

Rapport de projet:

Élaboré par :

Eya besbes

Mayssa Hemdena

Fatma Mannai

**Table des matières:**

[I. Introduction: 3](#_Toc95932778)

[II. Analyse lexicale: 3](#_Toc95932779)

[A. Description du langage: 3](#_Toc95932780)

[B. Les automates : 5](#_Toc95932781)

[III. Analyse syntaxique: 7](#_Toc95932782)

[A. Verification du type de la grammaire : 7](#_Toc95932783)

[1. Analyseur Syntaxiqe LL1: 7](#_Toc95932784)

[2. Analyseur SLR: 9](#_Toc95932785)

[IV. Analyse sémantique: 16](#_Toc95932786)

[V. Génération de code: 16](#_Toc95932787)

# Introduction:

Ce projet a été conçu par Eya Besbes, Mayssa Hemdana et Fatma Mannai. L’objectif de ce projet est de proposer un langage informatique de type impératif et de construire un compilateur, qui partant d’un programme source, génère un programme cible en langage Java, compilable et exécutable.la réalisation du projet se fera en Java à l’aide de l’outil INTELLIJ et comportera trois phases respectivement une phase d’analyse lexicale , une phase d’analyse syntaxique et une phase d’analyse sémantique.

# Analyse lexicale:

## Description du langage:

Le langage de ce projet comporte les éléments suivants:

* Des variables de type entier, reel et booléen .
* Des opérateurs arithmétiques, logiques et relationnels. des parenthèses et des accoladesDes instruction d’affectation, de lecture et d’écriture Des structures de contrôle simples et imbriquées.

La grammaire de ce langage est décrite par les règles de production suivantes:

D-> ID : Type PV D

D-> I

I-> If ( E ) Then { I } I

I-> If ( E ) Then { I } Else { I } I

I-> While ( E ) Do { I } I

I-> ID OPPAff terme PV I

I-> write E PV I

I-> read E PV I

I-> ''

E-> ( E ) K

E-> { E } K

E-> non ( E ) K

E-> terme K

E-> ''

K-> OpRel E

K-> OpAr E

K-> opLog E

K-> OPPAff E

K-> ''

terme-> entier

terme-> ID

terme-> Booleen

terme-> Reel

terme-> Chaine

où If, then,Else,While, id, Entier, Reel, Booleen, … sont les unités lexicales définies par les expressions régulières suivantes:

oprel —> <| <= | = | < > | > | >=

opaff —> :=

opAr —> +| -| \* | /

opLog —> ou | et

Else—>Else

If—> If

then—> then

While —>While

Do —> Do

entier—> chiffre+

réel—>chiffre+.chiffre+

Booléen—> vrai | faux

type —> integer |char | float | bool | Str

chiffre —>0|1|2|3|4|5|6|7|8|9

chaine—>lettre+

lettre —> [A..Z] | [a…z]

