LANGAGE C-PASCAL

Grammaire Prédictive LL1

Réalisé par :

* Abchir Ouijdane
* Amanzou Anas
* Bahbah Ibrahim
* Lahnech Hamza

Encadré par :

* Pr ELGHAZI Souhail

1. Introduction

Dans cette partie on rendra la grammaire du language C- Pascal LL( 1 ) en élevant toute éventuelle récursivité à gauche, non factorisation ou ambiguïté, dans le but de la rendre analysable par la méthode descendante récursive. Pour ce, on vérifiera ces conditions pour chaque production du grammaire, en évoquant toutes les justifications possibles .

1. La grammaire du langage C--

La première règle est **OK**, vu que c’est un seul choix.

La deuxième règle : **<liste-declarations> :<liste-declarations> <declaration> | epsilon** est récursive à gauche, après élimination de la récursivité, on a deux nouvelle règles :

* **<liste-declarations> : <liste-declarations’>**qui est **OK**
* **<liste-declarations’> : <declaration><liste-declarations’>  | epsilon** qui n’est pas de tout LL(1), vu que {**entie**r, **sturcture**} appartient à **Premier**(**<liste-declarations>’ ) et à Suivant**(**<liste-declarations>’ )**

La solution qu’on a abordé est de reformuler l’ensemble des première règles afin de former une grammaire déterministe.

* Grammaire finale

R1) <Programme> : : = <liste- declarateurs> < liste-fonctions’> **main** ( ) <bloc>

R2) <liste-declarateurs> : : = <declarateur> <liste-declarateurs’> ;

R3) <liste-declarateurs’> : : = , <declarateur> <liste-declarateurs’> | **epsilon**

R4) <liste-declarateurs-sp> : : = <declarateur-sp> <liste-declarateurs-sp’> ;

R5) <liste-declarateurs-sp’> : : = , <declarateur-sp> <liste-declarateurs-sp’> | **epsilon**

R6) <declarateur-sp> : : = **entier** <declarateur-sp’’>

R7) < declarateur-sp’’> : : = <variable><liste-declarateurs-sp’>

R8) < liste-fonctions>: : =< liste-fonctions’>

R8-1) < liste-fonctions’>: := <fonction> < liste-fonctions’> | **epsilon**

R9) <fonction>: : = <identificateur> ( <liste-parms> ) < liste-declarateurs’><bloc>

R10) <liste-parms> ::= <parm><liste-parms’>

R10-1) <liste-parms’> ::= , <parm><liste-parms’> | **epsilon**

R11) <parm>: : = **entier(** <variable> | <variable> [<nombre>]**)**

R12) <bloc>: : = { <liste-instructions> }

R13) <liste-instructions> : : = <liste-instructions’>

R13-1) <liste-instructions’> : : = < instruction > <liste-instructions’> | **epsilon**

R14) <instruction> : : = < iteration> I <selection> | <saut> |<affectation><bloc> I <appel>| **epsilon** ;

R15) <iteration>: : = **tanque** <condition> **faire** <bloc>

R16)<selection>: : = **si** <condition> **alors** <bloc>**( epsilon** | **sinon**<bloc>**)**

R17) <saut>: : = **retour** <expression>

R18)<affectation> : : = <variable> = <expression> | <variablestruct>.<variable>= <expression> | <variable> = **lire** ( )

R19)<appel> : : = <identificateur> ( <liste-expressions>) | **ecrire** (<expression>)

R20) <liste-expressions> : : = <expression> <liste-expressions’>

R20-1) <liste-expressions’> : : =,<expression> <liste-expressions’> | **epsilon**

R21) <condition> : : = !(<expression>) | <expression> **(**<binary-rel><expression> |

<binary-comp><expression>**)**

R22) <expression> : : = <terme>**(epsilon** | <binary-op><terme >**)**

R23) <terme> : : = <variable> | <nombre> | <expression>

R24) <variable> : : = <lettre><variable’>

R24-1) <variable’> ::= <variable’> | **epsilon**

R25) <variablestruct> : : = <lettre><variablestruct’>

R25-1) <variablestruct’> ::= ><variablestruct’>|**epsilon**

R26) <nombre> : : = <constante><nombre’>

R26-1) <nombre’> ::= <constante><nombre’> | **epsilon**

R27) <identificateur> : : = <lettre> <identificateur’>

R27-1)<identificateur’> ::= <lettre><identificateur’> | **epsilon**

R28)<lettre: : = A|B|C|...|Z|a|b|c|...­|z

R29)<constante>: : = 0|1|…­|9

R30)<binary-op>: : = +|-|\*|/

R31)<binary-rel>: : = **&** | |

R32)<binary-comp>: : = <|>|>=|<=|==|!=

* Justifications/Explications

Règle 1 :

<Programme> : : = < liste-declarateurs>< liste-fonctions’>**main** ( ) <bloc>

La règle 1 est bonne car il ne présente qu’un seul choix.

