



UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F-403
INTRODUCTION TO LANGUAGE THEORY AND COMPILING

Assignement 2 : The parser

Maxime DESCLEFS - 362626

Julian SCHEMBRI - 380446

2015 - 2016

1 Introduction

Dans le cadre du cours d'introduction à la théorie du langage et de compilation (INFO-F-403), il a été demandé d'implémenter et d'écrire un compilateur pour le langage SUPRALGOL²⁰¹⁶, une variante du langage Algol68.

La première partie consistait en la création du scanner. Ce scanner analyse et découpe, ensuite, un code SUPRALGOL²⁰¹⁶ permettant d'identifier chacun des éléments de ce dernier, et leur correspondance dans ce langage. La deuxième partie consistait en la création du parser. Ce parser permet de détecter les erreurs de syntaxe ne respectant pas la grammaire du langage à l'aide de tous les outils mis à sa disposition. Les différents outils seront énoncés dans le paragraphe suivant.

Il fallait, dans un premier temps, transformer la grammaire SUPRALGOL²⁰¹⁶ afin de la rendre LL(1). Quatre étapes sont nécessaires à cette transformation : la suppression des symboles inutiles, la suppression de la récursivité à gauche, la factorisation des règles, ainsi que la prise en compte de la priorité et l'associativité des opérateurs pour rendre la grammaire non-ambigüe. Sur base de cette nouvelle grammaire et quelques manipulations, une *action table* a pu être créée, rendant possible l'exécution du parser LL(1) qu'il a été demandé de réaliser.

Etant donné la demande de justificatifs des différentes étapes, l'attention a été portée sur l'implémentation des différents algorithmes en vue de l'obtention des réponses les plus adéquates. Dans la suite de ce rapport, seront expliqué les différentes opérations, ainsi que les résultats ayant pu être obtenus grâce à ces dernières.

2 Grammaire

Une grammaire permet de définir un langage en définissant sa syntaxe. Elle décrit donc l'ensemble des mots admissibles, que nous appellerons *terminaux*, à ce langage sur un alphabet donné, que nous appellerons *variables*.

2.1 Suppression des variables inutiles

Une variable est inutile si elle ne remplit pas l'une des fonction suivante : être accessible depuis le symbole de départ et produire quelque chose. Dès lors les variables inutiles détectées, elles peuvent être supprimées des différentes listes ainsi que des différentes règles dans lesquelles elles se trouvent.

2.2 Priorité et associativité des opérateurs

Afin de rendre la grammaire non-ambigüe, il est nécessaire d'appliquer les priorités et les associativités possible sur les différents opérateurs de cette dernière.

Dans les règles de la grammaire, les plus importantes, ç-à-d avec une priorité plus élevée, se trouvent les plus haut dans la grammaire. Ceci peut être expliqué par le parcours des règles (parcours de liste type *ArrayList*) ; en effet les premières règles, ç-à-d tout en haut de la liste, sont exécutées en premier. Il ne reste donc qu'à intervertir, remplacer, décaler, des règles en les montant pour en augmenter la priorité ou bien les descendre dans le cas échéant. Si des opérateurs ont la même priorité, ils sont mis sur une même ligne.

2.3 Suppression de la récursivité à gauche

La récursivité à gauche est un problème pour le «*Top-Down parsing*». En effet, cette classe de parseur utilise, comme demandé dans l'énoncé, la «*Left-most derivation*». Pour pallier à ce problème, la récursivité à gauche va être remplacée par une autre sorte de récursivité. L'introduction d'une nouvelle variable est nécessaire.

2.4 Factorisation à gauche

Pour les «*Top-Down parsing*», chacune des règles de la grammaire doit être factorisées à gauche. Comme pour le point précédent sur la récursivité, l'introduction d'une nouvelle variable est nécessaire. Cette étape consiste à rassembler les différents terminaux, ayant un préfixe commun (différent de ϵ), d'une variable, dans cette nouvelle variable récemment introduite. Cela enlève une ambiguïté et permet au parser de savoir directement quelle règle choisir sans devoir toutes les parcourir.

3 Table d'action

Comme mentionné dans l'introduction, la construction d'une table d'action est nécessaire au bon fonctionnement du parser. Pour chaque symbole, le parseur doit savoir deux choses sur ce dernier : *First(X)* et *Follow(X)* où X est le symbole. La grammaire étant de type LL(1), les différents algorithmes faisant présence d'un ordre de grandeur k se le voient attribué à 1. *First* est donc l'ensemble des premiers terminaux (vecteur d'un seul terminal) rencontré pouvant commencer un mot généré à partir d'une variable. *Follow* s'applique pour les terminaux qui suivent le mot. Sur base de ces deux vecteurs, la table d'action peut être construite. L'action table permet donc au parseur de faire la liaison, entre les variables et les terminaux, en indiquant le numéro de la règle qui permet cette dernière.

3.1 Valeur de First et Follow

Les différentes étapes permettant d'obtenir ces résultats sont proposées en annexe. Ces étapes ont été réalisées sur base des algorithmes présents dans les slides du cours.

	First	Follow
Program	<i>begin</i>	<i>eps</i>
Code	[<i>VarName</i>], <i>print</i> , <i>read</i> , <i>for</i> , <i>eps</i> , <i>while</i> , <i>if</i>	<i>fi</i> , <i>od</i> , <i>else</i> , <i>end</i>
InstList	[<i>VarName</i>], <i>print</i> , <i>read</i> , <i>for</i> , <i>while</i> , <i>if</i>	<i>fi</i> , <i>od</i> , <i>else</i> , <i>end</i>
Instruction	[<i>VarName</i>], <i>print</i> , <i>read</i> , <i>for</i> , <i>while</i> , <i>if</i>	<i>fi</i> , <i>od</i> , <i>else</i> , <i>end</i> , ;
InstList'	<i>eps</i> , ;	<i>fi</i> , <i>od</i> , <i>else</i> , <i>end</i>
Assign	[<i>VarName</i>]	<i>fi</i> , <i>od</i> , <i>else</i> , <i>end</i> , ;

