

UNIVERSITÉ LIBRE DE BRUXELLES

DÉPARTEMENT D'INFORMATIQUE



INFO-F403 - INTRODUCTION TO LANGUAGE THEORY AND
COMPILING

Project Report – Part 2

Author:

Hakim BOULAHYA

Professor:

Gilles GEERAERTS

November 19, 2017

Contents

1	Grammar	2
1.1	Unproductive and unreachable symbols (a)	2
1.2	Priority and associativity of the operators (b)	2
1.2.1	Arithmetic expressions	2
1.2.2	Boolean expressions	3
1.3	Removing left recursion	4
1.4	Factorisation	5
1.5	Transformed Grammar	5
2	LL(1)	5
2.1	First and Follow	5

1 Grammar

1.1 Unproductive and unreachable symbols (a)

In the given grammar, there is no unproductive and/or unreachable symbols.

1.2 Priority and associativity of the operators (b)

Note In this section, P&A refers to priority and associativity of the operators, AE to arithmetic expression and BE to boolean expression.

1.2.1 Arithmetic expressions

Since an arithmetic expression must always be process first before being compared to another one in a boolean expression, we will consider those two separately.

First let's consider the P&A of the arithmetic expressions. We have the following P&A:

-	right
*, /	left
+, -	left

And the following grammar:

$$\begin{aligned}
 \langle ExprArith \rangle &\rightarrow [VarName] \\
 \langle ExprArith \rangle &\rightarrow [Number] \\
 \langle ExprArith \rangle &\rightarrow (\langle ExprArith \rangle) \\
 \langle ExprArith \rangle &\rightarrow - \langle ExprArith \rangle \\
 \langle ExprArith \rangle &\rightarrow \langle ExprArith \rangle \langle Op \rangle \langle ExprArith \rangle \\
 \langle Op \rangle &\rightarrow + \\
 \langle Op \rangle &\rightarrow - \\
 \langle Op \rangle &\rightarrow * \\
 \langle Op \rangle &\rightarrow /
 \end{aligned}$$

As mention in the course page 111, an AE must be a *sum of products*, more specifically in our case a $\{sum, subtraction\}$ of $\{products, division\}$. We will use the same atom definition in the course, with **Number** as the constant rule and **VarName** as the id rule. The minus operator as a right associativity, meaning that it is always linked to the atom next to the operator, so we will set this operator directly as an atom rule.

Same thing goes for the parenthesis. The must be handled without considering the operators outside the parenthesis, so as an atom.

We have the following grammar results:

$$\begin{aligned}
 \langle ExprArith \rangle &\rightarrow \langle ExprArith \rangle \langle SumSubOp \rangle \langle ExprProd \rangle \\
 \langle ExprArith \rangle &\rightarrow \langle ExprProd \rangle \\
 \langle ExprProd \rangle &\rightarrow \langle ExprProd \rangle \langle ProdOp \rangle \langle Atom \rangle \\
 \langle ExprProd \rangle &\rightarrow \langle Atom \rangle \\
 \langle SumSubOp \rangle &\rightarrow +
 \end{aligned}$$

$$\langle SumSubOp \rangle \rightarrow -$$

$$\langle ProdOp \rangle \rightarrow *$$

$$\langle ProdOp \rangle \rightarrow /$$

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

$$\langle Atom \rangle \rightarrow [Number]$$

$$\langle Atom \rangle \rightarrow - \langle Atom \rangle$$

$$\langle Atom \rangle \rightarrow (\langle ExprArith \rangle)$$

1.2.2 Boolean expressions

For boolean expressions we have the following P&A:

not	right
>, <, >=, <=, =, <> /	left
and	left
or	left

And the following grammar:

$$\langle Cond \rangle \rightarrow \langle Cond \rangle \langle BinOp \rangle \langle Cond \rangle$$