Règle 2 :

<liste-declarateurs> : : = <declarateur> <liste-declarateurs’> ;

La règle 2 est bonne car il ne présente qu’un seul choix.

Règle 3 :

<liste-declarateurs’> : : = , <declarateur> <liste-declarateurs’> | **epsilon**

On a l’intersection des premiers des deux sous-règles de <liste-declarateurs’> est l’ensemble vide ; et comme une seule est non annulable, on a :

Premier(<liste-declarateurs’>) = {‘,’, **epsilon**} et

suivant(<liste-declarateurs’>) = { **‘ ;’**} d’où l’intersection est l’ensemble vide.

Alors la règle 3 est bonne.

Règle 4 :

<Programme> : : = <liste-declarateurs-sp> : <declarateur-sp> <liste-declarateurs-sp’ > ;

La règle 4 est bonne car il ne présente qu’un seul choix.

Règle 5 :

<liste-declarateurs-sp’> : : = **, <**declarateur-sp> <liste-declarateurs-sp’> | **epsilon**

On a l’intersection des premiers des deux sous-règles de <liste-declarateurs’> est l’ensemble vide ; et comme une seule est non annulable, on a :

Premier(<liste-declarateurs-sp’>) = {‘,’, **epsilon**} et

suivant(<liste-declarateurs-sp’>) = { ;} d’où l’intersection est l’ensemble vide.

Alors la règle 5 est bonne.

Règle 6 :

<declarateur-sp> : : = **entier** <declarateur-sp’’>

La règle 6 est bonne car il ne présente qu’un seul choix.

Règle 7 :

< declarateur-sp’’> : : = <variable>< liste-declarateurs-sp’>

La règle 7 est bonne car il ne présente qu’un seul choix.

Règle 8 :

< liste-fonctions>: : = < liste-fonctions> <fonction> | **epsilon**

**Eliminier la récursivité à gauche :**

R8) < liste-fonctions>: : =< liste-fonctions’>

R8-1) < liste-fonctions’>: := <fonction> < liste-fonctions’> | **epsilon**

Premier(<liste-fonctions’>) = {A,B,C…..,Z,a,b,c…..,z, **epsilon**} et

Suivant(<liste-fonctions’>) = { main(), ;}

L’intersection est vide .

Règle 9 :

<fonction>: : = <identificateur> ( <liste-parms> ) < liste-declarations><bloc>

La règle 9 est bonne car il ne présente qu’un seul choix.

Règle 10 :

<liste-parms>: : = <liste-parms> , <parm> | <parm>

**Eliminier la récursivité à gauche :**

<liste-parms> ::= <parm><liste-parms’>

<liste-parms’> ::= , <parm><liste-parms’> | **epsilon**

Premier(<liste-parms’>) = {‘,’,epsilon } et

Suivant(<liste-parms’>) = { ;} d’où l’intersection est vide.

Règle 11 :

<parm>: : = **entier** <variable> | **entier** <variable> [<nombre>]

**Factorisation à gauche :**

<parm>: : = **entier**<variable> **(epsilon** | [<nombre>]**)**

Premier(<parm>) = {entier } et

Suivant(<parm>) = {‘,’ , ;} . L’ensemble est vide.

Règle 12 :

<bloc>: : = { <liste-instructions> }

La règle 12 est bonne car il ne présente qu’un seul choix.

Règle 13 :

<liste-instructions> : : = <liste-instructions>< instruction > | **epsilon**

**Eliminier la récursivité à gauche :**

R13) <liste-instructions> : : = <liste-instructions’>

R13-1) <liste-instructions’> : : = < instruction > <liste-instructions’> | **epsilon**

Premier (<liste-instructions’>) = {tantque , si , retour,ecrire,A,B,C,…,Z,a,b…z,epsilon } et

Suivant(<liste-instructions’>) = {‘}’, ; } . L’ensemble est vide.

Règle 14 :

<instruction>: : = < iteration> I <selection> | <saut> | <affectation><bloc> I <appel>| **epsilon** ;

Premier (<liste-instructions’>) = {tantque , si , retour,ecrire,A,B,C,…,Z,a,b…z,epsilon } et

Suivant(<liste-instructions’>) = {‘}’, ; } . L’ensemble est vide.