If	<i>if</i>	<i>fi, od, else, end, ;</i>
While	<i>while</i>	<i>fi, od, else, end, ;</i>
For	<i>for</i>	<i>fi, od, else, end, ;</i>
Print	<i>print</i>	<i>fi, od, else, end, ;</i>
Read	<i>read</i>	<i>fi, od, else, end, ;</i>
ExprArith	<i>[VarName], (, −, [Number]</i>	<i><=, fi, for, do, while, od, else, by, end, if, [Number], read, (,), *, +, −, /=, /, [VarName], print, :, to, <, =, >, >=</i>
ExprArith0	<i>[VarName], (, [Number]</i>	<i><=, fi, for, do, while, od, else, by, end, if, [Number], read, (,), *, +, −, /=, /, [VarName], print, :, to, <, =, >, >=</i>
ExprArith0'	<i>eps, *, /</i>	<i><=, fi, for, do, while, od, else, by, end, if, [Number], read, (,), *, +, −, /=, /, [VarName], print, :, to, <, =, >, >=</i>
Op'	<i>*, /</i>	<i><=, fi, for, do, while, od, else, by, end, if, [Number], read, (,), *, +, −, /=, /, [VarName], print, :, to, <, =, >, >=</i>
ExprArith1	<i>[VarName], (, [Number]</i>	<i><=, fi, for, do, while, od, else, by, end, if, [Number], read, (,), *, +, −, /=, /, [VarName], print, :, to, <, =, >, >=</i>
ExprArith1'	<i>eps, +, −</i>	<i><=, fi, for, do, while, od, else, by, end, if, [Number], read, (,), *, +, −, /=, /, [VarName], print, :, to, <, =, >, >=</i>

Op''	$+$, $-$	\leq , <i>fi</i> , <i>for</i> , <i>do</i> , <i>while</i> , <i>od</i> , <i>else</i> , <i>by</i> , <i>end</i> , <i>if</i> , [<i>Number</i>], <i>read</i> , ($,$), $*$, $+$, $-$, $/$, $=$, [<i>VarName</i>], <i>print</i> , $;$, <i>to</i> , $<$, $=$, $>$, \geq
ExprArith2	[<i>VarName</i>], ($,$), [<i>Number</i>]	\leq , <i>fi</i> , <i>for</i> , <i>do</i> , <i>while</i> , <i>od</i> , <i>else</i> , <i>by</i> , <i>end</i> , <i>if</i> , [<i>Number</i>], <i>read</i> , ($,$), $*$, $+$, $-$, $/$, $=$, [<i>VarName</i>], <i>print</i> , $;$, <i>to</i> , $<$, $=$, $>$, \geq
Cond	[<i>VarName</i>], <i>not</i> , ($,$), $-$, [<i>Number</i>]	<i>read</i> , <i>fi</i> , <i>or</i> , <i>for</i> , <i>then</i> , <i>do</i> , <i>while</i> , [<i>VarName</i>], <i>print</i> , <i>od</i> , <i>else</i> , <i>end</i> , $;$, <i>if</i>
If'	<i>fi</i> , <i>else</i>	<i>fi</i> , <i>od</i> , <i>else</i> , <i>end</i> , $;$
Cond0	[<i>VarName</i>], ($,$), $-$, [<i>Number</i>]	<i>read</i> , <i>fi</i> , <i>or</i> , <i>for</i> , <i>then</i> , <i>do</i> , <i>while</i> , [<i>VarName</i>], <i>print</i> , <i>od</i> , <i>else</i> , <i>end</i> , $;$, <i>if</i>
SimpleCond	[<i>VarName</i>], ($,$), $-$, [<i>Number</i>]	<i>read</i> , <i>fi</i> , <i>or</i> , <i>for</i> , <i>then</i> , <i>do</i> , <i>while</i> , [<i>VarName</i>], <i>print</i> , <i>od</i> , <i>else</i> , <i>end</i> , $;$, <i>if</i>
Cond1	-	<i>read</i> , <i>fi</i> , <i>or</i> , <i>for</i> , <i>then</i> , <i>do</i> , <i>while</i> , [<i>VarName</i>], <i>print</i> , <i>od</i> , <i>else</i> , <i>end</i> , $;$, <i>if</i>
Cond1'	<i>eps</i>	<i>read</i> , <i>fi</i> , <i>or</i> , <i>for</i> , <i>then</i> , <i>do</i> , <i>while</i> , [<i>VarName</i>], <i>print</i> , <i>od</i> , <i>else</i> , <i>end</i> , $;$, <i>if</i>
BinOp'	<i>and</i>	<i>read</i> , <i>fi</i> , <i>or</i> , <i>for</i> , <i>then</i> , <i>do</i> , <i>while</i> , [<i>VarName</i>], <i>print</i> , <i>od</i> , <i>else</i> , <i>end</i> , $;$, <i>if</i>
Cond2	-	<i>read</i> , <i>fi</i> , <i>or</i> , <i>for</i> , <i>then</i> , <i>do</i> , <i>while</i> , [<i>VarName</i>], <i>print</i> , <i>od</i> , <i>else</i> , <i>end</i> , $;$, <i>if</i>
Cond2'	<i>or</i> , <i>eps</i>	<i>eps</i>

BinOp”	<i>or</i>	<i>[VarName], not, or, (, −, [Number]</i>
Comp	<i><=, <, =, >, / =, >=</i>	<i>[VarName], (, −, [Number]</i>

3.2 Résultats

[illegible]

4 Conclusion

L'implémentation des différents algorithmes n'a pas été une tâche facile. Certaines opérations et vérifications ne sont pas, toujours, entièrement décrites dans les slides. Cette implémentation n'est donc sûrement pas parfaite. Plusieurs tests ont été effectués et ont pointé de nombreuses erreurs, qui ont pu être corrigées par la suite. Cependant, pour les recommandations du projet, les résultats ont l'air dans l'ensemble corrects.

