# INFO-F-403 : Language theory and compiling
# Rapport projet partie 3 - Génération de code

Simon Picard
Arnaud Rosette

13 février 2015

# Table des matières

# 1 Introduction

Dans cette partie du projet, il nous est demandé de modifier la grammaire donnée dans l'énoncé afin de rendre celle-ci LL(1). Il faut donc appliquer toute une série de règles et d'algorithmes pour rendre cette grammaire moins ambigüe et LL(1), dans le but de construire "l'action table" à partir des ensembles first et follow. L'inclusion des fonctions dans la grammaire fait partie d'un bonus que nous avons décidé de réaliser.

# 2 Transformation de la grammaire donnée

## 2.1 Opérateurs binaires et unaires

La première étape pour rendre la grammaire LL(1) fut la distinction des deux types d'opérateurs : les opérateurs binaires qui agissent sur deux expressions l'une située à gauche et l'autre à droite de ces opérateurs et les opérateurs unaires qui agissent sur une seule expression située à droite de ces opérateurs. Il faut différencier les deux types d'opérateurs afin de ne pas avoir des expressions du style : »4, *2, 6|,... .Les opérateurs unaires sont au nombre de quatre : ! (negation), $\sim$ (not bit à bit), + et -. Les opérateurs + et - sont également des opérateurs binaires.

## 2.2 Priorité et associativité des opérateurs

Lors de cette étape, nous avons modifié la grammaire afin de fixer la priorité et l'associativité des différents opérateurs, ceci afin de rendre la grammaire moins ambigüe. Nous avons remarqué que les opérateurs unaires étaient plus prioritaires que les opérateurs binaires et que les opérateurs unaires étaient tous associatifs à droite, tandis que les opérateurs binaires l'étaient tous à gauche. Pour fixer la priorité, nous avons mis les opérateurs les moins prioritaires le plus "haut" possible dans la grammaire et les plus prioritaires le plus "bas" possible. Dans la grammaire ci-dessous où le "start symbol" est E, l'opérateur "+" est considéré comme plus "haut" que l'opérateur "*" dans la grammaire car il est possible d'obtenir l'opérateur "+" en utilisant moins de règles de production à partir du "start symbol" que pour obtenir l'opérateur "*".

```
E->E + T
 ->T
T->T * F
 ->F
F->ID
 ->( E )
```

## 2.3 Suppression des récursions à gauche

L'étape suivante fut l'étape de suppression des récursions à gauche qui sont incompatibles avec les "top-down parser" car cela fait entrer ce genre de parser dans une boucle ou récursion infinie. Cette étape rend tout opérateur associatif à droite si l'on ne fait rien pour résoudre ce problème lors de l'analyse sémantique.

## 2.4 Left factoring

Cette étape rassemble les règles de production d'une variable qui ont un préfixe commun en une seule règle de production qui contient ce préfixe commun et une nouvelle variable. Cette nouvelle variable possède des règles de production vers les différents suffixes qui étaient présents à l'origine dans les différentes règles de production qui ont été mises en commun. Ceci est nécessaire car le parser qu'on va créer est LL(1), il faut donc qu'il puisse choisir la bonne règle de production en regardant seulement le prochain token qui est en entrée.

## 2.5 Les variables <Instruction> et <InstructionList>

Dans la grammaire donnée, à chaque fois que la variable <Instruction> se trouvait dans la partie droite d'une règle de production excepté lorsque la partie gauche de la règle était <InstructionList>,

il y avait une autre règle de production pour cette même partie gauche où <Instruction> était remplacé par <InstructionList>. Par exemple, <If>→<Expression> <Empty> <InstructionList> <IfEnd> et <If>→<Expression> <Empty> <Instruction> <IfEnd>. Ceci permettant de ne pas mettre de END_OF_INSTRUCTION à la fin d'une <Instruction> lorsque le corps d'un block (ici, le corps du "if") ne contenait qu'une seule <Instruction>. Le problème posé par le doublement systématique des règles de production contenant des instructions (une règle pour <Instruction> et une autre pour <InstructionList>) est que ce n'est pas factorisé à gauche et que le first de <InstructionList> peut être <Instruction>. Afin de résoudre ce problème, nous avons décidé de ne plus utilisé que la variable <InstructionList> dans les autres règles de production. Ainsi, on évite le doublement des règles de production. Une <InstructionList> est une liste d'<Instruction> séparées par des END_OF_INSTRUCTION, avec la dernière <Instruction> qui n'est pas nécessairement suivie d'un END_OF_INSTRUCTION. Ceci permet d'avoir un programme du style : if(a>b) ;a=10 end ; Donc seule une <InstructionList> permet de produire des <Instruction>. De plus, vu qu'une <InstructionList> peut contenir des instructions vide (une instruction contenant seulement END_OF_INSTRUCTION) la variable <Empty> n'est plus nécessaire car celle-ci servait uniquement à indiquer qu'il fallait au moins un terminal END_OF_INSTRUCTION.

### 2.6 Fonctions

Les fonctions amènent deux types d'instructions en plus : les définitions de fonction et les appels de fonction. Les appels sont considérés comme des expressions atomiques afin de pourvoir mettre des appels de fonction au sein d'une expression. Par exemple, a=foo()+5 ; De plus, il faut qu'une fonction puisse être appelée sans avoir besoin de faire une assignation ou un autre type d'instruction car une fonction peut ne rien retourner et simplement avoir un effet de bord. Par exemple, la fonction println est une fonction qui ne retourne rien mais produit un effet de bord qui est l'affichage dans le terminal. Il faut donc pouvoir faire, par exemple, println(a). C'est pour cela qu'il faut qu'un appel de fonction soit une instruction à part entière.

### 2.7 Instructions commencant par un identifier

Plusieurs instructions commencent par un identifier. Il s'agit des assignations, des déclarations de variables et des appels de fonctions. Afin que la grammaire soit LL(1) il faut factoriser ces instructions. C'est pour cela que la variable <IdentifierInstruction> a été créée. Celle-ci représente une instruction qui commence par un identifier. La variable <IdentifierInstructionTail> a été également créée afin de distinguer les différents types d'instructions qui commencent par un identifier. La distinction est après aisément faite car lorsque le symbole suivant l'identifier est "=", on sait que c'est une assignation, lorsque ce symbole est " : :", on sait que c'est une déclaration de variable et lorsque que ce symbole est "(", on sait que c'est un appel de fonction.

