3

| + Factor Rvalue3

| identifier Rvalue3

| number Rvalue3

Compare -> ==

| <

| >

| <=

| >=

| !=

Mag -> Term Mag2 Mag3

| ( Expr ) Mag3

| - Factor Mag3

| + Factor Mag3

| identifier Mag3

| number Mag3

Term -> ( Expr ) Term2

| - Factor Term2

| + Factor Term2

| identifier Term2

| number Term2

Factor -> ( Expr )

| - Factor

| + Factor

| identifier

| number

IdentList1 -> , IdentList

| ϵ

OptExpr1 -> = Expr

| ϵ

OptExpr2 -> + Term

| - Term

OptExpr3 -> \* Factor

| / Factor

Expr1 -> = Expr

| ϵ

Expr2 -> + Term

| - Term

Expr3 -> \* Factor

| / Factor

Rvalue1 -> + Term

| - Term

Rvalue2 -> \* Factor

| / Factor

Mag1 -> + Term

| - Term

Mag2 -> \* Factor

| / Factor

Term1 -> \* Factor

| / Factor

ArgList1 -> , Arg ArgList1

| ϵ

StmtList1 -> Stmt StmtList1

| ϵ

Rvalue3 -> Compare Mag Rvalue3

| ϵ

Mag3 -> Mag1 Mag3

| ϵ

Term2 -> Term1 Term2

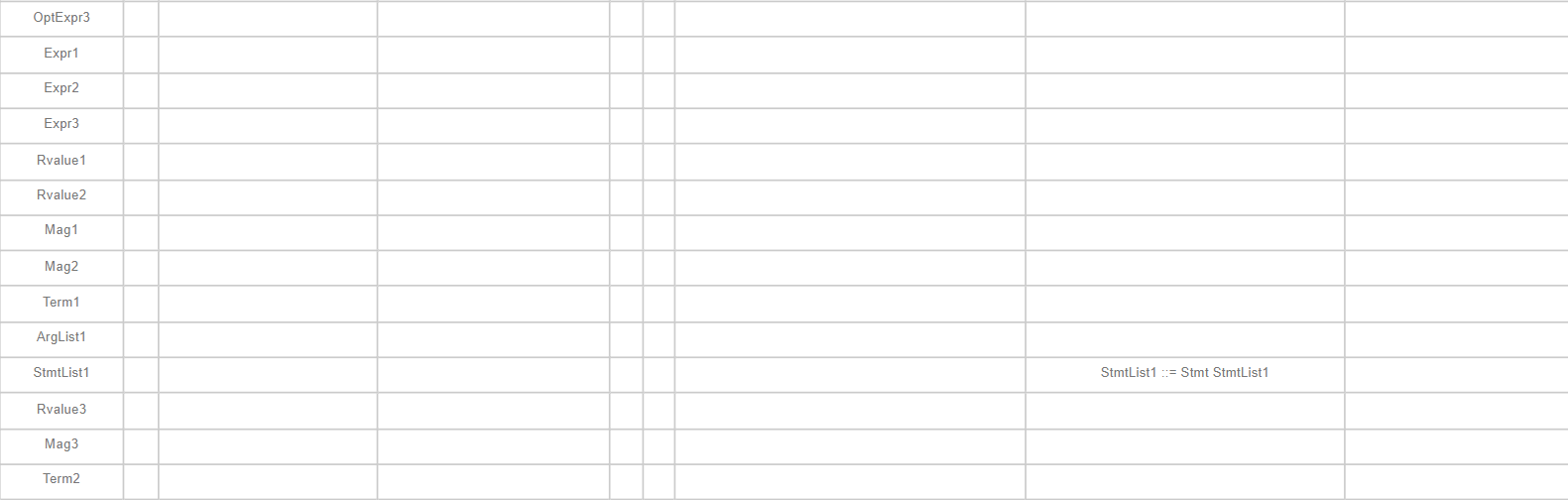
| ϵ

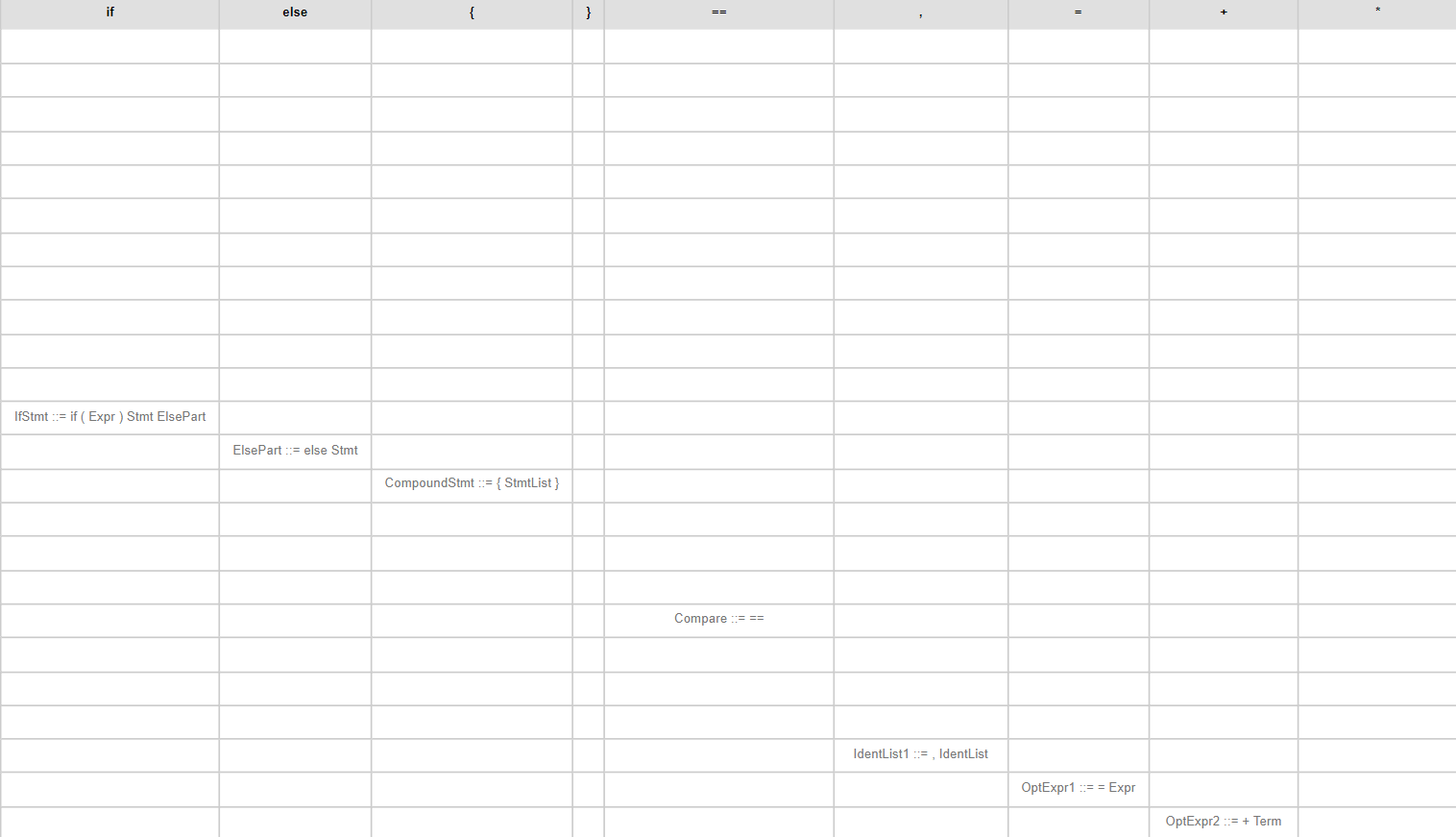
**Génération de la table LL(1) :**

**1 - Table des débuts/suivants :**

| **Non-terminal** | **Débuts** | **Suivants** |
| --- | --- | --- |
| **S** | **int** |  |
|  |  |  |
| **Function** | **int** | **$** |
|  |  |  |
| **ArgList** | **int** | **)** |
|  |  |  |
| **Arg** | **int** | **,** |
|  |  |  |
| **Declaration** | **int** |  |
|  |  |  |
| **Type** | **int** | **identifier** |
|  |  |  |
| **IdentList** | **identifier** | **;** |
|  |  |  |
| **Stmt** | **for** | **else, for** |
|  |  |  |
| **ForStmt** | **for** |  |
|  |  |  |
| **OptExpr** | **(** | **), ;** |
|  |  |  |
| **WhileStmt** | **while** |  |
|  |  |  |
| **IfStmt** | **if** |  |
|  |  |  |
| **ElsePart** | **else** |  |
|  |  |  |
| **CompoundStmt** | **{** | **$** |