Règle 15 :

<iteration>: : = **tanque** <condition> **faire** <bloc>

La règle 1( est bonne car il ne présente qu’un seul choix

Règle 16 :

<selection>: : = **si** <condition> **alors** <bloc> | **si** <condition> **alors** <bloc> <bloc>

**Factorisation à gauche :**

R16)<selection>: : = **si** <condition> **alors** <bloc>**( epsilon** | **sinon**<bloc>**)**

**Premier(**<selection>) = {si} et

Suivant (<selection>)={‘}’, ;} L’intersection est vide.

Règle 17 :

R17) <saut>: : = **retour** <expression>

La règle 17 est bonne car il ne présente qu’un seul choix

Règle 18 :

<affectation> : = <variable> = <expression> | <variablestruct>. <variable>= <expression> | <variable> = **lire** ( )

**Premier(**<affectation> ) = {A,B,…Z,a,b,…,z} et

Suivant (<affectation>)={‘=’, ‘{’}

Règle 19 :

<appel>: : = <identificateur> ( <liste-expressions>) | **ecrire** (<expression>)

**Premier(**<appel> ) = {A,B,…Z,a,b,…,z} et

Suivant (<appels>)={‘}’, ; }

Règle 20 :

<liste-expressions> : : = <expression> <liste-expressions’>

R20-1) <liste-expressions’> : : =,<expression> <liste-expressions’> | **epsilon**

**Premier(**<liste-expressions’ > ) = {‘,’,epsilon} et

Suivant (<liste-expressions’>)={‘}’, ; }

Règle 21 :

<condition> : : = !(<expression>) | <expression> **(**<binary-rel><expression> |

<binary-comp><expression>**)**

**Premier(**<condition > ) = { !, a,b,…,z,A,B,…Z,0,1….,9}et

Suivant (<condition>)={ faire,alors}

Règle 22 :

<expression> : : = <terme>**(epsilon** | <binary-op><terme >**)**

**Premier(**<expression > ) = {  a,b,…,z,A,B,…Z,0,1….,9}et

Suivant (<expression>) = { ; } U Suivant (<affectation>) = {‘=’, ‘{’, ;}

Règle 23 :

<terme> : : = <variable> | <nombre> | <expression>

Premier(<terme>) = {a, .. , z, A, .., Z, 0, .., 9,}

Suivant(<terme>) = {+,-,\*,/} U suivant(<expression>) = {+,-,\*,/,‘=’, ‘{’, ;}

Intersection vide.

Règle 24 :

<variable> : : = <variable’>

<variable’> ::= <lettre> <variable’> | **epsilon**

Premier(<variable’>) = {a, .. , z, A, .., Z}

Suivant(<variable’>)= {+,-,\*,/,‘=’, ‘{’, ;, [, ‘,’ , ;}

Règle 25 :

<variablestruct> : : = <variablestruct’>

<variablestruct’> ::= <lettre><variablestruct’>|**epsilon**

Premier(<variablestruct’>) = {a, .. , z, A, .., Z}

Suivant(<variablestruct’>) = {.}

Intersection vide.

Règle 26 :

<nombre> ::= <constante><nombre’>

<nombre’> ::= <constante><nombre’> | **epsilon**

Premier(<nombre>) = {0, .., 9}

Suivant(<nombre>) = {+,-,\*,/,‘=’, ‘{’, ;, ]}

Intersection vide.

Règle 27 :

<identificateur> : : = <lettre> <identificateur’>

<identificateur’> ::= <lettre><identificateur’> | **epsilon**

Premier(<identificateur>) = {a, .., z, A, .., Z}

Suivant(<identificateur>) = { **ecrire,** ‘,’, ‘}’, ;}

Intersection vide.

Règle 28 :

<lettre>: : = A|B|C|...|Z|a|b|c|...­|z

La règle 28 est bonne car il ne présente qu’un seul choix

Règle 29 :

<constante>: : = 0|1|…­|9

La règle 29 est bonne car il ne présente qu’un seul choix

Règle 30 :

<binary-op>: : = +|-|\*|/

La règle 30 est bonne car il ne présente qu’un seul choix

Règle 31 :

<binary-rel>: : = **&** | |

La règle 31 est bonne car il ne présente qu’un seul choix

Règle 32 :

<binary-comp>: : = <|>|>=|<=|==|!=

La règle 32 est bonne car il ne présente qu’un seul choix

**Tableau des premiers et des suivants**