### 2.8 Automatisation

Nous avons développé une série d'algorithm qui permettent de supprimer la reccurssion à gauche, de factoriser à gauche, de supprimer les symboles inutiles et de générer une table d'action ce qui nous permet de vérifier si une grammaire est bien LL(1) via l'action table ou de nous aider à la transformer en grammaire LL(1) grçace aux autres algorithmes.

## 3 Grammaire

Voici la grammaire donnée dans l'énoncé et transformée en grammaire LL(1). A chaque règle de production correspond un numéro qui identifie cette règle.

| | | | |
|---|---|---|---|
| [1] | <Program> | → | <InstructionList> |
| [2] | <Instruction> | → | <IdentifierInstruction> |
| [3] | | → | <ConstDefinition> |
| [4] | | → | <Block> |
| [5] | | → | <Loop> |
| [6] | | → | <BuiltInFunctionCall> |
| [7] | | → | <FunctionDefinition> |
| [8] | <InstructionList> | → | <Instruction> <InstructionListTail> |

| | | | |
|---|---|---|---|
| [9] | | → | <InstructionListTail> |
| [10] | <InstructionListTail> | → | END_OF_INSTRUCTION <InstructionList> |
| [11] | | → | $\epsilon$ |
| [12] | <IdentifierInstruction> | → | IDENTIFIER <IdentifierInstructionTail> |
| [13] | <IdentifierInstructionTail> | → | <AssignationTail> |
| [14] | | → | TYPE_DEFINITION <Type> |
| [15] | | → | <FunctionCallTail> |
| [16] | <AssignationTail> | → | ASSIGNATION <Expression> |
| [17] | | → | COMMA IDENTIFIER <AssignationTail> COMMA <Expression> |
| [18] | <ConstDefinition> | → | CONST IDENTIFIER <AssignationTail> |
| [19] | <Block> | → | LET IDENTIFIER <AssignationTail> END_OF_INSTRUCTION <InstructionList> END |
| [20] | <Loop> | → | <If> |
| [21] | | → | WHILE <Expression> END_OF_INSTRUCTION <InstructionList> END |
| [22] | | → | FOR IDENTIFIER ASSIGNATION <Expression> TERNARY_ELSE <Expression> <ForTail> |
| [23] | <ForTail> | → | END_OF_INSTRUCTION <InstructionList> END |
| [24] | | → | TERNARY_ELSE <Expression> END_OF_INSTRUCTION <InstructionList> END |
| [25] | <Type> | → | BOOLEAN_TYPE |
| [26] | | → | REAL_TYPE |
| [27] | | → | INTEGER_TYPE |
| [28] | <Expression> | → | <BinaryExpression> <TernaryIfExpression> |
| [29] | <TernaryIfExpression> | → | TERNARY_IF <Expression> <TernaryElseExpression> |
| [30] | | → | $\epsilon$ |
| [31] | <TernaryElseExpression> | → | TERNARY_ELSE <Expression> |
| [32] | <AtomicExpression> | → | <AtomicIdentifierExpression> |
| [33] | | → | INTEGER |
| [34] | | → | REAL |
| [35] | | → | BOOLEAN |
| [36] | | → | <BuiltInFunctionCall> |
| [37] | <AtomicIdentifierExpression> | → | IDENTIFIER <AtomicIdentifierExpressionTail> |
| [38] | <AtomicIdentifierExpressionTail> | → | <FunctionCallTail> |
| [39] | | → | $\epsilon$ |
| [40] | <UnaryExpression> | → | NEGATION <UnaryExpression> |
| [41] | | → | <UnaryBitwiseNotExpression> |
| [42] | <UnaryBitwiseNotExpression> | → | BITWISE_NOT <UnaryBitwiseNotExpression> |
| [43] | | → | <UnaryMinusPlusExpression> |
| [44] | <UnaryMinusPlusExpression> | → | MINUS <UnaryMinusPlusExpression> |
| [45] | | → | PLUS <UnaryMinusPlusExpression> |
| [46] | | → | <UnaryAtomicExpression> |
| [47] | <UnaryAtomicExpression> | → | <AtomicExpression> |
| [48] | | → | LEFT_PARENTHESIS <Expression> RIGHT_PARENTHESIS |
| [49] | <BinaryExpression> | → | <BinaryLazyOrExpression> <BinaryExpression'> |
| [50] | <BinaryExpression'> | → | LAZY_OR <BinaryLazyOrExpression> <BinaryExpression'> |
| [51] | | → | $\epsilon$ |
| [52] | <BinaryLazyOrExpression> | → | <BinaryLazyAndExpression> <BinaryLazyOrExpression'> |
| [53] | <BinaryLazyOrExpression'> | → | LAZY_AND <BinaryLazyAndExpression> <BinaryLazyOrExpression'> |
| [54] | | → | $\epsilon$ |