|  |  |  |
| **StmtList** | **for** | **}** |
|  |  |  |
| **Expr** | **(** | **;, )** |
|  |  |  |
| **Rvalue** | **(** |  |
|  |  |  |
| **Compare** | **==** | **(** |
|  |  |  |
| **Mag** | **(** | **==** |
|  |  |  |
| **Term** | **(** | **\*, +** |
|  |  |  |
| **Factor** | **(** | **), ;, ==, +, \*** |
|  |  |  |
| **IdentList1** | **,** | **;** |
|  |  |  |
| **OptExpr1** | **=** |  |
|  |  |  |
| **OptExpr2** | **+** |  |
|  |  |  |
| **OptExpr3** | **\*** | **), ;** |
|  |  |  |
| **Expr1** | **=** |  |
|  |  |  |
| **Expr2** | **+** |  |
|  |  |  |
| **Expr3** | **\*** | **;, )** |
|  |  |  |
| **Rvalue1** | **+** |  |
|  |  |  |
| **Rvalue2** | **\*** | **==** |
|  |  |  |
| **Mag1** | **+** | **+** |
|  |  |  |
| **Mag2** | **\*** | **+** |
|  |  |  |
| **Term1** | **\*** | **\*** |
|  |  |  |
| **ArgList1** | **,** | **)** |
|  |  |  |
| **StmtList1** | **for** | **}** |
|  |  |  |
| **Rvalue3** | **==** |  |
|  |  |  |
| **Mag3** | **+** | **==** |
|  |  |  |
| **Term2** | **\*** | **\*, +** |









**Analyse sémantique avec javacc :**

L'analyseur sémantique est la dernière partie du frontal d'un compilateur. Avant que le compilateur puisse produire une version exécutable du code, il doit constituer une large base de connaissances sur les détails encodés dans le programme. L'entrée de la phase d'analyse sémantique est le programme analysé produit par l'analyseur syntaxique. Notre analyseur sémantique crée un arbre syntaxique abstrait pour le code source.

Pour générer l'analyseur sémantique, nous devons d'abord utiliser jjtree pour créer le fichier .jj, puis nous utilisons ce fichier .jj comme fichier d'entrée pour javacc pour créer l'analyseur sémantique.