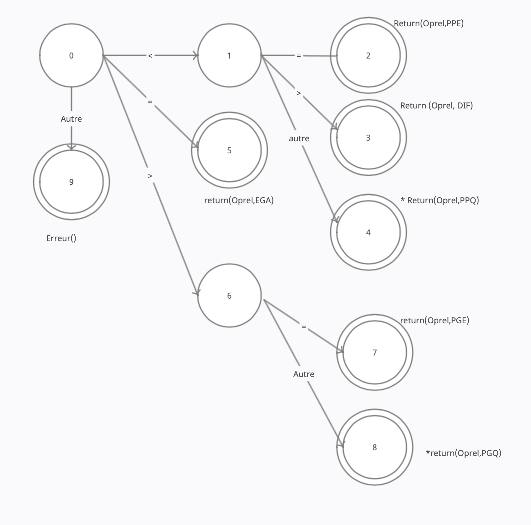
id —> lettre(lettre|chiffre)\*

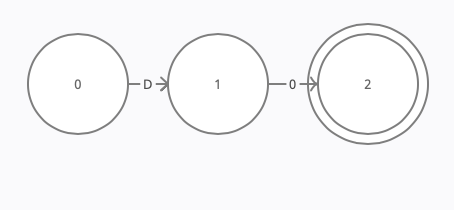
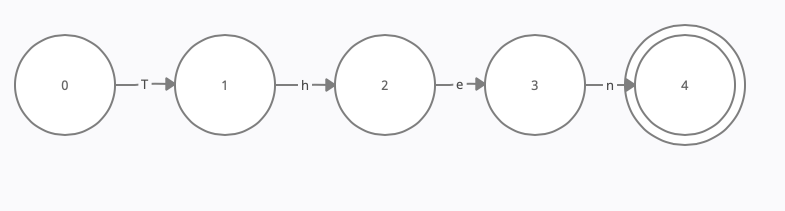
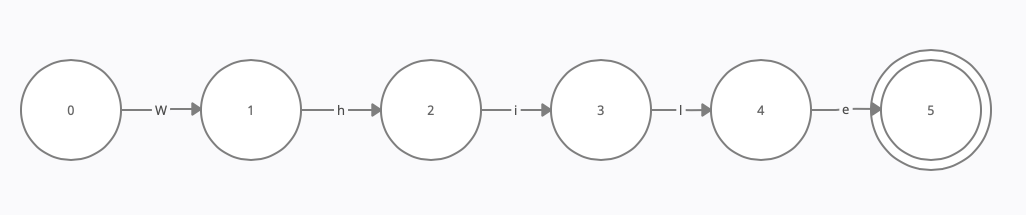
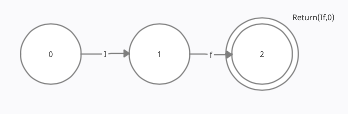
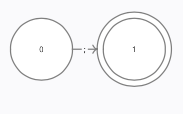
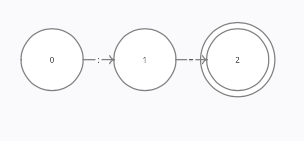
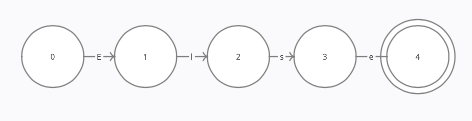
pv —> ;

**NB:**

Vous trouverez le code source de l’analyse lexicale sur machine sur l’outil INTELLIJ ( Dans le package Compilation

## Les automates :



Return(Then,0)

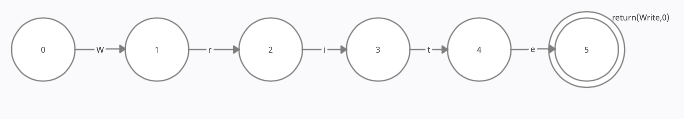
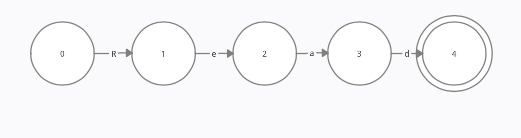
Return(While,0)

Return(Else,0)

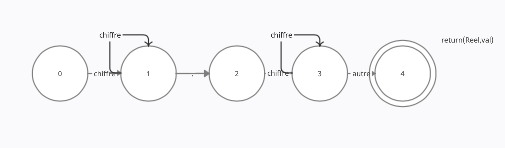
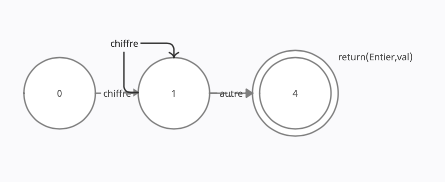
Return(Do,0)

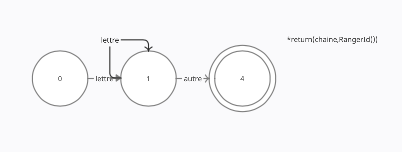
Return(opaff, :=)

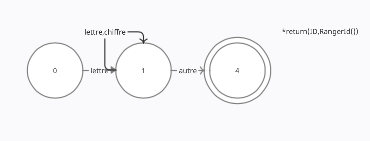
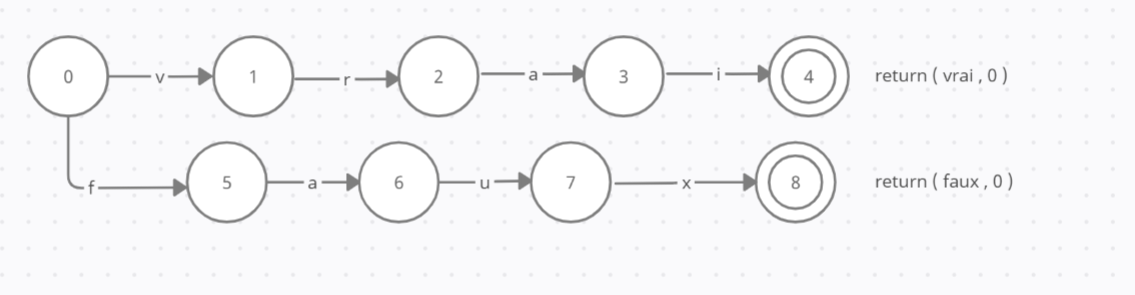
Return(pv, ;)