$$\langle Cond \rangle \rightarrow not \langle SimpleCond \rangle$$

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

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

$$\langle BinOp \rangle \rightarrow and$$

$$\langle BinOp \rangle \rightarrow or$$

$$\langle Comp \rangle \rightarrow =$$

$$\langle Comp \rangle \rightarrow >=$$

$$\langle Comp \rangle \rightarrow >$$

$$\langle Comp \rangle \rightarrow <=$$

$$\langle Comp \rangle \rightarrow <$$

$$\langle Comp \rangle \rightarrow \diamond$$

Following the same principle as for AE, we have here *disjonction of conjunctions of comparaisons*. By using the same mechanics as above, we have this grammar:

$$\langle Cond \rangle \rightarrow \langle Cond \rangle or \langle ConjCond \rangle$$

$$\langle Cond \rangle \rightarrow \langle ConjCond \rangle$$

$$\langle ConjCond \rangle \rightarrow \langle ConjCond \rangle and \langle AtomCond \rangle$$

$$\langle ConjCond \rangle \rightarrow \langle AtomCond \rangle$$

$$\langle AtomCond \rangle \rightarrow \langle SimpleCond \rangle$$

$$\langle AtomCond \rangle \rightarrow not \langle SimpleCond \rangle$$

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

$$\langle Comp \rangle \rightarrow =$$

$$\langle Comp \rangle \rightarrow >=$$

$\langle Comp \rangle \rightarrow >$
 $\langle Comp \rangle \rightarrow <=$
 $\langle Comp \rangle \rightarrow <$
 $\langle Comp \rangle \rightarrow \diamond$

1.3 Removing left recursion

For the new AE:

$\langle ExprArith \rangle \rightarrow \langle ExprProd \rangle \langle ExprArithPrime \rangle$
 $\langle ExprArithPrime \rangle \rightarrow \langle SumSubOp \rangle \langle ExprProd \rangle \langle ExprArithPrime \rangle$
 $\langle ExprArithPrime \rangle \rightarrow \backslash \epsilon$

$\langle ExprProd \rangle \rightarrow \langle Atom \rangle \langle ExprProdPrime \rangle$
 $\langle ExprProdPrime \rangle \rightarrow \langle ProdOp \rangle \langle Atom \rangle \langle ExprProdPrime \rangle$
 $\langle ExprProdPrime \rangle \rightarrow \backslash \epsilon$

$\langle SumSubOp \rangle \rightarrow +$
 $\langle SumSubOp \rangle \rightarrow -$

$\langle ProdOp \rangle \rightarrow *$
 $\langle ProdOp \rangle \rightarrow /$

$\langle Atom \rangle \rightarrow [VarName]$
 $\langle Atom \rangle \rightarrow [Number]$
 $\langle Atom \rangle \rightarrow - \langle Atom \rangle$
 $\langle Atom \rangle \rightarrow (\langle ExprArith \rangle)$

For the new BE:

$\langle Cond \rangle \rightarrow \langle ConjCond \rangle \langle CondPrime \rangle$

$\langle CondPrime \rangle \rightarrow or \langle ConjCond \rangle \langle CondPrime \rangle$
 $\langle CondPrime \rangle \rightarrow \backslash \epsilon$

$\langle ConjCond \rangle \rightarrow \langle AtomCond \rangle \langle ConjCondPrime \rangle$

$\langle ConjCondPrime \rangle \rightarrow and \langle AtomCond \rangle \langle ConjCondPrime \rangle$
 $\langle ConjCondPrime \rangle \rightarrow \backslash \epsilon$

$\langle AtomCond \rangle \rightarrow \langle SimpleCond \rangle$

$$\begin{aligned}
\langle AtomCond \rangle &\rightarrow not \langle SimpleCond \rangle \\
\langle SimpleCond \rangle &\rightarrow \langle ExprArith \rangle \langle Comp \rangle \langle ExprArith \rangle \\
\langle Comp \rangle &\rightarrow = \\
\langle Comp \rangle &\rightarrow >= \\
\langle Comp \rangle &\rightarrow > \\
\langle Comp \rangle &\rightarrow <= \\
\langle Comp \rangle &\rightarrow < \\
\langle Comp \rangle &\rightarrow \Diamond
\end{aligned}$$

Those are the only rules where left-recursion appears, all the other rules are right-recursion or no recursion at all.