Lors de la précédente itération, le rapport avait été quelque'un peu négligé et cela s'était senti. C'est donc pour cela que celui-ci a été mieux préparé, et réalisé de la manière la plus claire possible.

5 Annexe

1. Algorithmes de transformation de la grammaire

(a.1) Suppression des variables non productives

1) Au départ $V_0 = \{ \}$

2)

$\langle ExprArith \rangle \rightarrow [VarName]$

$\langle Op \rangle \rightarrow +$

$\langle BinOp \rangle \rightarrow \text{and}$

$\langle Comp \rangle \rightarrow =$

$\langle Print \rangle \rightarrow \text{print}([VarName])$

$\langle Read \rangle \rightarrow \text{read}([VarName])$

$V_1 \leftarrow \{ \langle ExprArith \rangle, \langle Op \rangle, \langle BinOp \rangle, \langle Comp \rangle, \langle Print \rangle, \langle Read \rangle \} \cup V_0$

3)

$\langle Instruction \rangle \rightarrow \langle Print \rangle$

$\langle Assign \rangle \rightarrow [VarName] := \langle ExprArith \rangle$

$\langle SimpleCond \rangle \rightarrow \langle ExprArith \rangle \langle Comp \rangle \langle ExprArith \rangle$

$V_2 \leftarrow \{ \langle Instruction \rangle, \langle Assign \rangle, \langle SimpleCond \rangle \} \cup V_1$

4)

$\langle InstList \rangle \rightarrow \langle Instruction \rangle$

$\langle Cond \rangle \rightarrow \langle SimpleCond \rangle$

$V_3 \leftarrow \{ \langle InstList \rangle, \langle Cond \rangle \} \cup V_2$

5)

$\langle Code \rangle \rightarrow \langle InstList \rangle$

$V_4 \leftarrow \{ \langle Code \rangle \} \cup V_3$

6)

$\langle Program \rangle \rightarrow \text{begin} \langle Code \rangle \text{end}$

$\langle If \rangle \rightarrow \text{if} \langle Cond \rangle \text{ then} \langle Code \rangle \text{ fi}$

$\langle While \rangle \rightarrow \text{while} \langle Cond \rangle \text{ do} \langle Code \rangle \text{ od}$

$\langle For \rangle \rightarrow \text{for} [VarName] \text{ from} \langle ExprArith \rangle \text{ by} \langle ExprArith \rangle \text{ to} \langle ExprArith \rangle \text{ do} \langle Code \rangle \text{ od}$

$V_5 \leftarrow \{ \langle Program \rangle, \langle If \rangle, \langle While \rangle, \langle For \rangle \} \cup V_4$

7)

$V_6 \leftarrow \{ \langle Program \rangle, \langle If \rangle, \langle While \rangle, \langle For \rangle, \langle Code \rangle, \langle InstList \rangle, \langle Cond \rangle, \langle Instruction \rangle, \langle Assign \rangle,$

$\langle SimpleCond \rangle, \langle ExprArith \rangle, \langle Op \rangle, \langle BinOp \rangle, \langle Comp \rangle, \langle Print \rangle, \langle Read \rangle \}$

En ayant $V_5 = V_6$, La boucle s'arrête.

8) $V' \leftarrow V_6$

$P' =$ L'ensemble des règles de P qui ne contiennent pas des variables dans $V \setminus V'$

Ici, $V' = V$ il n'y a plus rien à enlever, $P' = P$ and $G' = G$

(a.2) Suppression des variables inaccessible

1) Au départ $V_0 \leftarrow \{ \langle Program \rangle \}$

2) $\langle Program \rangle \rightarrow \text{begin} \langle Code \rangle \text{end}$, $V_1 \leftarrow \{ \text{begin}, \langle Code \rangle, \text{end} \} \cup V_0$

3) $\langle Code \rangle \rightarrow \langle InstList \rangle$, $V_2 \leftarrow \{ \langle InstList \rangle \} \cup V_1$

4) $\langle InstList \rangle \rightarrow \langle Instruction \rangle$; $\langle InstList \rangle$, $V_3 \leftarrow \{ \langle Instruction \rangle, ; \} \cup V_2$

5) $\langle Instruction \rangle \rightarrow \langle Assign \rangle | \langle If \rangle | \langle While \rangle | \langle For \rangle | \langle Print \rangle | \langle Read \rangle$,

$V_4 \leftarrow \{ \langle Assign \rangle, \langle If \rangle, \langle While \rangle, \langle For \rangle, \langle Print \rangle, \langle Read \rangle \} \cup V_3$

