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 27, 2017

Contents

1 Grammar

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

1.1 Priority and associativity of the operators

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

An AE must always be *processed* before being compared to another AE in a BE. Logically, we will consider first to unambiguous the arithmetic expressions and then the boolean expressions.

1.1.1 Arithmetic expressions

Let's consider the P&A of arithmetic expressions in Figure 1 and the initial grammar in Figure 2.

Operators	Associativity
-	right
*, /	left
+, -	left

Figure 1: Priority and associativity of AE

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

Figure 2: Initial grammar of AE

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 as in the course, with `[Number]` as the constant rule and `[VarName]` as the id rule. The minus operator as a right associativity means that it is always linked to the atom next to it, so we will set this operator directly as an atom rule. Same thing goes for the parenthesis, it should be handled without considering the operators outside the parenthesis, so as an atom. Figure 3 show the unambiguous grammar of AE.

<ExprArith> → <ExprArith> <SumSubOp> <ExprProd>
→ <ExprProd>
<ExprProd> → <ExprProd> <ProdOp> <Atom>
→ <Atom>
<SumSubOp> → +
→ -
<ProdOp> → *
→ /
<Atom> → [VarName]
→ [Number]
→ - <Atom>
→ (<ExprArith>)

Figure 3: Unambiguous grammar of AE

1.1.2 Boolean expressions

Let's consider the P&A of boolean expressions in Figure 4 and the initial grammar in Figure 5.

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

Figure 4: Priority and associativity of BE

<Cond> → <Cond> <BinOp> <Cond>
→ `not` <SimpleCond>
→ <SimpleCond>
<SimpleCond> → <ExprArith> <Comp> <ExprArith>
<BinOp> → `and`
→ `or`
<Comp> → =
→ >=
→ >
→ <=
→ <
→ <>

Figure 5: Initial grammar of BE

Following the same principle as for AE, we have here *disjonction of conjonctions*. Figure 6 shows the unambiguous grammar of BE.

<Cond>	→ <Cond> `or` <ConjCond>
	→ <ConjCond>
<ConjCond>	→ <ConjCond> `and` <AtomCond>
	→ <AtomCond>
<AtomCond>	→ <SimpleCond>
	→ `not` <SimpleCond>
<SimpleCond>	→ <ExprArith> <Comp> <ExprArith>
<Comp>	→ =
	→ >=
	→ >
	→ <=
	→ <
	→ <>

Figure 6: Unambiguous grammar of BE

1.2 Removing left recusion

The only rules where left-recursion appears with <`ExprArith`>, <`ExprProd`>, <`Cond`> and <`ConjCond`> variables that are in the unambiguous grammar. We have to remove the left-recursion for the unambiguous grammar in Figure 3 and 6. The transformed grammars are shown in Figure 7 and 8. To enhance readability of new introduced rules, we used the word *Prime* instead of the character itself.

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

<ExprArith>	→ <ExprProd> <ExprArithPrime>
<ExprArithPrime>	→ <SumSubOp> <ExprProd> <ExprArithPrime>
	→ ϵ
<ExprProd>	→ <Atom> <ExprProdPrime>
<ExprProdPrime>	→ <ProdOp> <Atom> <ExprProdPrime>
	→ ϵ
<SumSubOp>	→ +
	→ −
<ProdOp>	→ ∗
	→ /
<Atom>	→ [VarName]
	→ [Number]
	→ - <Atom>
	→ (<ExprArith>)

Figure 7: Left-recursion removed for AE grammar

<Cond>	→ <ConjCond> <CondPrime>
<CondPrime>	→ `or` <ConjCond> <CondPrime>
	→ ϵ
<ConjCond>	→ <AtomCond> <ConjCondPrime>
<ConjCondPrime>	→ and <AtomCond> <ConjCondPrime>
	→ ϵ
<AtomCond>	→ <SimpleCond>
	→ `not` <SimpleCond>
<SimpleCond>	→ <ExprArith> <Comp> <ExprArith>
<Comp>	→ =
	→ >=
	→ >
	→ <=
	→ <
	→ <>

Figure 8: Left-recursion removed for BE grammar

1.3 Factorisation

We can only factorize the following variables: <`InstList`>, <`If`> and <`For`>. Figure 9 shows the factorized rules.