[55] <BinaryLazyAndExpression> → <BinaryNumericExpression> <BinaryLazyAndExpression'>
[56] <BinaryLazyAndExpression'> → GREATER_THAN <BinaryNumericExpression> <BinaryLazyAndExpression'>
[57] → LESS_THAN <BinaryNumericExpression> <BinaryLazyAndExpression'>
[58] → GREATER_OR_EQUALS_THAN <BinaryNumericExpression> <BinaryLazyAndExpression'>
[59] → LESS_OR_EQUALS_THAN <BinaryNumericExpression> <BinaryLazyAndExpression'>
[60] → EQUALITY <BinaryNumericExpression> <BinaryLazyAndExpression'>
[61] → INEQUALITY <BinaryNumericExpression> <BinaryLazyAndExpression'>
[62] → $\epsilon$
[63] <BinaryNumericExpression> → <BinaryTermExpression> <BinaryNumericExpression'>
[64] <BinaryNumericExpression'> → PLUS <BinaryTermExpression> <BinaryNumericExpression'>
[65] → MINUS <BinaryTermExpression> <BinaryNumericExpression'>
[66] → BITWISE_OR <BinaryTermExpression> <BinaryNumericExpression'>
[67] → BITWISE_XOR <BinaryTermExpression> <BinaryNumericExpression'>
[68] → $\epsilon$
[69] <BinaryTermExpression> → <BinaryShiftedExpression> <BinaryTermExpression'>
[70] <BinaryTermExpression'> → ARITHMETIC_SHIFT_LEFT <BinaryShiftedExpression> <BinaryTermExpression'>
[71] → ARITHMETIC_SHIFT_RIGHT <BinaryShiftedExpression> <BinaryTermExpression'>
[72] → $\epsilon$
[73] <BinaryShiftedExpression> → <BinaryFactorExpression> <BinaryShiftedExpression'>
[74] <BinaryShiftedExpression'> → TIMES <BinaryFactorExpression> <BinaryShiftedExpression'>
[75] → DIVIDE <BinaryFactorExpression> <BinaryShiftedExpression'>
[76] → REMAINDER <BinaryFactorExpression> <BinaryShiftedExpression'>
[77] → BITWISE_AND <BinaryFactorExpression> <BinaryShiftedExpression'>
[78] → INVERSE_DIVIDE <BinaryFactorExpression> <BinaryShiftedExpression'>
[79] → $\epsilon$
[80] <BinaryFactorExpression> → <UnaryExpression> <BinaryFactorExpression'>
[81] <BinaryFactorExpression'> → POWER <UnaryExpression> <BinaryFactorExpression'>
[82] → $\epsilon$
[83] <If> → IF <Expression> END_OF_INSTRUCTION <InstructionList> <IfEnd>

[84] <IfEnd> → ELSE_IF <Expression> END_OF_INSTRUCTION <InstructionList> <IfEnd>
[85] → ELSE <InstructionList> END
[86] → END
[87] <BuiltInFunctionCall> → READ_REAL LEFT_PARENTHESIS RIGHT_PARENTHESIS
[88] → READ_INTEGER LEFT_PARENTHESIS RIGHT_PARENTHESIS
[89] → INTEGER_CAST LEFT_PARENTHESIS <Expression> RIGHT_PARENTHESIS
[90] → REAL_CAST LEFT_PARENTHESIS <Expression> RIGHT_PARENTHESIS
[91] → BOOLEAN_CAST LEFT_PARENTHESIS <Expression> RIGHT_PARENTHESIS
[92] → PRINTLN LEFT_PARENTHESIS <Expression> RIGHT_PARENTHESIS
[93] <FunctionCallTail> → LEFT_PARENTHESIS <Parameter> RIGHT_PARENTHESIS
[94] <Parameter> → <Expression> <ParameterTail>
[95] → $\epsilon$
[96] <ParameterTail> → COMMA <Expression> <ParameterTail>
[97] → $\epsilon$
[98] <FunctionDefinition> → FUNCTION IDENTIFIER LEFT_PARENTHESIS <Argument> RIGHT_PARENTHESIS <InstructionList> <FunctionDefinitionEnd>
[99] <FunctionDefinitionEnd> → RETURN <Expression> END
[100] → END
[101] <Argument> → IDENTIFIER TYPE_DEFINITION <Type> <ArgumentTail>
[102] → $\epsilon$
[103] <ArgumentTail> → COMMA IDENTIFIER TYPE_DEFINITION <Type> <ArgumentTail>
[104] → $\epsilon$

# 4 Ensembles First et Follow

Dans le tableau suivant, EPSILON_VALUE $\equiv \epsilon$.

| Variable | First | Follow |
|---|---|---|
| <Program> | FUNCTION, WHILE, READ_REAL EPSILON_VALUE IDENTIFIER, CONST BOOLEAN_CAST, PRINTLN END_OF_INSTRUCTION READ_INTEGER, FOR INTEGER_CAST, LET, IF REAL_CAST | |
| <Instruction> | FUNCTION, READ_INTEGER FOR, WHILE, READ_REAL INTEGER_CAST BOOLEAN_CAST, CONST IDENTIFIER, PRINTLN, LET, IF REAL_CAST | END, END_OF_INSTRUCTION ELSE_IF, ELSE, RETURN |

| | | |
|---|---|---|
| <InstructionList> | FUNCTION, WHILE, READ_REAL<br>EPSILON_VALUE<br>BOOLEAN_CAST, IDENTIFIER<br>CONST, PRINTLN<br>END_OF_INSTRUCTION<br>READ_INTEGER, FOR<br>INTEGER_CAST, LET, IF<br>REAL_CAST | END, ELSE_IF, ELSE, RETURN |
| <InstructionListTail> | EPSILON_VALUE<br>END_OF_INSTRUCTION | END, ELSE_IF, ELSE, RETURN |
| <IdentifierInstruction> | IDENTIFIER | END, END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <IdentifierInstructionTail> | ASSIGNATION<br>TYPE_DEFINITION, COMMA<br>LEFT_PARENTHESIS | END, END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <AssignationTail> | ASSIGNATION, COMMA | END, COMMA<br>END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <ConstDefinition> | CONST | END, END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <Block> | LET | END, END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <Loop> | FOR, WHILE, IF | END, END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <ForTail> | END_OF_INSTRUCTION<br>TERNARY_ELSE | END, END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <Type> | REAL_TYPE, BOOLEAN_TYPE<br>INTEGER_TYPE | RIGHT_PARENTHESIS, END<br>COMMA<br>END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <Expression> | READ_INTEGER, INTEGER<br>REAL, PLUS, BOOLEAN<br>NEGATION, MINUS<br>BITWISE_NOT, READ_REAL<br>INTEGER_CAST<br>LEFT_PARENTHESIS<br>BOOLEAN_CAST, IDENTIFIER<br>PRINTLN, REAL_CAST | RIGHT_PARENTHESIS, END<br>COMMA<br>END_OF_INSTRUCTION<br>ELSE_IF, ELSE<br>TERNARY_ELSE, RETURN |
| <TernaryIfExpression> | TERNARY_IF<br>EPSILON_VALUE | RIGHT_PARENTHESIS, END<br>COMMA<br>END_OF_INSTRUCTION<br>ELSE_IF, ELSE<br>TERNARY_ELSE, RETURN |
| <TernaryElseExpression> | TERNARY_ELSE | RIGHT_PARENTHESIS, END<br>COMMA<br>END_OF_INSTRUCTION<br>ELSE_IF, ELSE<br>TERNARY_ELSE, RETURN |