- 6) $\langle Assign \rangle \rightarrow [VarName] := \langle ExprArith \rangle, V_5 \leftarrow \{ [VarName], :=, \langle ExprArith \rangle \} \cup V_4$
7) $\langle ExprArith \rangle \rightarrow [Number] | (\langle ExprArith \rangle) | - \langle ExprArith \rangle | \langle ExprArith \rangle \langle Op \rangle \langle ExprArith \rangle, V_6 \leftarrow \{ [Number], (,), -, \langle Op \rangle \} \cup V_5$
8) $\langle Op \rangle \rightarrow + | - | * | /, V_7 \leftarrow \{ +, -, *, / \} \cup V_6$
9) $\langle If \rangle \rightarrow if \langle Cond \rangle then \langle Code \rangle else \langle Code \rangle fi, V_7 \leftarrow \{ if, \langle Cond \rangle, then, else, fi \} \cup V_6$
10) $\langle Cond \rangle \rightarrow \langle Cond \rangle \langle BinOp \rangle \langle Cond \rangle | not \langle SimpleCond \rangle, V_8 \leftarrow \{ \langle BinOp \rangle, not, \langle SimpleCond \rangle \} \cup V_7$
11) $\langle SimpleCond \rangle \rightarrow \langle ExprArith \rangle \langle Comp \rangle \langle ExprArith \rangle, V_9 \leftarrow \{ \langle Comp \rangle \} \cup V_8$
12) $\langle BinOp \rangle \rightarrow and | or, V_{10} \leftarrow \{ and, or \} \cup V_9$
13) $\langle Comp \rangle \rightarrow = | > | > | < | < | / =, V_{11} \leftarrow \{ =, > =, >, < =, <, / = \} \cup V_{10}$
14) $\langle While \rangle \rightarrow while \langle Cond \rangle do \langle Code \rangle od, V_{12} \leftarrow \{ while, do, od \} \cup V_{11}$
15) $\langle For \rangle \rightarrow for [VarName] from \langle ExprArith \rangle by \langle ExprArith \rangle to \langle ExprArith \rangle do \langle Code \rangle od, V_{13} \leftarrow \{ for, from, by, to \} \cup V_{12}$
16) $\langle Print \rangle \rightarrow print ([VarName]), V_{14} \leftarrow \{ print \} \cup V_{13}$
17) $\langle Read \rangle \rightarrow read ([VarName]), V_{15} \leftarrow \{ read \} \cup V_{14}$
18) $V_{16} \leftarrow \{ \langle Program \rangle, begin, \langle Code \rangle, end, \langle InstList \rangle, \langle Instruction \rangle, ;, \langle Assign \rangle, \langle If \rangle, \langle While \rangle, \langle For \rangle, \langle Print \rangle, \langle Read \rangle, [VarName], :=, \langle ExprArith \rangle, [Number], (,), -, \langle Op \rangle, +, -, *, /, if, \langle Cond \rangle, then, else, fi, \langle BinOp \rangle, not, \langle SimpleCond \rangle, \langle Comp \rangle, and, or, =, > =, >, < =, <, / =, while, do, od, for, from, by, to, print, read \}$
Having $V_{16} = V_{15}$, the loop stops.
19) $V' \leftarrow V_{16} \cap V = V; T' \leftarrow V_{16} \cap T = T$
 $P' \leftarrow$ L'ensemble des règles de P qui contiennent seulement des variables de V_{16}

Ici $V_{16} = V$ vu que $P' = P$ on a $G' = G$.

(b.1) Suppression de la récursivité à gauche

Dans cette grammaire, il y a quatre cas de récursivité à gauche :

$\langle ExprArith \rangle \rightarrow \langle ExprArith \rangle \langle Op' \rangle \langle ExprArith \rangle | \langle ExprArith 0 \rangle$
 $\langle ExprArith 0 \rangle \rightarrow \langle ExprArith 0 \rangle \langle Op'' \rangle \langle ExprArith \rangle | \langle ExprArith 1 \rangle$
 $\langle Cond \rangle \rightarrow \langle Cond \rangle \langle BinOp' \rangle \langle Cond \rangle | \langle Cond 0 \rangle$
 $\langle Cond 0 \rangle \rightarrow \langle Cond 0 \rangle \langle BinOp'' \rangle \langle Cond \rangle | \langle Cond 1 \rangle$

Pour le première cas, introduisons une nouvelle variable $\langle ExprArith' \rangle$ et supprimons la récursivité:

$\langle ExprArith' \rangle \rightarrow \langle Op' \rangle \langle ExprArith \rangle \langle ExprArith' \rangle | \epsilon$
 $\langle ExprArith \rangle \rightarrow \langle ExprArith 0 \rangle \langle ExprArith' \rangle$

Pour le deuxième cas, introduisons une nouvelle variable $\langle ExprArith 0' \rangle$ et supprimons la récursivité :

$\langle ExprArith 0' \rangle \rightarrow \langle Op'' \rangle \langle ExprArith \rangle \langle ExprArith 0' \rangle | \epsilon$
 $\langle ExprArith 0 \rangle \rightarrow \langle ExprArith 1 \rangle \langle ExprArith 0' \rangle$

Pour le troisième cas, introduisons une nouvelle variable $\langle Cond' \rangle$ et supprimons la récursivité :

$\langle Cond' \rangle \rightarrow \langle BinOp' \rangle \langle Cond \rangle \langle Cond' \rangle | \epsilon$
 $\langle Cond \rangle \rightarrow \langle Cond 0 \rangle \langle Cond' \rangle$

Pour le quatrième cas , introduisons une nouvelle variable $\langle Cond0' \rangle$ and remove the left-recursion:

$$\begin{aligned}\langle Cond0' \rangle &\rightarrow \langle BinOp' \rangle \langle Cond \rangle \langle Cond0' \rangle | \epsilon \\ \langle Cond0 \rangle &\rightarrow \langle Cond1 \rangle \langle Cond0' \rangle\end{aligned}$$

(b.2) Factorisation

Dans cette grammaire, il y a seulement un type de règles avec un préfixe commun

$$\begin{aligned}\langle If \rangle &\rightarrow \text{if} \langle Cond \rangle \text{ then} \langle Code \rangle \text{ fi} \\ \langle If \rangle &\rightarrow \text{if} \langle Cond \rangle \text{ then} \langle Code \rangle \text{ else} \langle Code \rangle \text{ fi}\end{aligned}$$

Pour ce premier cas, introduisons une nouvelle variable $\langle If' \rangle$ et factorisons là

$$\begin{aligned}\langle If \rangle &\rightarrow \text{if} \langle Cond \rangle \text{ then} \langle Code \rangle \langle If' \rangle \\ \langle If' \rangle &\rightarrow \text{fi} \\ \langle If' \rangle &\rightarrow \text{else} \langle Code \rangle \text{ fi}\end{aligned}$$

(c) Rendre la grammaire non ambiguë

Pour la règle $\langle ExprArith \rangle$, il faut appliquer les priorités suivante:

$$\begin{aligned}&-x \\ &* / \\ &+ - \\ &[VarName][Number](x)\end{aligned}$$

Réécrivons la règle $\langle Op \rangle$:

$$\begin{aligned}\langle Op' \rangle &\rightarrow + | - \\ \langle Op'' \rangle &\rightarrow * | /\end{aligned}$$