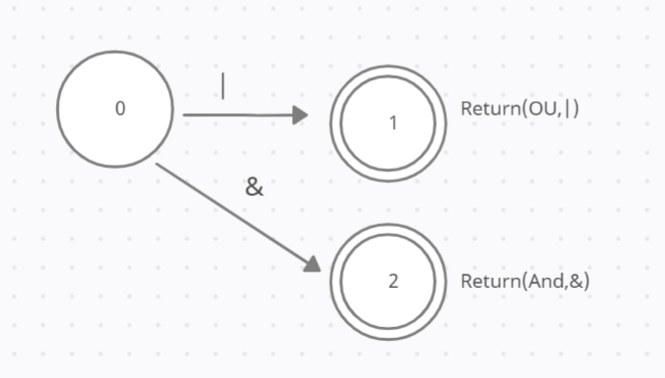
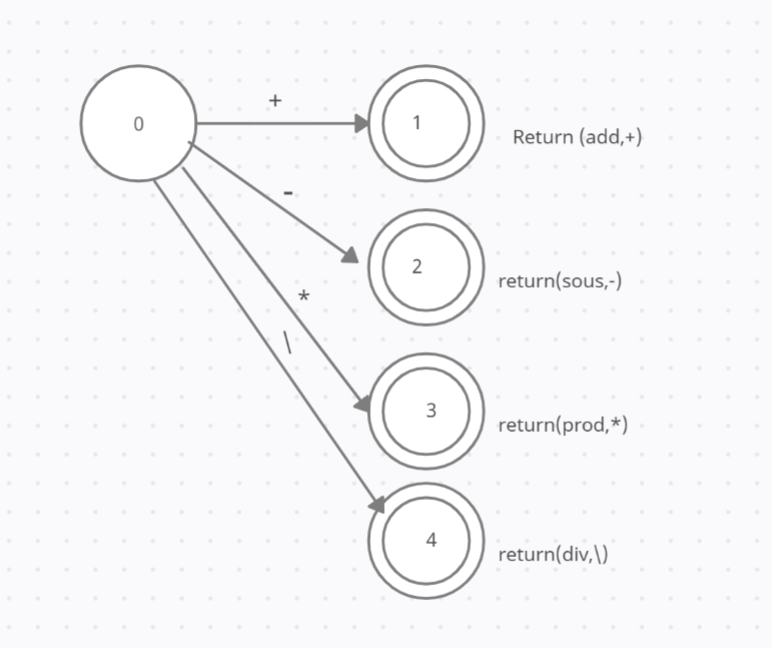
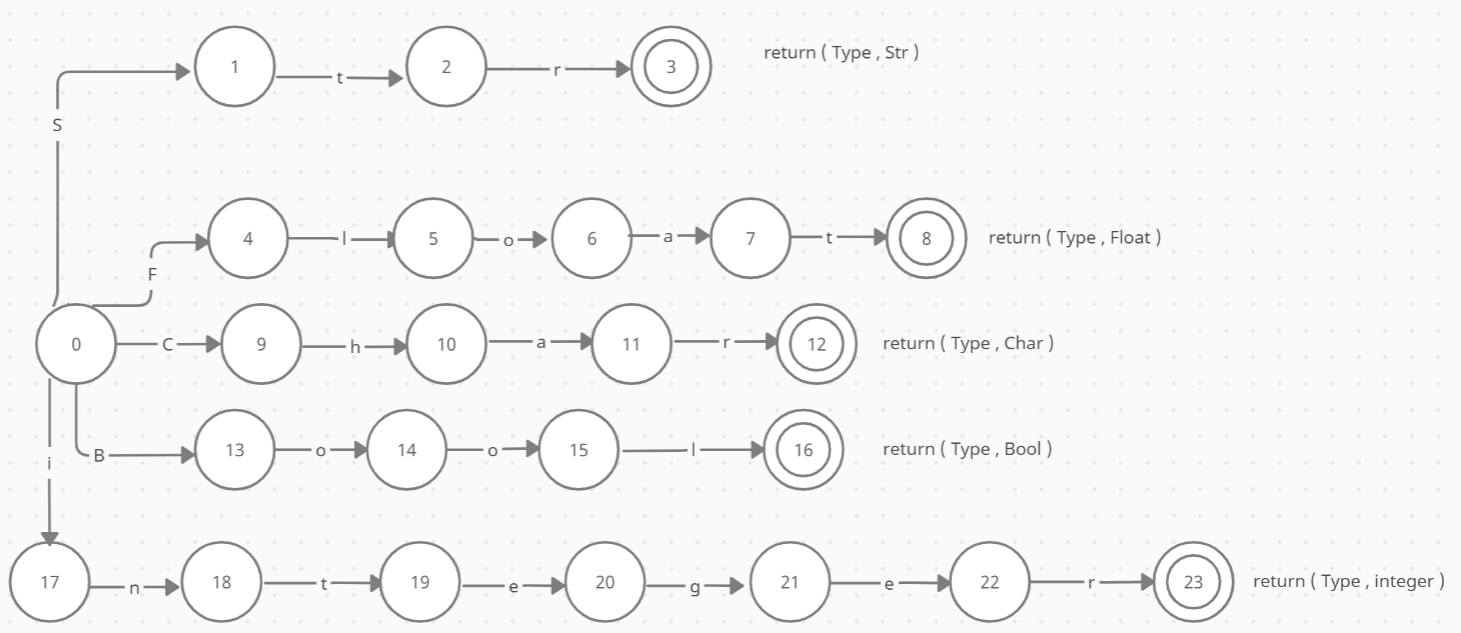


