UNIVERSITÉ LIBRE DE BRUXELLES

DÉPARTMENT D'INFORMATIQUE

INFO-F403
INTRODUCTION TO LANGUAGE THEORY AND COMPILING

Project Report – Part 2

Author:
Hakim BOULAHYA

November 27, 2017

Contents

1 Introduction **2**

2 Grammar transformation **2**
2.1 Priority and associativity of the operators . . . 2
2.1.1 Arithmetic expressions . . . 2
2.1.2 Boolean expressions . . . 3
2.2 Removing left recusion . . . 4
2.3 Factorisation . . . 5
2.4 Transformed Grammar . . . 5

3 LL(1) parser **5**
3.1 $\text{First}^1(\cdot)$ and $\text{Follow}^1(\cdot)$ . . . 5
3.2 Action table . . . 8

4 Scanner improvements **8**

5 LL(1) Parser implementation **8**
5.1 Grammar and action table files . . . 8
5.2 Complete action table . . . 9
5.3 Explicit Stack . . . 9

6 Project details **11**
6.1 Project files . . . 11
6.2 How to run . . . 11

7 Conclusion **12**

1 Introduction

As part of the course of *Introduction to Language theory and Compiling* we were asked to write a compiler of the IMP language. The goal of the second part of the project was to implement the *syntactic analysis* or *parsing*. As it was done for the first part, the parser will be implemented in `Java`. It will use the scanner, that was implemented in the part 1 of the project, to read the input. This part consits in transforming a given grammar to obtain a LL(1) grammar, and uses this newly transformed grammar to implement a parser for the IMP compiler.

2 Grammar transformation

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

2.1 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.

2.1.1 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

2.1.2 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

2.2 Removing left recusion

The only rules where left-recursion appears are <`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 punctuation mark.

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

2.3 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

2.4 Transformed Grammar

The complete transformed grammar is shown in Figure 10.

3 LL(1) parser

3.1 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

3.2 Action table

Let $G < V, T, P, S >$ our transformed grammar. Let the action table be $M(v, t)$ where $v \in V$, and $t \in T$.

M is filled in as follow:

- $\forall v \in V$ and $\forall t \in \text{First}^1(v)$, then set $M(v, t)$ as the rule number where t is first.
- $\forall v \in V$ if $\epsilon \in \text{First}^1(v)$ then $\forall t \in \text{Follow}^1(v)$ set $M(v, t)$ as the rule number where $v \to \epsilon$.
- All others cell should remain empty.

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 5.2 discuss in more details the implementation of the action table.

4 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 6.2. Figure 12 shows an example of use of the scanner.

```
FileReader source = ImpCompilo.file("file.imp");
Scanner scanner = new GeneratedScanner(source);
List<Symbol> symbols = scanner.scan();
```

Figure 12: Scanner call 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 throws a `UnknownTokenException` which logically stops the scan.

5 LL(1) Parser implementation

5.1 Grammar and action table files

Location of files The grammar and the action table `csv` files are stored as resources in **more/src/resources** directory. [1].

[1] The csv file are *comma separated*

Action table parsing To be able to run the parser it is required to be able to access the action table to implement the recursive descent LL(1) parser. As mention in section 3.2, the action table is stored in a `csv` file **actionTable.csv**. The first line of this file contains the terminals, and all the other lines contains in the first column a variable and the following columns either the rules to apply for each terminal. If there is no rule *i.e.* the cell is empty.

The method `LL1Parser.buildActionTable()` will read the file to create a map that matches a variable and a terminal to a rule number *i.e.* a `Map<String, Map<String, Integer>>`.

Grammar parsing It is also necessary to access to set of rules of the grammar within the parser. The grammar is stored in a `csv` file **grammar.csv**. The first line of the file is the set of variables, and the second line the set of terminals. All the following lines are the rules. Since it is a *Context-Free Grammar*, we know that the first element of a rule is the left-hand side, and all the following element are the the right-hand side. The order of the rules is important since they will be numeroted in the parsing order, and will be used to print the *left-most derivation*. The grammar in *grammar.csv* is the same as shown in Figure 10.

The method `LL1Parser.buildGrammar()` will read the file to create a map that matches a rule number to a list of {*terminals, variables*} *i.e.* `Map<Integer, List<String>>`. A list could have been used, but a map was prefered since the first rule is numeroted as 1.

The content of the list of a rule is the following: the first element is the left-hand side, and all the other elements are the right-hand side. Note that when the right-hand side is ϵ, the list of element is just the left-hand side, since there will be nothing to push on the stack for such rules (see section 5.3).