Ensuite réécrivons les règle $\langle ExprArith \rangle$ s:

$$\begin{aligned}\langle ExprArith \rangle &\rightarrow \langle ExprArith \rangle \langle Op' \rangle \langle ExprArith \rangle | \langle ExprArith0 \rangle \\ \langle ExprArith0 \rangle &\rightarrow \langle ExprArith0 \rangle \langle Op'' \rangle \langle ExprArith \rangle | \langle ExprArith1 \rangle \\ \langle ExprArith1 \rangle &\rightarrow - \langle ExprArith \rangle | [VarName] \vee [Number] \vee (\langle ExprArith \rangle)\end{aligned}$$

Pour la règle $\langle Cond \rangle$, il faut appliquer les priorités suivante

$$\begin{aligned}&\text{not } x \\ &> < >= <= \neq \\ &\text{and} \\ &\text{or}\end{aligned}$$

Réécrivons la règle $\langle BinOp \rangle$:

$$\begin{aligned}\langle BinOp' \rangle &\rightarrow \text{or} \\ \langle BinOp'' \rangle &\rightarrow \text{and}\end{aligned}$$

Pour finir, réécrivons les règles $\langle Cond \rangle$:

$$\begin{aligned}\langle Cond \rangle &\rightarrow \langle Cond \rangle \langle BinOp' \rangle \langle Cond \rangle | \langle Cond0 \rangle \\ \langle Cond0 \rangle &\rightarrow \langle Cond0 \rangle \langle BinOp'' \rangle \langle Cond \rangle | \langle Cond1 \rangle \\ \langle Cond1 \rangle &\rightarrow \text{not} \langle SimpleCond \rangle | \langle SimpleCond \rangle\end{aligned}$$

2. Les différentes règles de la grammaire transformée

```
1 : <Program> -> begin <Code> end
2 : <Code> -> eps
3 : <Code> -> <InstList>
4 : <InstList> -> <Instruction> <InstList'>
5 : <InstList'> -> eps
6 : <InstList'> -> ; <InstList>
7 : <Instruction> -> <Assign>
8 : <Instruction> -> <If>
9 : <Instruction> -> <While>
10 : <Instruction> -> <For>
11 : <Instruction> -> <Print>
12 : <Instruction> -> <Read>
13 : <Assign> -> [VarName] := <ExprArith>
14 : <ExprArith> -> - <ExprArith0>
15 : <ExprArith> -> <ExprArith0>
16 : <ExprArith0'> -> <Op'> <ExprArith> <ExprArith0'>
17 : <ExprArith0'> -> eps
18 : <ExprArith0> -> <ExprArith1> <ExprArith0'>
19 : <ExprArith1'> -> <Op''> <ExprArith> <ExprArith1'>
20 : <ExprArith1'> -> eps
21 : <ExprArith1> -> <ExprArith2> <ExprArith1'>
22 : <ExprArith2> -> [VarName]
23 : <ExprArith2> -> [Number]
24 : <ExprArith2> -> ( <ExprArith> )
25 : <Op'> -> *
26 : <Op'> -> /
27 : <Op''> -> +
28 : <Op''> -> -
29 : <If> -> if <Cond> then <Code> <If'>
30 : <If'> -> fi
31 : <If> -> else <Code> fi
32 : <Cond> -> not <Cond0>
33 : <Cond> -> <Cond0>
34 : <Cond0> -> <SimpleCond>
35 : <Cond0> -> <Cond1>
36 : <Cond1'> -> <BinOp'> <Cond2> <Cond1>
37 : <Cond1'> -> eps
38 : <Cond1> -> <Cond2> <Cond1'>
39 : <Cond2'> -> <BinOp''> <Cond> <Cond2'>
40 : <Cond2'> -> eps
41 : <SimpleCond> -> <ExprArith> <Comp> <ExprArith>
42 : <BinOp'> -> and
43 : <BinOp''> -> or
44 : <Comp> -> =
45 : <Comp> -> >=
46 : <Comp> -> >
47 : <Comp> -> <=
48 : <Comp> -> <
49 : <Comp> -> /=
50 : <While> -> while <Cond> do <Code> od
51 : <For> -> for [VarName] from <ExprArith> by <ExprArith> to <ExprArith> do <Code> od
52 : <Print> -> print ( [VarName] )
```

53 : <Read> -> read ([VarName])

3. Calcul des différentes listes First

===== STEP 0 =====

<Program> :
<Code> : eps ,
<InstList> :
<Instruction> :
<InstList'> : eps ,
<Assign> :
<If> :
<While> :
<For> :
<Print> : print ,
<Read> : read ,
<ExprArith> :
<ExprArith0> :
<ExprArith0'> : eps ,
<Op'> : * , / ,
<ExprArith1> :
<ExprArith1'> : eps ,
<Op''> : + , - ,
<ExprArith2> : [VarName] , [Number] ,
<Cond> :
<If'> : fi , else ,
<Cond0> :
<SimpleCond> :
<Cond1> :
<Cond1'> : eps ,
<BinOp'> : and ,
<Cond2> :
<Cond2'> : eps ,
<BinOp''> : or ,
<Comp> : <= , < , = , > , /= , >= ,

===== STEP 1 =====

<Program> : begin ,
<Code> : eps ,
<InstList> :
<Instruction> : print , read ,
<InstList'> : eps ,
<Assign> :
<If> :
<While> :
<For> :
<Print> : print ,
<Read> : read ,
<ExprArith> :
<ExprArith0> :
<ExprArith0'> : eps ,
<Op'> : * , / ,

<ExprArith1> : [VarName] , [Number] ,
 <ExprArith1'> : eps ,
 <Op'> : + , - ,
 <ExprArith2> : [VarName] , [Number] ,
 <Cond> :
 <If'> : fi , else ,
 <Cond0> :
 <SimpleCond> :
 <Cond1> :
 <Cond1'> : eps ,
 <BinOp'> : and ,
 <Cond2> :
 <Cond2'> : eps ,
 <BinOp''> : or ,
 <Comp> : <= , < , = , > , /= , >= ,