Return(Read,0)









# Analyse syntaxique:

## Verification du type de la grammaire :

### Analyseur Syntaxiqe LL1:

on doit vérifier si cette grammaire est de type LL1.Pour cela, on doit construire un analyseur prédictif non récursif. De ce fait, on doit éliminer la récursivité à gauche et puis enlever l’ambiguïté.

La grammaire de ce langage n’est pas récursive à gauche car ne elle comporte pas des règles de production de la forme :

**A—> αß1| αß2| αß3|…..| αßn| α1| α2| ….| αm**

De même elle n’est pas ambigue car elle ne comporte pas des règles de production de la forme:

**A—> Aα1|Aα2|…… |Aαn|ß1|ß2|….|ßm**

#### Calcul des premiers et des suivants:

##### Calcul des premiers:

Premier(D) = {id,If,while,write,read,''} // dans ce langage ‘’ désigne epsilon

Premier(I)= {If,while,id,write,read,''}

Premier(E)={(,{,non,'',entier,id,booléen,réel,chaîne}

Premier(K)= {oprel,opar,opLog,opaff,''}

Premier(terme)= {entier,id,booléen,réel,chaîne}

##### Calcul de suivants:

Suivant(D)= {$}

Suivant(I){$,},Else}

Suivant (E) = {),pv,}}

Suivant(K) = {),pv,}}

Suivant(terme)= {pv,oprel,opar,opLog,opaff,),}}

##### Table LL1:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **VN** | **id** | **:** | **type** | **pv** | **If** | **(** | **)** | **then** | **{** |
| **D** | D-> id : type pv D D-> I |  |  |  | D-> I |  |  |  |  |
| **I** | I-> id opaff terme pv I |  |  |  | I-> If ( E ) then { I } I I-> If ( E ) then I Else { I } I |  |  |  |  |
| **E** | E-> terme K |  |  | E-> '' | E-> ( E ) K |  | E-> '' |  | E-> { E } K |
| **K** |  |  |  | K-> '' |  |  | K-> '' |  |  |
| **terme** | terme-> id |  |  | **sync** |  |  | **sync** |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **VN** | **}** | **Else** | **while** | **Do** | **opaff** | **write** | **read** | **non** |
| **D** |  |  | D-> I |  |  | D-> I | D-> I |  |
| **I** | I-> '' | I-> '' | I-> while ( E ) Do { I } I |  |  | I-> write E pv I | I-> read E pv I |  |
| **E** | E-> '’ |  |  |  |  |  |  | E-> non ( E ) K |
| **K** | K-> '' |  |  |  | K-> opaff E |  |  |  |
| **terme** | **sync** |  |  |  | **sync** |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **VN** | **oprel** | **opar** | **opLog** | **entier** | **booléen** | **réel** | **chaîne** | **$** |
| **D'** |  |  |  |  |  |  |  | **Sync** |
| **D** |  |  |  |  |  |  |  | D-> I |
| **I** |  |  |  |  |  |  |  | I-> '' |
| **E** |  |  |  | E-> terme K | E-> terme K | E-> terme K | E-> terme K |  |
| **K** | K-> oprel E | K-> opar E | K-> opLog E |  |  |  |  |  |
| **terme** | **Sync** | **sync** | **sync** | terme-> entier | terme-> booléen | terme-> réel | terme->chaîne |  |

##### Exemple d’éxécution:

##### 

On constate qu’il existe deux règles de production dans une même case du tableau alors cette grammaire n’est pas de type LL(1) .On passe alors à l’analyseur SLR.