| | | |
|---|---|---|
| <AtomicExpression> | READ_INTEGER, INTEGER<br>REAL, BOOLEAN, READ_REAL<br>INTEGER_CAST<br>BOOLEAN_CAST, IDENTIFIER<br>PRINTLN, REAL_CAST | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, BITWISE_AND<br>ARITHMETIC_SHIFT_LEFT<br>TERNARY_IF, GREATER_THAN<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, POWER<br>INEQUALITY, BITWISE_OR<br>MINUS, RIGHT_PARENTHESIS<br>BITWISE_XOR, REMAINDER<br>COMMA<br>END_OF_INSTRUCTION<br>RETURN, LESS_THAN<br>LAZY_AND, PLUS, LAZY_OR<br>INVERSE_DIVIDE, END, TIMES<br>ELSE_IF, DIVIDE<br>GREATER_OR_EQUALS_THAN |
| <AtomicIdentifierExpression> | IDENTIFIER | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, BITWISE_AND<br>ARITHMETIC_SHIFT_LEFT<br>TERNARY_IF, GREATER_THAN<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, POWER<br>INEQUALITY, BITWISE_OR<br>MINUS, RIGHT_PARENTHESIS<br>BITWISE_XOR, REMAINDER<br>COMMA<br>END_OF_INSTRUCTION<br>RETURN, LESS_THAN<br>LAZY_AND, PLUS, LAZY_OR<br>INVERSE_DIVIDE, END, TIMES<br>ELSE_IF, DIVIDE<br>GREATER_OR_EQUALS_THAN |
| <AtomicIdentifierExpressionTail> | EPSILON_VALUE<br>LEFT_PARENTHESIS | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, BITWISE_AND<br>ARITHMETIC_SHIFT_LEFT<br>TERNARY_IF, GREATER_THAN<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, POWER<br>INEQUALITY, BITWISE_OR<br>MINUS, RIGHT_PARENTHESIS<br>BITWISE_XOR, REMAINDER<br>COMMA<br>END_OF_INSTRUCTION<br>RETURN, LESS_THAN<br>LAZY_AND, PLUS, LAZY_OR<br>INVERSE_DIVIDE, END, TIMES<br>ELSE_IF, DIVIDE<br>GREATER_OR_EQUALS_THAN |

| | | |
|---|---|---|
| <UnaryExpression> | INTEGER, REAL, BOOLEAN<br>NEGATION, BITWISE_NOT<br>READ_REAL<br>LEFT_PARENTHESIS<br>IDENTIFIER, BOOLEAN_CAST<br>PRINTLN, READ_INTEGER<br>PLUS, MINUS, INTEGER_CAST<br>REAL_CAST | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, BITWISE_AND<br>ARITHMETIC_SHIFT_LEFT<br>TERNARY_IF, GREATER_THAN<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, POWER<br>INEQUALITY, BITWISE_OR<br>MINUS, RIGHT_PARENTHESIS<br>BITWISE_XOR, REMAINDER<br>COMMA<br>END_OF_INSTRUCTION<br>RETURN, LESS_THAN<br>LAZY_AND, PLUS, LAZY_OR<br>INVERSE_DIVIDE, END, TIMES<br>ELSE_IF, DIVIDE<br>GREATER_OR_EQUALS_THAN |
| <UnaryBitwiseNotExpression> | READ_INTEGER, INTEGER<br>REAL, PLUS, BOOLEAN, MINUS<br>BITWISE_NOT, READ_REAL<br>INTEGER_CAST<br>LEFT_PARENTHESIS<br>BOOLEAN_CAST, IDENTIFIER<br>PRINTLN, REAL_CAST | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, BITWISE_AND<br>ARITHMETIC_SHIFT_LEFT<br>TERNARY_IF, GREATER_THAN<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, POWER<br>INEQUALITY, BITWISE_OR<br>MINUS, RIGHT_PARENTHESIS<br>BITWISE_XOR, REMAINDER<br>COMMA<br>END_OF_INSTRUCTION<br>RETURN, LESS_THAN<br>LAZY_AND, PLUS, LAZY_OR<br>INVERSE_DIVIDE, END, TIMES<br>ELSE_IF, DIVIDE<br>GREATER_OR_EQUALS_THAN |
| <UnaryMinusPlusExpression> | READ_INTEGER, INTEGER<br>REAL, PLUS, BOOLEAN, MINUS<br>READ_REAL, INTEGER_CAST<br>LEFT_PARENTHESIS<br>BOOLEAN_CAST, IDENTIFIER<br>PRINTLN, REAL_CAST | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, BITWISE_AND<br>ARITHMETIC_SHIFT_LEFT<br>TERNARY_IF, GREATER_THAN<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, POWER<br>INEQUALITY, BITWISE_OR<br>MINUS, RIGHT_PARENTHESIS<br>BITWISE_XOR, REMAINDER<br>COMMA<br>END_OF_INSTRUCTION<br>RETURN, LESS_THAN<br>LAZY_AND, PLUS, LAZY_OR<br>INVERSE_DIVIDE, END, TIMES<br>ELSE_IF, DIVIDE<br>GREATER_OR_EQUALS_THAN |