===== STEP 2 =====

<Program> : begin ,
 <Code> : eps ,
 <InstList> : print , read ,
 <Instruction> : print , read ,
 <InstList'> : eps , ; ,
 <Assign> :
 <If> :
 <While> :
 <For> :
 <Print> : print ,
 <Read> : read ,
 <ExprArith> :
 <ExprArith0> : [VarName] , [Number] ,
 <ExprArith0'> : eps ,
 <Op'> : * , / ,
 <ExprArith1> : [VarName] , [Number] ,
 <ExprArith1'> : eps ,
 <Op''> : + , - ,
 <ExprArith2> : [VarName] , [Number] ,
 <Cond> :
 <If'> : fi , else ,
 <Cond0> :
 <SimpleCond> :
 <Cond1> :
 <Cond1'> : eps ,
 <BinOp'> : and ,
 <Cond2> :
 <Cond2'> : eps ,
 <BinOp''> : or ,
 <Comp> : <= , < , = , > , /= , >= ,

===== STEP 3 =====

<Program> : begin ,
 <Code> : print , read , eps ,
 <InstList> : print , read ,
 <Instruction> : print , read ,
 <InstList'> : eps , ; ,

```

<Assign> :
<If> :
<While> :
<For> : for ,
<Print> : print ,
<Read> : read ,
<ExprArith> : [VarName] , - , [Number] ,
<ExprArith0> : [VarName] , [Number] ,
<ExprArith0'> : eps , * , / ,
<Op'> : * , / ,
<ExprArith1> : [VarName] , [Number] ,
<ExprArith1'> : eps , + , - ,
<Op''> : + , - ,
<ExprArith2> : [VarName] , ( , [Number] ,
<Cond> :
<If'> : fi , else ,
<Cond0> :
<SimpleCond> : [VarName] , - , [Number] ,
<Cond1> :
<Cond1'> : eps ,
<BinOp'> : and ,
<Cond2> :
<Cond2'> : eps ,
<BinOp''> : or ,
<Comp> : <= , < , = , > , /= , >= ,

```

```

===== STEP 4 =====
<Program> : begin ,
<Code> : print , read , eps ,
<InstList> : print , read ,
<Instruction> : print , read , for ,
<InstList'> : eps , ; ,
<Assign> : [VarName] ,
<If> :
<While> :
<For> : for ,
<Print> : print ,
<Read> : read ,
<ExprArith> : [VarName] , - , [Number] ,
<ExprArith0> : [VarName] , [Number] ,
<ExprArith0'> : eps , * , / ,
<Op'> : * , / ,
<ExprArith1> : [VarName] , ( , [Number] ,
<ExprArith1'> : eps , + , - ,
<Op''> : + , - ,
<ExprArith2> : [VarName] , ( , [Number] ,
<Cond> :
<If'> : fi , else ,
<Cond0> : [VarName] , - , [Number] ,
<SimpleCond> : [VarName] , - , [Number] ,
<Cond1> :
<Cond1'> : eps ,
<BinOp'> : and ,
<Cond2> :
<Cond2'> : eps ,

```

<BinOp''> : or ,
 <Comp> : <= , < , = , > , /= , >= ,

===== STEP 5 =====

<Program> : begin ,
 <Code> : print , read , eps ,
 <InstList> : print , read , for ,
 <Instruction> : [VarName] , print , read , for ,
 <InstList'> : eps , ; ,
 <Assign> : [VarName] ,
 <If> :
 <While> : while ,
 <For> : for ,
 <Print> : print ,
 <Read> : read ,
 <ExprArith> : [VarName] , - , [Number] ,
 <ExprArith0> : [VarName] , (, [Number] ,
 <ExprArith0'> : eps , * , / ,
 <Op'> : * , / ,
 <ExprArith1> : [VarName] , (, [Number] ,
 <ExprArith1'> : eps , + , - ,
 <Op''> : + , - ,
 <ExprArith2> : [VarName] , (, [Number] ,
 <Cond> : [VarName] , not , - , [Number] ,
 <If'> : fi , else ,
 <Cond0> : [VarName] , - , [Number] ,
 <SimpleCond> : [VarName] , - , [Number] ,
 <Cond1> :
 <Cond1'> : eps ,
 <BinOp'> : and ,
 <Cond2> :
 <Cond2'> : or , eps ,
 <BinOp''> : or ,
 <Comp> : <= , < , = , > , /= , >= ,

===== STEP 6 =====

<Program> : begin ,
 <Code> : print , read , for , eps ,
 <InstList> : [VarName] , print , read , for ,
 <Instruction> : [VarName] , print , read , for , while ,
 <InstList'> : eps , ; ,
 <Assign> : [VarName] ,
 <If> : if ,
 <While> : while ,
 <For> : for ,
 <Print> : print ,
 <Read> : read ,
 <ExprArith> : [VarName] , (, - , [Number] ,
 <ExprArith0> : [VarName] , (, [Number] ,
 <ExprArith0'> : eps , * , / ,
 <Op'> : * , / ,
 <ExprArith1> : [VarName] , (, [Number] ,
 <ExprArith1'> : eps , + , - ,
 <Op''> : + , - ,

<ExprArith2> : [VarName] , (, [Number] ,
 <Cond> : [VarName] , not , - , [Number] ,
 <If'> : fi , else ,
 <Cond0> : [VarName] , - , [Number] ,
 <SimpleCond> : [VarName] , (, - , [Number] ,
 <Cond1> :
 <Cond1'> : eps ,
 <BinOp'> : and ,
 <Cond2> :
 <Cond2'> : or , eps ,
 <BinOp''> : or ,
 <Comp> : <= , < , = , > , /= , >= ,