1.4 Factorisation

We can only factorize the following set of rules: $\langle InstList \rangle$, $\langle If \rangle$, $\langle For \rangle$.

We have the following new rules:

$$\begin{aligned}
\langle InstList \rangle &\rightarrow \langle Instruction \rangle \langle InstListSeq \rangle \\
\langle InstListSeq \rangle &\rightarrow ; \langle InstList \rangle \\
\langle InstListSeq \rangle &\rightarrow \backslash \epsilonpsilon \\
\langle If \rangle &\rightarrow if \langle Cond \rangle then \langle Code \rangle \langle IfSeq \rangle \\
\langle IfSeq \rangle &\rightarrow endif \\
\langle IfSeq \rangle &\rightarrow else \langle Code \rangle endif \\
\langle For \rangle &\rightarrow for [VarName] from \langle ExprArith \rangle \langle ForOp \rangle to \langle ExprArith \rangle do \langle Code \rangle done \\
\langle ForOp \rangle &\rightarrow by \langle ExprArith \rangle \\
\langle ForOp \rangle &\rightarrow \backslash \epsilonpsilon
\end{aligned}$$

1.5 Transformed Grammar

2 LL(1)

2.1 First and Follow

[1]	<Program>	→ begin <Code> end
[2]	<Code>	→ ϵ
[3]		→ <InstList>
[4]	<InstList>	→ <Instruction> <InstListSeq>
[5]	<InstListSeq>	→ ϵ
[6]		→ ; <InstList>
[7]	<Instruction>	→ <Assign>
[8]		→ <If>
[9]		→ <While>
[10]		→ <For>
[11]		→ <Print>
[12]		→ <Read>
[13]	<Assign>	→ [VarName] := <ExprArith>
[14]	<ExprArith>	→ <ExprProd> <ExprArithPrime>
[15]	<ExprArithPrime>	→ <SumSubOp> <ExprProd> <ExprArithPrime>
[16]		→ ϵ
[17]	<ExprProd>	→ <Atom> <ExprProdPrime>
[18]	<ExprProdPrime>	→ ϵ
[19]		→ <ProdOp> <Atom> <ExprProdPrime>
[20]	<SumSubOp>	→ +
[21]		→ -
[22]	<ProdOp>	→ *
[23]		→ /
[24]	<Atom>	→ [VarName]
[25]		→ [Number]
[26]		→ - <Atom>
[27]		→ (<ExprArith>)
[28]	<If>	→ if <Cond> then <Code> <IfSeq>
[29]	<IfSeq>	→ endif
[30]		→ else <Code> endif
[31]	<Cond>	→ <ConjCond> <CondPrime>
[32]	<CondPrime>	→ or <ConjCond> <CondPrime>
[33]		→ ϵ
[34]	<ConjCond>	→ <AtomCond> <ConjCondPrime>
[35]	<ConjCondPrime>	→ and <AtomCond> <ConjCondPrime>
[36]		→ ϵ
[37]	<AtomCond>	→ <SimpleCond>
[38]		→ not <SimpleCond>
[39]	<SimpleCond>	→ <ExprArith> <Comp> <ExprArith>
[40]	<Comp>	→ =
[41]		→ >=
[42]		→ >
[43]		→ <=
[44]		→ <
[45]		→ <>
[46]	<While>	→ while <Cond> do <Code> done
[47]	<For>	→ for [VarName] from <ExprArith> <ForOp> to <ExprArith> do <Code> done
[48]	<ForOp>	→ by <ExprArith>
[49]		→ ϵ
[50]	<Print>	→ print([VarName])
[51]	<Read>	→ read([VarName])

Figure 1: Complete transformed CFG