| | | |
|---|---|---|
| <UnaryAtomicExpression> | READ_INTEGER, INTEGER REAL, BOOLEAN, READ_REAL INTEGER_CAST LEFT_PARENTHESIS BOOLEAN_CAST, IDENTIFIER PRINTLN, REAL_CAST | ARITHMETIC_SHIFT_RIGHT EQUALITY, BITWISE_AND ARITHMETIC_SHIFT_LEFT TERNARY_IF, GREATER_THAN LESS_OR_EQUALS_THAN ELSE, TERNARY_ELSE, POWER INEQUALITY, BITWISE_OR MINUS, RIGHT_PARENTHESIS BITWISE_XOR, REMAINDER COMMA END_OF_INSTRUCTION RETURN, LESS_THAN LAZY_AND, PLUS, LAZY_OR INVERSE_DIVIDE, END, TIMES ELSE_IF, DIVIDE GREATER_OR_EQUALS_THAN |
| <BinaryExpression> | READ_INTEGER, INTEGER REAL, PLUS, BOOLEAN NEGATION, MINUS BITWISE_NOT, READ_REAL INTEGER_CAST BOOLEAN_CAST LEFT_PARENTHESIS IDENTIFIER, PRINTLN REAL_CAST | TERNARY_IF RIGHT_PARENTHESIS, END COMMA END_OF_INSTRUCTION ELSE_IF, ELSE TERNARY_ELSE, RETURN |
| <BinaryExpression'> | LAZY_OR, EPSILON_VALUE | TERNARY_IF RIGHT_PARENTHESIS, END COMMA END_OF_INSTRUCTION ELSE_IF, ELSE TERNARY_ELSE, RETURN |
| <BinaryLazyOrExpression> | READ_INTEGER, INTEGER REAL, PLUS, BOOLEAN NEGATION, MINUS BITWISE_NOT, READ_REAL INTEGER_CAST LEFT_PARENTHESIS BOOLEAN_CAST, IDENTIFIER PRINTLN, REAL_CAST | TERNARY_IF, LAZY_OR RIGHT_PARENTHESIS, END COMMA END_OF_INSTRUCTION ELSE_IF, ELSE TERNARY_ELSE, RETURN |
| <BinaryLazyOrExpression'> | LAZY_AND, EPSILON_VALUE | TERNARY_IF, LAZY_OR RIGHT_PARENTHESIS, END COMMA END_OF_INSTRUCTION ELSE_IF, ELSE TERNARY_ELSE, RETURN |
| <BinaryLazyAndExpression> | READ_INTEGER, INTEGER REAL, PLUS, BOOLEAN NEGATION, MINUS BITWISE_NOT, READ_REAL INTEGER_CAST BOOLEAN_CAST LEFT_PARENTHESIS IDENTIFIER, PRINTLN REAL_CAST | LAZY_AND, TERNARY_IF LAZY_OR RIGHT_PARENTHESIS, END COMMA END_OF_INSTRUCTION ELSE_IF, ELSE TERNARY_ELSE, RETURN |

| | | |
|---|---|---|
| <BinaryLazyAndExpression'> | EQUALITY, INEQUALITY<br>GREATER_THAN<br>EPSILON_VALUE<br>LESS_OR_EQUALS_THAN<br>GREATER_OR_EQUALS_THAN<br>LESS_THAN | LAZY_AND, TERNARY_IF<br>LAZY_OR<br>RIGHT_PARENTHESIS, END<br>COMMA<br>END_OF_INSTRUCTION<br>ELSE_IF, ELSE<br>TERNARY_ELSE, RETURN |
| <BinaryNumericExpression> | READ_INTEGER, INTEGER<br>REAL, PLUS, BOOLEAN<br>NEGATION, MINUS<br>BITWISE_NOT, READ_REAL<br>INTEGER_CAST<br>LEFT_PARENTHESIS<br>BOOLEAN_CAST, IDENTIFIER<br>PRINTLN, REAL_CAST | EQUALITY, TERNARY_IF<br>GREATER_THAN<br>RIGHT_PARENTHESIS, COMMA<br>END_OF_INSTRUCTION<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, RETURN<br>LESS_THAN, LAZY_AND<br>INEQUALITY, LAZY_OR, END<br>ELSE_IF<br>GREATER_OR_EQUALS_THAN |
| <BinaryNumericExpression'> | BITWISE_OR, PLUS, MINUS<br>BITWISE_XOR<br>EPSILON_VALUE | EQUALITY, TERNARY_IF<br>GREATER_THAN<br>RIGHT_PARENTHESIS, COMMA<br>END_OF_INSTRUCTION<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, RETURN<br>LESS_THAN, LAZY_AND<br>INEQUALITY, LAZY_OR, END<br>ELSE_IF<br>GREATER_OR_EQUALS_THAN |
| <BinaryTermExpression> | READ_INTEGER, INTEGER<br>REAL, PLUS, BOOLEAN<br>NEGATION, MINUS<br>BITWISE_NOT, READ_REAL<br>INTEGER_CAST<br>BOOLEAN_CAST<br>LEFT_PARENTHESIS<br>IDENTIFIER, PRINTLN<br>REAL_CAST | EQUALITY, TERNARY_IF<br>RIGHT_PARENTHESIS<br>GREATER_THAN<br>BITWISE_XOR, COMMA<br>END_OF_INSTRUCTION<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, RETURN<br>LESS_THAN, LAZY_AND<br>INEQUALITY, BITWISE_OR<br>PLUS, LAZY_OR, MINUS, END<br>ELSE_IF<br>GREATER_OR_EQUALS_THAN |
| <BinaryTermExpression'> | ARITHMETIC_SHIFT_RIGHT<br>ARITHMETIC_SHIFT_LEFT<br>EPSILON_VALUE | EQUALITY, TERNARY_IF<br>RIGHT_PARENTHESIS<br>GREATER_THAN<br>BITWISE_XOR, COMMA<br>END_OF_INSTRUCTION<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, RETURN<br>LESS_THAN, LAZY_AND<br>INEQUALITY, BITWISE_OR<br>PLUS, LAZY_OR, MINUS, END<br>ELSE_IF<br>GREATER_OR_EQUALS_THAN |