### Analyseur SLR:

#### Augmentation de la grammaire:

D'-> D

D-> ID : Type PV D

D-> I

I-> If ( E ) Then { I } I

I-> If ( E ) Then { I } Else { I } I

I-> While ( E ) Do { I } I

I-> ID OPPAff terme PV I

I-> write E PV I

I-> read E PV I

I-> ''

E-> ( E ) K

E-> { E } K

E-> non ( E ) K

E-> terme K

E-> ''

K-> OpRel E

K-> OpAr E

K-> opLog E

K-> OPPAff E

K-> ''

terme-> entier

terme-> ID

terme-> Booleen

terme-> Reel

terme-> Chaine

#### Calcul des premiers et des suivants:

##### Calcul des premiers:

Premier(D’) = {id,If,while,write,read,''}

Premier(D) = {id,If,while,write,read,''} // dans ce langage ‘’ désigne epsilon

Premier(I)= {If,while,id,write,read,''}

Premier(E)={(,{,non,'',entier,id,booléen,réel,chaîne}

Premier(K)= {oprel,opar,opLog,opaff,''}

Premier(terme)= {entier,id,booléen,réel,chaîne}

##### Calcul des suivants:

Suivant(D’)= {$}

Suivant(D)= {$}

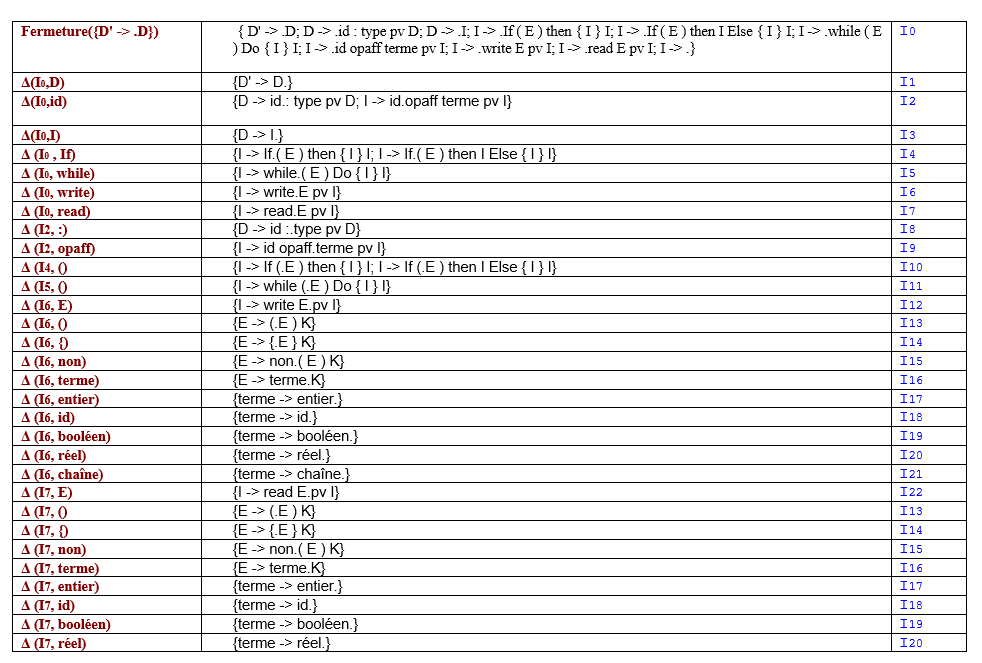
Suivant(I){$,},Else}

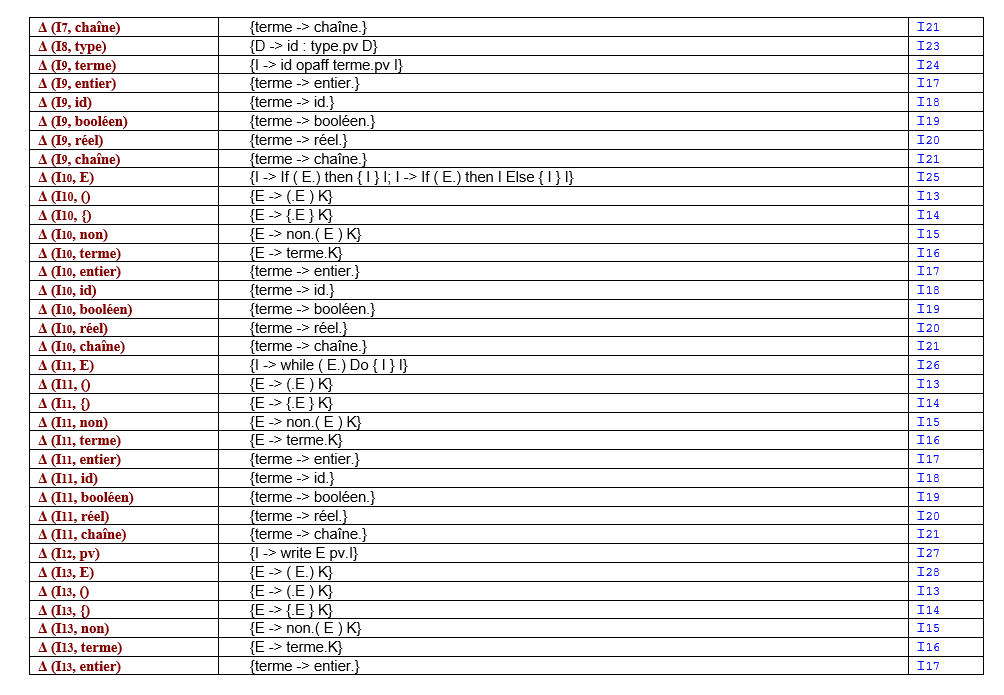
Suivant (E) = {),pv,}}

Suivant(K) = {),pv,}}

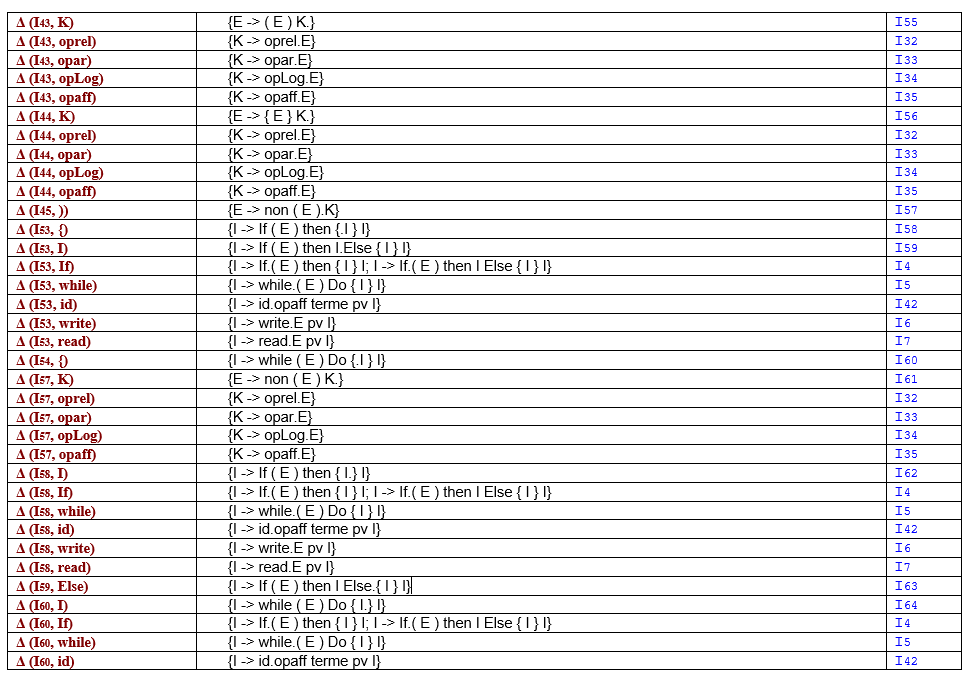
Suivant(terme)= {pv,oprel,opar,opLog,opaff,),}}

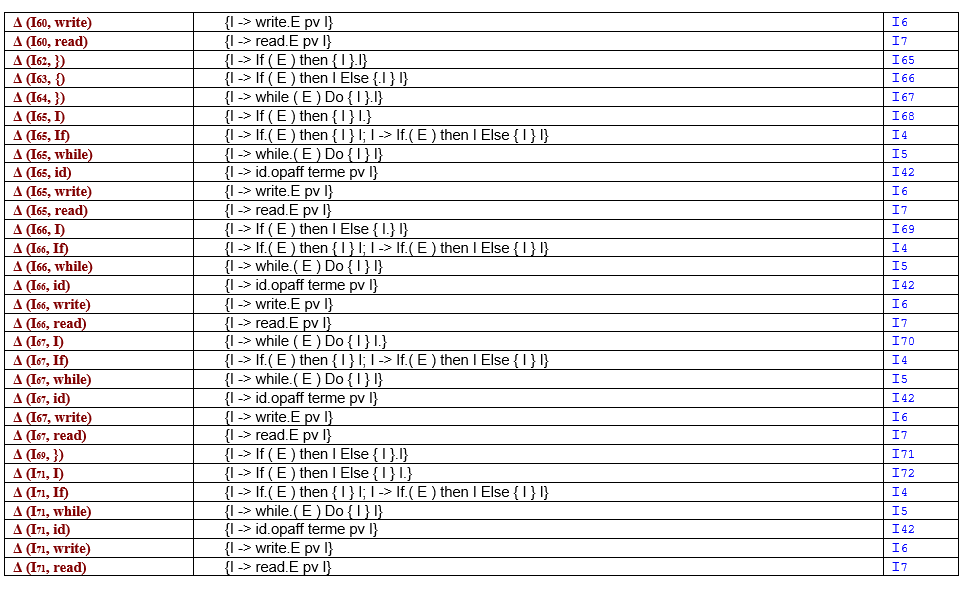
##### Calcul des transitions:



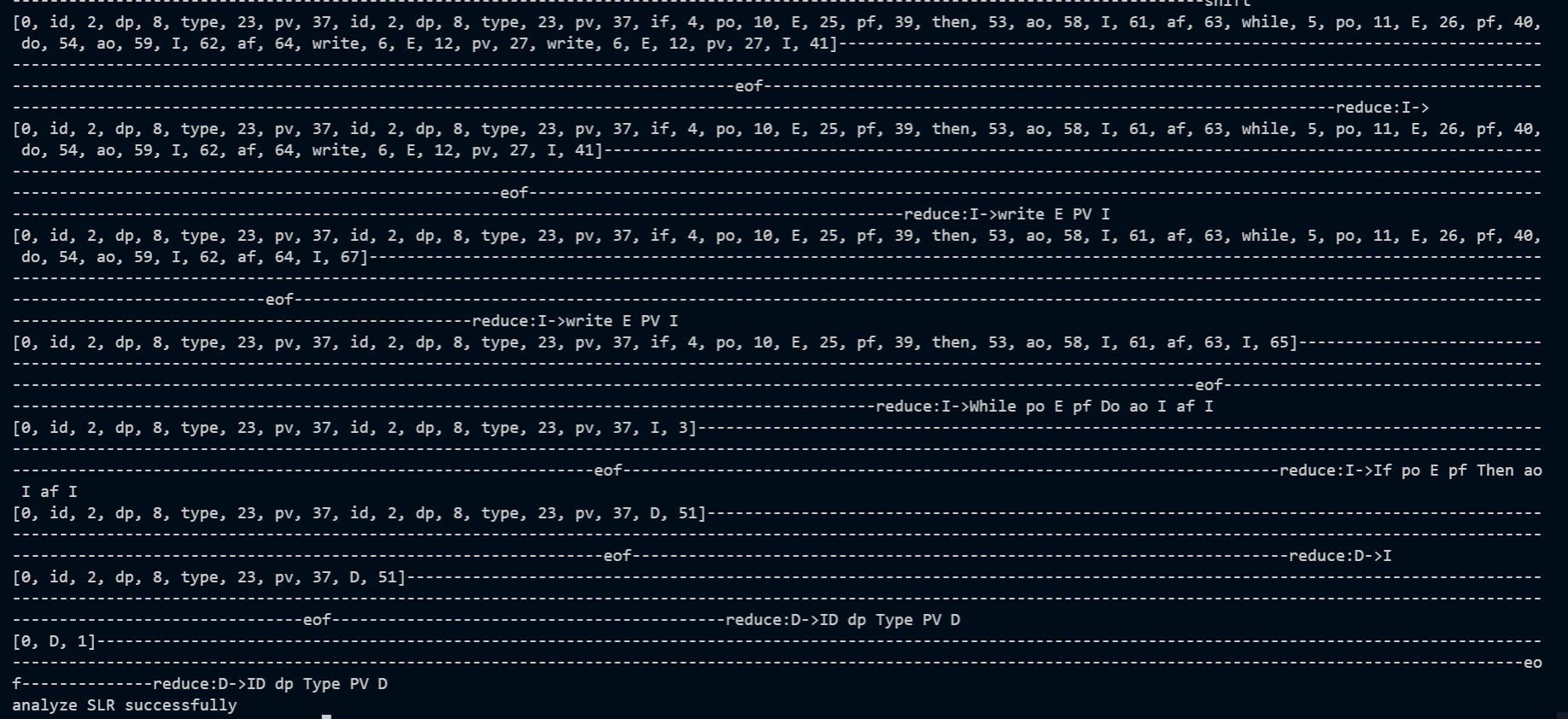








##### Exécution du code:



# Analyse sémantique:

Le rôle de l’analyse sémantique est de vérifier certaines contraintes. Elle se fait généralement en même temps que l’analyse syntaxique. Dans ce projet, on va s’intéresser uniquement au contrôle de type.