===== STEP 7 =====

<Program> : begin ,
 <Code> : [VarName] , print , read , for , eps ,
 <InstList> : [VarName] , print , read , for , while ,
 <Instruction> : [VarName] , print , read , for , while , if ,
 <InstList'> : eps , ; ,
 <Assign> : [VarName] ,
 <If> : if ,
 <While> : while ,
 <For> : for ,
 <Print> : print ,
 <Read> : read ,
 <ExprArith> : [VarName] , (, - , [Number] ,
 <ExprArith0> : [VarName] , (, [Number] ,
 <ExprArith0'> : eps , * , / ,
 <Op'> : * , / ,
 <ExprArith1> : [VarName] , (, [Number] ,
 <ExprArith1'> : eps , + , - ,
 <Op''> : + , - ,
 <ExprArith2> : [VarName] , (, [Number] ,
 <Cond> : [VarName] , not , - , [Number] ,
 <If'> : fi , else ,
 <Cond0> : [VarName] , (, - , [Number] ,
 <SimpleCond> : [VarName] , (, - , [Number] ,
 <Cond1> :
 <Cond1'> : eps ,
 <BinOp'> : and ,
 <Cond2> :
 <Cond2'> : or , eps ,
 <BinOp''> : or ,
 <Comp> : <= , < , = , > , /= , >= ,

===== STEP 8 =====

<Program> : begin ,
 <Code> : [VarName] , print , read , for , eps , while ,
 <InstList> : [VarName] , print , read , for , while , if ,
 <Instruction> : [VarName] , print , read , for , while , if ,
 <InstList'> : eps , ; ,
 <Assign> : [VarName] ,
 <If> : if ,
 <While> : while ,

```

<For> : for ,
<Print> : print ,
<Read> : read ,
<ExprArith> : [VarName] , ( , - , [Number] ,
<ExprArith0> : [VarName] , ( , [Number] ,
<ExprArith0'> : eps , * , / ,
<Op'> : * , / ,
<ExprArith1> : [VarName] , ( , [Number] ,
<ExprArith1'> : eps , + , - ,
<Op''> : + , - ,
<ExprArith2> : [VarName] , ( , [Number] ,
<Cond> : [VarName] , not , ( , - , [Number] ,
<If'> : fi , else ,
<Cond0> : [VarName] , ( , - , [Number] ,
<SimpleCond> : [VarName] , ( , - , [Number] ,
<Cond1> :
<Cond1'> : eps ,
<BinOp'> : and ,
<Cond2> :
<Cond2'> : or , eps ,
<BinOp''> : or ,
<Comp> : <= , < , = , > , /= , >= ,

```

STEP 9

```

<Program> : begin ,
<Code> : [VarName] , print , read , for , eps , while , if ,
<InstList> : [VarName] , print , read , for , while , if ,
<Instruction> : [VarName] , print , read , for , while , if ,
<InstList'> : eps , ; ,
<Assign> : [VarName] ,
<If> : if ,
<While> : while ,
<For> : for ,
<Print> : print ,
<Read> : read ,
<ExprArith> : [VarName] , ( , - , [Number] ,
<ExprArith0> : [VarName] , ( , [Number] ,
<ExprArith0'> : eps , * , / ,
<Op'> : * , / ,
<ExprArith1> : [VarName] , ( , [Number] ,
<ExprArith1'> : eps , + , - ,
<Op''> : + , - ,
<ExprArith2> : [VarName] , ( , [Number] ,
<Cond> : [VarName] , not , ( , - , [Number] ,
<If'> : fi , else ,
<Cond0> : [VarName] , ( , - , [Number] ,
<SimpleCond> : [VarName] , ( , - , [Number] ,
<Cond1> :
<Cond1'> : eps ,
<BinOp'> : and ,
<Cond2> :
<Cond2'> : or , eps ,
<BinOp''> : or ,
<Comp> : <= , < , = , > , /= , >= ,

```

STEP 10

```

<Program> : begin ,
<Code> : [VarName] , print , read , for , eps , while , if ,
<InstList> : [VarName] , print , read , for , while , if ,
<Instruction> : [VarName] , print , read , for , while , if ,
<InstList'> : eps , ; ,
<Assign> : [VarName] ,
<If> : if ,
<While> : while ,
<For> : for ,
<Print> : print ,
<Read> : read ,
<ExprArith> : [VarName] , ( , - , [Number] ,
<ExprArith0> : [VarName] , ( , [Number] ,
<ExprArith0'> : eps , * , / ,
<Op'> : * , / ,
<ExprArith1> : [VarName] , ( , [Number] ,
<ExprArith1'> : eps , + , - ,
<Op''> : + , - ,
<ExprArith2> : [VarName] , ( , [Number] ,
<Cond> : [VarName] , not , ( , - , [Number] ,
<If'> : fi , else ,
<Cond0> : [VarName] , ( , - , [Number] ,
<SimpleCond> : [VarName] , ( , - , [Number] ,
<Cond1> :
<Cond1'> : eps ,
<BinOp'> : and ,
<Cond2> :
<Cond2'> : or , eps ,
<BinOp''> : or ,
<Comp> : <= , < , = , > , /= , >= ,

```

```

<Program> : begin ,
<Code> : [VarName] , print , read , for , eps , while , if ,
<InstList> : [VarName] , print , read , for , while , if ,
<Instruction> : [VarName] , print , read , for , while , if ,
<InstList'> : eps , ; ,
<Assign> : [VarName] ,
<If> : if ,
<While> : while ,
<For> : for ,
<Print> : print ,
<Read> : read ,
<ExprArith> : [VarName] , ( , - , [Number] ,
<ExprArith0> : [VarName] , ( , [Number] ,
<ExprArith0'> : eps , * , / ,
<Op'> : * , / ,
<ExprArith1> : [VarName] , ( , [Number] ,
<ExprArith1'> : eps , + , - ,
<Op''> : + , - ,
<ExprArith2> : [VarName] , ( , [Number] ,
<Cond> : [VarName] , not , ( , - , [Number] ,
<If'> : fi , else ,
<Cond0> : [VarName] , ( , - , [Number] ,

```

<SimpleCond> : [VarName] , (, - , [Number] ,
 <Cond1> :
 <Cond1'> : eps ,
 <BinOp'> : and ,
 <Cond2> :
 <Cond2'> : or , eps ,
 <BinOp''> : or ,
 <Comp> : <= , < , = , > , /= , >= ,