This method of implementation of the grammar and the action table is very interesting, since it allows to use the LL(1) parser for any LL(1) grammar, by just changing the `csv` files.

5.2 Complete action table

The action table of an LL(1) parser in addition to the rule number of the *produce* action that is provided in the `csv` file must also be able to provide the *match* and *accept* actions. It is also necessary to provide a way to return a *syntax error* if none of these actions are possible. The method `LL1Parser.M(a, l)` [2] will return an `Integer` value, based on the parameters `a` and `l`, that is corresponding to either:

- A rule number from the `csv` file if $a \in V \wedge l \in T$. If cell is empty in the file, then it returns the constant `SYNTAX_ERROR`
- The constant `MATCH` if $(a, l \in T) \wedge (a = l)$
- The constant `ACCEPT` if $(a = l) \wedge (a = \texttt{end})$
- The constant `SYNTAX_ERROR` for all other cases

5.3 Explicit Stack

PDA The implementation of the LL(1) parser is done by using a pushdown automaton. We can actually avoid implementing any automaton states and only use an explicit stack, since the LL(1) parse is a PDA with a single state [3].

[2] The function name is taken from the Lecture Notes (p135)

[3] Mention in the Lecture Notes page 135

Recursive descent The stack only solution is a possible implementation of a recursive descent parser. Indeed, by using recursion it uses the *call stack* of subroutines. The parser doesn't use the *call stack* but an explicit stack.

Implementation The parsing of the input is implemented in the method `LL1Parser.parse()`. This method uses the stack and the scanned tokens to run the parsing. The scanner throws a `UnknownTokenException` if a unexpected token is detected. The method also initialize the stack with the start symbol. The parsing consists of a loop that runs an action from the action table *i.e.* the results of the method M, based on the current top of stack and the current parsed token. The loop stops only in two ways: (1) if the input is accepted *i.e.* the action *accept* is ran; (2) a *syntax error* is returned from the action table. The stack is implemented using the built-in class `Stack` of `Java`. Each action performs specific changes either on the stack, on the current token or on the parsing process:

- *accept*: stops the parsing
- *match*: pop the stack and read next token
- *produce*: replace the top of stack by the right-hand side of the rule from the action table
- *syntax error*: throws a `SyntaxError`, with details about the expected tokens from the action table

The implementation of the `parse` method is in Figure 13 and the actions in Figure 14.

```
stack.push(getStartSymbol());
scannedTokens = scanner.scan();
nextToken();
parsing = token != null;
while (parsing) {
    Integer ruleNumber = M(stack.peek(),
        lexicalUnitOf(token));
    switch (ruleNumber) {
        case ACCEPT:
            accept(token);
            break;
        case MATCH:
            match(token);
            break;
        case SYNTAX_ERROR:
            syntaxError(token);
            break;
        default:
            produce(token, ruleNumber);
            break;
    }
}
```

Figure 13: `LL1Parser.parse()` method

```
parsing = false;
```

(a) *accept*

```
stack.pop();
nextToken();
```

(b) *match*

```
stack.pop();
pushRule(ruleNumber);
```

(c) *produce*

```
throw new
    SyntaxError(message);
```

(d) *syntax error*

Figure 14: Actions method

6 Project details

6.1 Project files

Files Java source files are in **more/src** directory. The implementation of our compiler is divided in 4 parts:

1. Files that composes the scanner: `Scanner.java` abstract class and the `LexicalAnalyzer.lex` that is used to generate the `GeneratedScanner.java` class, and the provided `Symbol.java` and `LexicalUnit.java` classes.
2. Files that composes the parser: `LL1Parser.java` and the `TreeNode.java`, that is used to generate and print the derivation tree, classes.
3. Utilities methods in `ImpCompilo.java` classes used by the scanner and the parser to read files, log informations or exit safely the compiler.
4. A set of exceptions to be thrown by the scanner/parser.

Javadoc The JavaDoc is available in **doc/javadoc**. The documentation for `Scanner.java` is **Scanner.html**, this document the scanner of part 1, including the improvements made in section 4. The documentation for the parser `LL1parser.java` is in **LL1Parser.html**

6.2 How to run

- To run the parser use the following command:

  ```
  java -jar dist/part2.jar file.imp
  ```

  This command will output the *left-most derivation*.

- It is also possible to output the *derivation tree*:

  ```
  java -jar dist/part2.jar file.imp printTree
  ```

- It is still possible to run the scanner as a standalone appliction:

  ```
  java -jar dist/scanner.jar file.imp
  ```

7 Conclusion