| | | |
|---|---|---|
| <BinaryShiftedExpression> | READ_INTEGER, INTEGER<br>REAL, PLUS, BOOLEAN<br>NEGATION, MINUS<br>BITWISE_NOT, READ_REAL<br>INTEGER_CAST<br>LEFT_PARENTHESIS<br>BOOLEAN_CAST, IDENTIFIER<br>PRINTLN, REAL_CAST | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, TERNARY_IF<br>ARITHMETIC_SHIFT_LEFT<br>RIGHT_PARENTHESIS<br>GREATER_THAN<br>BITWISE_XOR, COMMA<br>END_OF_INSTRUCTION<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, RETURN<br>LESS_THAN, LAZY_AND<br>INEQUALITY, PLUS<br>BITWISE_OR, LAZY_OR, MINUS<br>END, ELSE_IF<br>GREATER_OR_EQUALS_THAN |
| <BinaryShiftedExpression'> | BITWISE_AND<br>INVERSE_DIVIDE<br>REMAINDER, TIMES<br>EPSILON_VALUE, DIVIDE | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, TERNARY_IF<br>ARITHMETIC_SHIFT_LEFT<br>RIGHT_PARENTHESIS<br>GREATER_THAN<br>BITWISE_XOR, COMMA<br>END_OF_INSTRUCTION<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE, RETURN<br>LESS_THAN, LAZY_AND<br>INEQUALITY, PLUS<br>BITWISE_OR, LAZY_OR, MINUS<br>END, ELSE_IF<br>GREATER_OR_EQUALS_THAN |
| <BinaryFactorExpression> | READ_INTEGER, INTEGER<br>REAL, PLUS, BOOLEAN<br>NEGATION, MINUS<br>BITWISE_NOT, READ_REAL<br>INTEGER_CAST<br>BOOLEAN_CAST<br>LEFT_PARENTHESIS<br>IDENTIFIER, PRINTLN<br>REAL_CAST | ARITHMETIC_SHIFT_RIGHT<br>EQUALITY, BITWISE_AND<br>ARITHMETIC_SHIFT_LEFT<br>TERNARY_IF, GREATER_THAN<br>LESS_OR_EQUALS_THAN<br>ELSE, TERNARY_ELSE<br>INEQUALITY, BITWISE_OR<br>MINUS, RIGHT_PARENTHESIS<br>BITWISE_XOR, REMAINDER<br>COMMA<br>END_OF_INSTRUCTION<br>RETURN, LESS_THAN<br>LAZY_AND, PLUS, LAZY_OR<br>INVERSE_DIVIDE, END, TIMES<br>ELSE_IF, DIVIDE<br>GREATER_OR_EQUALS_THAN |

| | | |
|---|---|---|
| <BinaryFactorExpression'> | POWER, EPSILON_VALUE | ARITHMETIC_SHIFT_RIGHT EQUALITY, BITWISE_AND ARITHMETIC_SHIFT_LEFT TERNARY_IF, GREATER_THAN LESS_OR_EQUALS_THAN ELSE, TERNARY_ELSE INEQUALITY, BITWISE_OR MINUS, RIGHT_PARENTHESIS BITWISE_XOR, REMAINDER COMMA END_OF_INSTRUCTION RETURN, LESS_THAN LAZY_AND, PLUS, LAZY_OR INVERSE_DIVIDE, END, TIMES ELSE_IF, DIVIDE GREATER_OR_EQUALS_THAN |
| <If> | IF | END, END_OF_INSTRUCTION ELSE_IF, ELSE, RETURN |
| <IfEnd> | END, ELSE_IF, ELSE | END, END_OF_INSTRUCTION ELSE_IF, ELSE, RETURN |
| <BuiltInFunctionCall> | READ_INTEGER, READ_REAL INTEGER_CAST BOOLEAN_CAST, PRINTLN REAL_CAST | ARITHMETIC_SHIFT_RIGHT EQUALITY, BITWISE_AND TERNARY_IF ARITHMETIC_SHIFT_LEFT GREATER_THAN LESS_OR_EQUALS_THAN TERNARY_ELSE, ELSE, POWER INEQUALITY, BITWISE_OR MINUS, RIGHT_PARENTHESIS REMAINDER, BITWISE_XOR COMMA END_OF_INSTRUCTION RETURN, LESS_THAN LAZY_AND, PLUS, LAZY_OR INVERSE_DIVIDE, TIMES, END ELSE_IF, DIVIDE GREATER_OR_EQUALS_THAN |
| <FunctionCallTail> | LEFT_PARENTHESIS | ARITHMETIC_SHIFT_RIGHT EQUALITY, BITWISE_AND TERNARY_IF ARITHMETIC_SHIFT_LEFT GREATER_THAN LESS_OR_EQUALS_THAN TERNARY_ELSE, ELSE, POWER INEQUALITY, BITWISE_OR MINUS, RIGHT_PARENTHESIS REMAINDER, BITWISE_XOR COMMA END_OF_INSTRUCTION RETURN, LESS_THAN LAZY_AND, PLUS, LAZY_OR INVERSE_DIVIDE, TIMES, END ELSE_IF, DIVIDE GREATER_OR_EQUALS_THAN |

| | | |
|---|---|---|
| <Parameter> | INTEGER, REAL, BOOLEAN<br>NEGATION, BITWISE_NOT<br>READ_REAL, EPSILON_VALUE<br>BOOLEAN_CAST, IDENTIFIER<br>LEFT_PARENTHESIS<br>PRINTLN, READ_INTEGER<br>PLUS, MINUS, INTEGER_CAST<br>REAL_CAST | RIGHT_PARENTHESIS |
| <ParameterTail> | COMMA, EPSILON_VALUE | RIGHT_PARENTHESIS |
| <FunctionDefinition> | FUNCTION | END, END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <FunctionDefinitionEnd> | END, RETURN | END, END_OF_INSTRUCTION<br>ELSE_IF, ELSE, RETURN |
| <Argument> | EPSILON_VALUE<br>IDENTIFIER | RIGHT_PARENTHESIS |
| <ArgumentTail> | COMMA, EPSILON_VALUE | RIGHT_PARENTHESIS |

# 5 Action Table

Les lignes de "l'action table" représentent les variables et les colonnes représentent les terminaux. Une cellule de cette table contient le numéro d'une règle de production correspondant au numéro repris dans la section grammaire.