4. Calcul des différentes listes Follow

===== STEP 0 =====

<Program> :
 <Code> : fi , od , else , end ,
 <InstList> : fi , od , else , end ,
 <Instruction> : fi , od , else , end , ; ,
 <InstList'> : fi , od , else , end ,
 <Assign> : fi , od , else , end , ; ,
 <If> : fi , od , else , end , ; ,
 <While> : fi , od , else , end , ; ,
 <For> : fi , od , else , end , ; ,
 <Print> : fi , od , else , end , ; ,
 <Read> : fi , od , else , end , ; ,
 <ExprArith> : <= , fi , for , do , while , od , else , by , end , if , [
 Number] , read , (,) , * , + , - , /= , / , [VarName] , print , ; ,
 to , < , = , > , >= ,
 <ExprArith0> : <= , fi , for , do , while , od , else , by , end , if , [
 Number] , read , (,) , * , + , - , /= , / , [VarName] , print , ; ,
 to , < , = , > , >= ,
 <ExprArith0'> : <= , fi , for , do , while , od , else , by , end , if , [
 Number] , read , (,) , * , + , - , /= , / , [VarName] , print , ; ,
 to , < , = , > , >= ,
 <Op'> : <= , fi , for , do , while , od , else , by , end , if , [Number]
 , read , (,) , * , + , - , /= , / , [VarName] , print , ; , to , < ,
 = , > , >= ,
 <ExprArith1> : <= , fi , for , do , while , od , else , by , end , if , [
 Number] , read , (,) , * , + , - , /= , / , [VarName] , print , ; ,
 to , < , = , > , >= ,
 <ExprArith1'> : <= , fi , for , do , while , od , else , by , end , if , [
 Number] , read , (,) , * , + , - , /= , / , [VarName] , print , ; ,
 to , < , = , > , >= ,
 <Op''> : <= , fi , for , do , while , od , else , by , end , if , [Number]
 , read , (,) , * , + , - , /= , / , [VarName] , print , ; , to , < ,
 = , > , >= ,
 <ExprArith2> : <= , fi , for , do , while , od , else , by , end , if , [
 Number] , read , (,) , * , + , - , /= , / , [VarName] , print , ; ,
 to , < , = , > , >= ,
 <Cond> : read , fi , or , for , then , do , while , [VarName] , print , od
 , else , end , ; , if ,
 <If'> : fi , od , else , end , ; ,
 <Cond0> : read , fi , or , for , then , do , while , [VarName] , print ,
 od , else , end , ; , if ,
 <SimpleCond> : read , fi , or , for , then , do , while , [VarName] ,
 print , od , else , end , ; , if ,

```

<Cond1> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond1'> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<BinOp'> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond2> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond2'> :
<BinOp''> : [VarName] , not , or , ( , - , [Number] ,
<Comp> : [VarName] , ( , - , [Number] ,

```

STEP 1

```

<Program> :
<Code> : fi , od , else , end ,
<InstList> : fi , od , else , end ,
<Instruction> : fi , od , else , end , ; ,
<InstList'> : fi , od , else , end ,
<Assign> : fi , od , else , end , ; ,
<If> : fi , od , else , end , ; ,
<While> : fi , od , else , end , ; ,
<For> : fi , od , else , end , ; ,
<Print> : fi , od , else , end , ; ,
<Read> : fi , od , else , end , ; ,
<ExprArith> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<ExprArith0> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<ExprArith0'> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<Op'> : <= , fi , for , do , while , od , else , by , end , if , [Number]
        , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; , to , < ,
        = , > , >= ,
<ExprArith1> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<ExprArith1'> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<Op''> : <= , fi , for , do , while , od , else , by , end , if , [Number]
        , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; , to , < ,
        = , > , >= ,
<ExprArith2> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<Cond> : read , fi , or , for , then , do , while , [VarName] , print , od
        , else , end , ; , if ,
<If'> : fi , od , else , end , ; ,
<Cond0> : read , fi , or , for , then , do , while , [VarName] , print ,
        od , else , end , ; , if ,
<SimpleCond> : read , fi , or , for , then , do , while , [VarName] ,
        print , od , else , end , ; , if ,

```

```

<Cond1> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond1'> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<BinOp'> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond2> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond2'> :
<BinOp''> : [VarName] , not , or , ( , - , [Number] ,
<Comp> : [VarName] , ( , - , [Number] ,

```

STEP 2

```

<Program> : eps ,
<Code> : fi , od , else , end ,
<InstList> : fi , od , else , end ,
<Instruction> : fi , od , else , end , ; ,
<InstList'> : fi , od , else , end ,
<Assign> : fi , od , else , end , ; ,
<If> : fi , od , else , end , ; ,
<While> : fi , od , else , end , ; ,
<For> : fi , od , else , end , ; ,
<Print> : fi , od , else , end , ; ,
<Read> : fi , od , else , end , ; ,
<ExprArith> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<ExprArith0> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<ExprArith0'> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<Op'> : <= , fi , for , do , while , od , else , by , end , if , [Number]
        , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; , to , < ,
        = , > , >= ,
<ExprArith1> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<ExprArith1'> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<Op''> : <= , fi , for , do , while , od , else , by , end , if , [Number]
        , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; , to , < ,
        = , > , >= ,
<ExprArith2> : <= , fi , for , do , while , od , else , by , end , if , [
               Number] , read , ( , ) , * , + , - , /= , / , [VarName] , print , ; ,
               to , < , = , > , >= ,
<Cond> : read , fi , or , for , then , do , while , [VarName] , print , od
        , else , end , ; , if ,
<If'> : fi , od , else , end , ; ,
<Cond0> : read , fi , or , for , then , do , while , [VarName] , print ,
        od , else , end , ; , if ,
<SimpleCond> : read , fi , or , for , then , do , while , [VarName] ,
               print , od , else , end , ; , if ,

```

```

<Cond1> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond1'> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<BinOp'> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond2> : read , fi , or , for , then , do , while , [VarName] , print ,
         od , else , end , ; , if ,
<Cond2'> : eps ,
<BinOp''> : [VarName] , not , or , ( , - , [Number] ,
<Comp> : [VarName] , ( , - , [Number] ,

```