|  |  |  |
| --- | --- | --- |
| Production | Règle sémantique | Traduction avec une pile |
| Terme -> entier | Terme(0).type :=entier | Empiler(entier) |
| Terme -> id | Terme(0).type :=Rechercher(id.entree ́) |  |
| Terme -> booleen | Terme(0).type :=booleen | Empiler(booleen) |
| Terme -> reel | Terme(0).type :=reel | Empiler(reel) |
| Terme -> Chaine | Terme(0).type :=chaine | Empiler(chaine) |
| D -> id : type pv D | AjouterType(id.entree ́ , Type(0).type) |  |
| D -> I | D.type = I.type |  |
| D’ -> D | D’.type=D.type |  |
| I -> if ( E ) then { I } I | I(0).type:= si E(0).type=booléen alors si I(1).type =vide et I(2).type alors vide  sinon erreur-type |  |
| I -> if ( E ) then { I } I else { I } I | I(0).type:= si E(0).type = booléen alors si I(1).type sinon si I(2).type= vide et I(3).type=vide alors vide sinon erreur-type. |  |
| I -> While ( E ) Do { I } I | I(0).type=si E(0).type= booleen alors si I(1).type = vide et I(2).type =vide alors vide sinon erreur de type |  |
| I-> id opaff terme pv I | I(0).type := si id.type=terme.type=I(1).type sinon erreur-type |  |
| I-> write E pv I | I(0):= si E.type=I(1).Type sinon erreur-type |  |
| I-> read E pv I | I(0) := si E.type = I(1).type sinon erreur-type |  |
| I-> '' |  |  |
| E-> ( E ) K | E(0).type := si (E(1).type=K.type)  Sinon erreur-type |  |
| E-> { E } K | E(0).type := si (E(1).type=K.type)  Sinon erreur-type |  |
| E-> non ( E ) K | E(0).type := si (E(1).type=K.type)  Sinon erreur-type |  |
| E-> terme K | E(0).type := si (E(1).type=K.type)  Sinon erreur-type |  |
| E-> '' | E(0).type := si (E(1).type=K.type)  Sinon erreur-type |  |
| K-> oprel E | K.type=E.Type |  |
| K-> opar E | K.type=E.Type |  |
| K-> opLog E | K.Type=E.type |  |
| K-> opaff E | K.Type=E.Type |  |
| K-> '' |  |  |

# Génération de code:

|  |  |
| --- | --- |
| D’-> D | D’.place :=D.place  D’.code :=D.code |
| Terme -> entier | Terme.place :=entier.val |
| Terme -> id | Terme.ple :=id.place ;  E.code :=’’ |
| Terme -> booleen | Terme.place :=booleen.val |
| Terme -> reel | Terme.place :=reel.val |
| Terme -> Chaine | Terme.place :=chaine.val |
| D -> id : type pv D | D.place :=nouvtemp() ||  D(0).code :=D(1).code||gen(D.place :=id.place’ :’type.place ‘pv’D(1).code) |
| D -> I | D.place :=I.place  D.code :=I.code |
| D’ -> D | D’.place :=D.place ;  D.code :=D.code |
| I -> if ( E ) then { I } I | I.lSortie:=NouvEtiq()  I(0).code:=E.code|| gen(‘If’’(‘ E.place’=’’0’ ‘)’’then’ goto I.Lsortie)||I(1).code||I(2).code |
| I -> if ( E ) then { I } I else { I } I | I.LSinon:=Nouvetiq()  I.Lsortie:=Nouvetiq()  I(0).code:=E.code||gen(‘If’ ‘(‘ E.place’=’’0’ ‘)’ ‘then’ goto I.LSinon)||gen(I.LSinon”:”)I(2).code|| I(3).code |
| I -> While ( E ) Do { I } I | I.Ldebut:=NouvEtiq()  I.code:=gen(I.LDebut:’)||I(1).code||E.code||gen(“if’E.place ‘<>0’ aller à I.Ldébut) |
| I-> id opaff terme pv I | I(0).code:=terme.code||gen(id.place’ :=’terme.place)||I(1).code |
| I-> write E pv I | I(0).code:=E.code||I(1).code  I(0).place=E.place |
| I-> read E pv I | I(0).code :=E.code  I(0).place :=E.place||I(1).code |
| I-> '' |  |
| E-> ( E ) K | E(0) :=E(1).code || K.code;E(0).place :=E(1).place ; |
| E-> { E } K | E(0) :=E(1).code || K.code;E(0).place :=E(1).place ; |
| E-> non ( E ) K | E(0) :=E(1).code || K.code;E(0).place :=E(1).place ; |
| E-> terme K | E(0) :=E(1).code || K.code;E(0).place :=E(1).place ; |
| E-> '' |  |
| K-> oprel E | K.code. :=E.code  K.place :=E.place |
| K-> opar E | K.code :=E.code  K.place :=E.place |
| K-> opLog E | K.code :=E.code  K.place :=E.place |
| K-> opaff E | K.code :=E.code |
| K-> '' |  |