<InstList>	→ <Instruction> <InstListSeq>
<InstListSeq>	→ ; <InstList>
	→ ϵ
<If>	→ `if` <Cond> `then` <Code> <IfSeq>
<IfSeq>	→ `endif`
	→ `else` <Code> `endif`
<For>	→ `for` [VarName] `from` <ExprArith> <ForOp> `to` <ExprArith> `do` <Code> `done`
<ForOp>	→ `by` <ExprArith>
	→ ϵ

Figure 9: Factorized rules

1.4 Transformed Grammar

The complete transformed grammar is shown in Figure 10.

2 LL(1) parser

2.1 First$^1(\cdot)$ and Follow$^1(\cdot)$

To be able to generate the action table of our LL(1) parser, it is necessary to show the results of the First$^1(\cdot)$ and Follow$^1(\cdot)$ computation of each variables belonging to the grammar. The results are shown in Figure 11.

[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 10: Transformed Grammar

Variables	**First**$^1(\cdot)$	**Follow**$^1(\cdot)$
Program	begin	
Code	[VarName] if while for print read ϵ	end else endif done
InstList	[VarName] if while for print read	end else endif done
InstListSeq	; ϵ	end else endif done
Instruction	[VarName] if while for print read	; end else endif done
Assign	[VarName]	; end else endif done
ExprArith	[VarName] [Number] - (	; end else endif done) = >= > <= < <> and or then do by to
ExprArithPrime	+ - ϵ	Follow^1(<ExprArith>)
ExprProd	[VarName] [Number] - (	+ - Follow^1(<ExprArith>)
ExprProdPrime	* / ϵ	+ - Follow^1(<ExprArith>)
SumSubOp	+ -	[VarName] [Number] - (
ProdOp	* /	[VarName] [Number] - (
If	if	; end else endif done
Atom	[VarName] [Number] - (	* / + - Follow^1(<ExprArith>)
IfSeq	endif else	; end else endif done
Cond	[VarName] [Number] - (not	then do
CondPrime	or ϵ	then do
ConjCond	[VarName] [Number] - (not	or then do
ConjCondPrime	and ϵ	or then do
AtomCond	[VarName] [Number] - (not	and or then do
SimpleCond	[VarName] [Number] - (	and or then do
Comp	= >= > <= < <>	[VarName] [Number] - (
While	while	; end else endif done
For	for	; end else endif done
ForOp	by ϵ	to
Print	print	; end else endif done
Read	read	; end else endif done

Figure 11: First^1 and Follow^1 results

2.2 Action table

The action table is available as a comma separated `.csv` file in **doc/actionTable.csv**. This file only contains the *produce* rules. It is not necessary to include the table that matches terminals together since the only results possible is the action *match* for terminals that are the same. In the case of the `end` terminal, the action is *accept*. An empty cell in the table induce a syntax error. Section **??** discuss in more details the implementation of the action table.

3 Implementation

3.0.1 Scanner improvements

Read input In the first part of the project we were asked to implement a scanner. To be able to use the scanner within the parser, we modified some part of the scanner. The scanner is implemented in the abstract class `Scanner`. The lexer generated class is named `GeneratedScanner` and will implement the `Scanner`. To be able to read the input, *i.e.* the tokens, the method `Scanner.scan()` must be called, and will return a list of `Symbol`. It is still possible to run the scanner as a standalone application, see section 4. Figure 12 shows an example of use of the scanner.

```
Scanner scanner = new GeneratedScanner(source);
List<Symbol> symbols = scanner.scan();
```

Figure 12: Call the scanner to read the tokens from a source file

Lexer improvements Two improvements were made in the lexer. (1) We modified the any ASCII characters regex `.` (dot) to regex that matches the complement of the empty set `[^]`. (2) Instead of printing `"Unkown symbol"` and continue the scan when a unrecognize symbol is detected, the scanner throw a `UnknownTokenException` which logically stops the scan.

4 HowTo