Dans le tableau suivant, EPSILON_VALUE $\equiv \epsilon$.

| | EPSILON_VALUE | IDENTIFIER | TYPE_DEFINITION | CONST | END | ASSIGNATION | END_OF_INSTRUCTION | BOOLEAN_TYPE | REAL_TYPE | INTEGER_TYPE | WHILE | FOR | TERNARY_ELSE | COMMA | TERNARY_IF | INTEGER | REAL | BOOLEAN | NEGATION | BITWISE_NOT | MINUS | PLUS | LEFT_PARENTHESIS | RIGHT_PARENTHESIS | LAZY_OR | LAZY_AND | GREATER_THAN | LESS_THAN | GREATER_OR_EQUALS_THAN | LESS_OR_EQUALS_THAN | EQUALITY | INEQUALITY | BITWISE_OR | BITWISE_XOR | ARITHMETIC_SHIFT_LEFT | ARITHMETIC_SHIFT_RIGHT | TIMES | DIVIDE | REMAINDER | BITWISE_AND | INVERSE_DIVIDE | POWER | IF | ELSE_IF | ELSE | LET | BOOLEAN_TYPE | REAL_TYPE | INTEGER_TYPE | READ_REAL | READ_INTEGER | INTEGER_CAST | REAL_CAST | BOOLEAN_CAST | PRINTLN | FUNCTION | RETURN |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <Program> | 1 | 1 | | 1 | | | 1 | | | | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 1 | | | 1 | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| <Instruction> | | 2 | | 3 | | | | | | | 5 | 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 5 | | | 4 | | | | 6 | 6 | 6 | 6 | 6 | 6 | 7 | |
| <InstructionList> | 9 | 8 | | 8 | 9 | | 9 | | | | 8 | 8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 8 | 9 | 9 | 8 | | | | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 9 |
| <InstructionListTail> | 11 | | | | 11 | | 10 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 11 | 11 | | | | | | | | | | | | | 11 |
| <IdentifierInstruction> | | 12 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <IdentifierInstructionTail> | | | 14 | | | 13 | | | | | | | | 13 | | | | | | | | | 15 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <AssignationTail> | | | | | | 16 | | | | | | | | 17 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <ConstDefinition> | | | | 18 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <Block> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 19 | | | | | | | |
| <Loop> | | | | | | | | | | | 21 | 22 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 20 | | | | | | | | | | | | | |
| <ForTail> | | | | | | | 23 | | | | | | 24 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <Type> | | | | | | | | 25 | 26 | 27 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 25 | 26 | 27 | | | | | |
| <Expression> | | 28 | | | | | | | | | | | | | | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 28 | 28 | 28 | 28 | 28 | 28 | | |
| <TernaryIfExpression> | 30 | | | | 30 | | 30 | | | | | | 30 | 30 | 29 | | | | | | | | | 30 | | | | | | | | | | | | | | | | | | | | 30 | 30 | | | | | | | | | | | | | 30 |
| <TernaryElseExpression> | | | | | | | | | | | | | 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <AtomicExpression> | | 32 | | | | | | | | | | | | | | 33 | 34 | 35 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 36 | 36 | 36 | 36 | 36 | 36 | | |
| <AtomicIdentifierExpression> | | 37 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <AtomicIdentifierExpressionTail> | 39 | | | | 39 | | 39 | | | | | | 39 | 39 | 39 | | | | | | 39 | 39 | 38 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | | | 39 | 39 | | | | | | | | | | | | | 39 |
| <UnaryExpression> | | 41 | | | | | | | | | | | | | | 41 | 41 | 41 | 40 | 41 | 41 | 41 | 41 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 41 | 41 | 41 | 41 | 41 | 41 | | |
| <UnaryBitwiseNotExpression> | | 43 | | | | | | | | | | | | | | 43 | 43 | 43 | | 42 | 43 | 43 | 43 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 43 | 43 | 43 | 43 | 43 | 43 | | |
| <UnaryMinusPlusExpression> | | 46 | | | | | | | | | | | | | | 46 | 46 | 46 | | | 44 | 45 | 46 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 46 | 46 | 46 | 46 | 46 | 46 | | |
| <UnaryAtomicExpression> | | 47 | | | | | | | | | | | | | | 47 | 47 | 47 | | | | | 48 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 47 | 47 | 47 | 47 | 47 | 47 | | |
| <BinaryExpression> | | 49 | | | | | | | | | | | | | | 49 | 49 | 49 | 49 | 49 | 49 | 49 | 49 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 49 | 49 | 49 | 49 | 49 | 49 | | |
| <BinaryExpression'> | 51 | | | | 51 | | 51 | | | | | | 51 | 51 | 51 | | | | | | | | | 51 | 50 | | | | | | | | | | | | | | | | | | | | 51 | 51 | | | | | | | | | | | | | 51 |
| <BinaryLazyOrExpression> | | 52 | | | | | | | | | | | | | | 52 | 52 | 52 | 52 | 52 | 52 | 52 | 52 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 52 | 52 | 52 | 52 | 52 | 52 | | |
| <BinaryLazyOrExpression'> | 54 | | | | 54 | | 54 | | | | | | 54 | 54 | 54 | | | | | | | | | 54 | 54 | 53 | | | | | | | | | | | | | | | | | | | | 54 | 54 | | | | | | | | | | | | | 54 |
| <BinaryLazyAndExpression> | | 55 | | | | | | | | | | | | | | 55 | 55 | 55 | 55 | 55 | 55 | 55 | 55 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 55 | 55 | 55 | 55 | 55 | 55 | | |
| <BinaryLazyAndExpression'> | 62 | | | | 62 | | 62 | | | | | | 62 | 62 | 62 | | | | | | | | | 62 | 62 | 62 | 56 | 57 | 58 | 59 | 60 | 61 | | | | | | | | | | | | 62 | 62 | | | | | | | | | | | | | 62 |
| <BinaryNumericExpression> | | 63 | | | | | | | | | | | | | | 63 | 63 | 63 | 63 | 63 | 63 | 63 | 63 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 63 | 63 | 63 | 63 | 63 | 63 | | |
| <BinaryNumericExpression'> | 68 | | | | 68 | | 68 | | | | | | 68 | 68 | 68 | | | | | | 65 | 64 | | 68 | 68 | 68 | 68 | 68 | 68 | 68 | 68 | 68 | 66 | 67 | | | | | | | | | | | 68 | 68 | | | | | | | | | | | | | 68 |
| <BinaryTermExpression> | | 69 | | | | | | | | | | | | | | 69 | 69 | 69 | 69 | 69 | 69 | 69 | 69 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 69 | 69 | 69 | 69 | 69 | 69 | | |
| <BinaryTermExpression'> | 72 | | | | 72 | | 72 | | | | | | 72 | 72 | 72 | | | | | | 72 | 72 | | 72 | 72 | 72 | 72 | 72 | 72 | 72 | 72 | 72 | 72 | 72 | 70 | 71 | | | | | | | | 72 | 72 | | | | | | | | | | | | | 72 |
| <BinaryShiftedExpression> | | 73 | | | | | | | | | | | | | | 73 | 73 | 73 | 73 | 73 | 73 | 73 | 73 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 73 | 73 | 73 | 73 | 73 | 73 | | |
| <BinaryShiftedExpression'> | 79 | | | | 79 | | 79 | | | | | | 79 | 79 | 79 | | | | | | 79 | 79 | | 79 | 79 | 79 | 79 | 79 | 79 | 79 | 79 | 79 | 79 | 79 | 79 | 79 | 74 | 75 | 76 | 77 | 78 | | | 79 | 79 | | | | | | | | | | | | | 79 |
| <BinaryFactorExpression> | | 80 | | | | | | | | | | | | | | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 80 | 80 | 80 | 80 | 80 | 80 | | |
| <BinaryFactorExpression'> | 82 | | | | 82 | | 82 | | | | | | 82 | 82 | 82 | | | | | | 82 | 82 | | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 82 | 81 | | 82 | 82 | | | | | | | | | | | | | 82 |
| <If> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 83 | | | | | | | | | | | | | | |
| <IfEnd> | | | | | 86 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 84 | 85 | | | | | | | | | | | | | |
| <BuiltInFunctionCall> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 87 | 88 | 89 | 90 | 91 | 92 | | |
| <FunctionCallTail> | | | | | | | | | | | | | | | | | | | | | | | 93 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <Parameter> | 95 | 94 | | | | | | | | | | | | | | 94 | 94 | 94 | 94 | 94 | 94 | 94 | 94 | 95 | | | | | | | | | | | | | | | | | | | | | | | | | | | | 94 | 94 | 94 | 94 | 94 | 94 | | |
| <ParameterTail> | 97 | | | | | | | | | | | | | 96 | | | | | | | | | | 97 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <FunctionDefinition> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 98 | |
| <FunctionDefinitionEnd> | | | | | 100 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 99 |
| <Argument> | 102 | 101 | | | | | | | | | | | | | | | | | | | | | | 102 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <ArgumentTail> | 104 | | | | | | | | | | | | | 103 | | | | | | | | | | 104 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

