UNIVERSITÉ LIBRE DE BRUXELLES

DÉPARTMENT 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

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 bing compared to another one in a boolean expression, we will consider those two separatly.

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

-	right
*, /	left
+, -	left

And the following grammar:

$$
\begin{array}{l}
<ExprArith> \rightarrow [VarName] \\
<ExprArith> \rightarrow [Number] \\
<ExprArith> \rightarrow (<ExprArith>) \\
<ExprArith> \rightarrow - <ExprArith> \\
<ExprArith> \rightarrow <ExprArith> <Op> <ExprArith> \\
<Op> \rightarrow + \\
<Op> \rightarrow - \\
<Op> \rightarrow * \\
<Op> \rightarrow /
\end{array}
$$

As mention in the course page 111, an AE must be a *sum of products*, more specifically in our case a {*sum, substraction*} *of* {*produts, 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:

$$
\begin{array}{l}
<ExprArith> \rightarrow <ExprArith> <SumSubOp> <ExprProd> \\
<ExprArith> \rightarrow <ExprProd> \\
\\
<ExprProd> \rightarrow <ExprProd> <ProdOp> <Atom> \\
<ExprProd> \rightarrow <Atom> \\
\\
<SumSubOp> \rightarrow +
\end{array}
$$

$<SumSubOp> \rightarrow -$

$<ProdOp> \rightarrow *$
$<ProdOp> \rightarrow /$

$<Atom> \rightarrow [VarName]$
$<Atom> \rightarrow [Number]$
$<Atom> \rightarrow - <Atom>$
$<Atom> \rightarrow (<ExprArith>)$

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:

$<Cond> \rightarrow <Cond> <BinOp> <Cond>$
$<Cond> \rightarrow not <SimpleCond>$
$<Cond> \rightarrow <SimpleCond>$
$<SimpleCond> \rightarrow <ExprArith> <Comp> <ExprArith>$
$<BinOp> \rightarrow and$
$<BinOp> \rightarrow or$
$<Comp> \rightarrow =$
$<Comp> \rightarrow >=$
$<Comp> \rightarrow >$
$<Comp> \rightarrow <=$
$<Comp> \rightarrow <$
$<Comp> \rightarrow <>$

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

$<Cond> \rightarrow <Cond> or <ConjCond>$
$<Cond> \rightarrow <ConjCond>$

$<ConjCond> \rightarrow <ConjCond> and <AtomCond>$
$<ConjCond> \rightarrow <AtomCond>$

$<AtomCond> \rightarrow <SimpleCond>$
$<AtomCond> \rightarrow not <SimpleCond>$

$<SimpleCond> \rightarrow <ExprArith> <Comp> <ExprArith>$

$<Comp> \rightarrow =$
$<Comp> \rightarrow >=$

$<Comp> \rightarrow >$
$<Comp> \rightarrow <=$
$<Comp> \rightarrow <$
$<Comp> \rightarrow <>$

1.3 Removing left recusion

For the new AE:

$<ExprArith> \rightarrow <ExprProd> <ExprArithPrime>$

$<ExprArithPrime> \rightarrow <SumSubOp> <ExprProd> <ExprArithPrime>$
$<ExprArithPrime> \rightarrow \backslash epsilon$

$<ExprProd> \rightarrow <Atom> <ExprProdPrime>$

$<ExprProdPrime> \rightarrow <ProdOp> <Atom> <ExprProdPrime>$
$<ExprProdPrime> \rightarrow \backslash epsilon$

$<SumSubOp> \rightarrow +$
$<SumSubOp> \rightarrow -$

$<ProdOp> \rightarrow *$
$<ProdOp> \rightarrow /$

$<Atom> \rightarrow [VarName]$
$<Atom> \rightarrow [Number]$
$<Atom> \rightarrow - <Atom>$
$<Atom> \rightarrow (<ExprArith>)$

For the new BE:

$<Cond> \rightarrow <ConjCond> <CondPrime>$

$<CondPrime> \rightarrow or <ConjCond> <CondPrime>$
$<CondPrime> \rightarrow \backslash epsilon$

$<ConjCond> \rightarrow <AtomCond> <ConjCondPrime>$

$<ConjCondPrime> \rightarrow and <AtomCond> <ConjCondPrime>$
$<ConjCondPrime> \rightarrow \backslash epsilon$

$<AtomCond> \rightarrow <SimpleCond>$

$<AtomCond> \rightarrow not <SimpleCond>$

$<SimpleCond> \rightarrow <ExprArith> <Comp> <ExprArith>$

$<Comp> \rightarrow =$
$<Comp> \rightarrow >=$
$<Comp> \rightarrow >$
$<Comp> \rightarrow <=$
$<Comp> \rightarrow <$
$<Comp> \rightarrow <>$

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

1.4 Factorisation

We can only factorize the following set of rules: $< InstList >$, $< If >$, $< For >$.

We have the following new rules:

$<InstList> \rightarrow <Instruction> <InstListSeq>$
$<InstListSeq> \rightarrow ; <InstList>$
$<InstListSeq> \rightarrow \backslash epsilon$

$<If> \rightarrow if <Cond> then <Code> <IfSeq>$
$<IfSeq> \rightarrow endif$
$<IfSeq> \rightarrow else <Code> endif$

$<For> \rightarrow for\ [VarName]\ from <ExprArith> <ForOp> to <ExprArith> do <Code> done$

$<ForOp> \rightarrow by <ExprArith>$
$<ForOp> \rightarrow \backslash epsilon$

1.5 Transformed Grammar

2 LL(1)

2.1 First and Follow

[1]	<Program>	→ begin <Code> end
[2]	<Code>	→ <InstList>
[3]		→ ϵ
[4]	<InstList>	→ <Instruction> <InstListSeq>
[5]	<InstListSeq>	→ ; <InstList>
[6]		→ ϵ
[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>	→ <ProdOp> <Atom> <ExprProdPrime>
[19]		→ ϵ
[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: Transformed Grammar