# 6 Génération de code

## 6.1 Introduction

Dans la génération code, la prise en charge du type réel et de la création de fonction n'a pas été faite car non requise.

Pour la génération de code nous somme passé par un parser reccursif ordinaire, ce dernier appel des méthode de la classe *LlvmCodeGenerate* qui génèrent le code correspodant.
*LlvmCodeGenerate* possède un stack global qui permet d'y stocker des expressions, donc des variables, des valeurs entières ou booléennes.
Ce stack est indispenssable car toute instruction combinée doit être divisée en une série d'actions binaires, par exemple :
$a = 1 + 1$
Il faut commencer par evaluer $1 + 1$, sotcker le résultat dans une variable temporaire et puis assigner cette variable temporaire à $a$.
Le stack permet ici de transférer la variable temporaire.

## 6.2 Structures conditionnelles

Le $if\ else$ est déjà implémenté dans LLVM grâce une fonction qui permet de jumper à une ancre ou une autre en fonction de la valeur d'une expression booléenne.
Exemple :

```
def i32 compareTo ( i32 %a, i32 %b){
    entry :
        % cond = icmp slt i32 %a ,%b
        br i1 %cond , label %lower , label % greaterOREquals
    lower :
        ret i32 -1
    greaterOREquals :
        %1 = sub i32 %a ,%b
        ret i32 %1
```

Le $else\ if$ est une cascade de $if\ else$ où le $else$ est une ancre vers le prochaine $if$.

Le if ternaire n'est qu'une autre façons d'écrire un if.

## 6.3 Boucles

La boucle *while* fonctionne en utilisant trois ancres, une pour le test de la condition, une pour le coeur de la boucle et une pour sortir de la boucle.
On commence par tester la condition, si elle est fausse, on saute à la fin de la boucle, si elle est vraie, on saute au coeur de la boucle, une fois celui ci terminé, on saute au test de la condition.

La boucle $for$ est une autre façons d'écrire une boucle *while* en définnisant une variable avec une valeur de base qui permettra de définir la condition de la boucle, celle ci est que cette valeur de base doit être inférieure à un maximum défini par l'utilisateur.
Enfin, après avoir éxecuter le coeur de la boucle il faut incrémenter la valeur de base par un nombre défini lui aussi par l'utilisateur et puis sauter au test de la condition de la boucle.

## 6.4 Fonctionnalités du langages implémentées

Comme mentionné ci-dessus, la grammaire contient tous les élements présents dans le langage, en ce compris les fonctions. De ce fait, le parser est capable de reconnaitre la syntaxe de tous les éléments du langage Iulius.

Pour ce qui est de l'analyse sémantique et la génération de code, seuls deux types de données sont manipulables : les booléens et les entiers. Les fonctions ne sont également pas disponibles pour la génération de code. Le reste des fonctionnalités est quand à lui disponible lors de la génération de code. Ces fonctionnalités sont, par exemple : la déclaration de variables, de constantes (vérification qu'il n'y a

pas d'assignation après l'assignation d'origine), l'assignation (et l'assignation multiple), la conversion, les opérations arithméthiques de base (avec vérification des types de chaque opérande afin de vérifier qu'ils sont compatibles), les opérations bits à bits, les boucles (while et for), les branchements conditionnels (if, elseif, else), l'affichage, la récupération de caractères de l'input, la création de bloc. La gestion du scope des variables est également réalisé.