Introduction to language theory and compiling

INFO-F-403

Project : Part 2

16 November 2020

BAKKALI Yahya : 000445166
HAUWAERT Maxime : 000461714

Université Libre de Bruxelles (ULB)

Contents

1 Introduction

The objective of this project is to design a compiler for a new language Fortr-S. The first component of a compiler, the lexical analyser, has already been implemented in the first part of this project. In this second part, the second component, the syntax analyser, will be implemented.

2 Description

As explained in the first part of the project, some languages are not regular. For example, the "well-commented" language $L_{/**/}$ in this case. Due to the length of words in $L_{/**/}$ is not delimited, the use of an unbounded counter is necessary. Unfortunately, such an unbounded counter cannot be encoded in the finite automata structure. To solve this problem, it has been chosen to use grammars, which are a much more powerful formalism for language specification to handle a larger number of languages than finite automata. The grammar can be recognized if an LL(k) (Left-to-right, Leftmost derivation) parser can be constructed from it. For this part of project, the LL(1) parser will be implemented.

A LL(k) parser is a top-down parser made for context-free languages. It reads the list of tokens from left to right and performs the left most derivation by using k numbers of tokens of look-ahead.

3 Grammar

A grammar is a quadruplet $G =< V, T, P, S >$ where

- V is a finite set of *variables*
- T is a finite set of *terminals*
- P is a finite set of *production rules* of the form $\alpha \rightarrow \beta$ with :
  - $\alpha \in (V \cup T)^* V (V \cup T)^*$
  - $\beta \in (V \cup T)^*$
- $S \in V$ is a variable called the *start symbol*

In project grammar the *variables* and the *terminals* are represented by the following regex :

- *variables* : $< [A - Za - z]^+ >$
- *terminals* : $[A - Za - z]^+$ | \[$[A - Za - z]^+$ \] | := | [+-*/=()>$]

Here is the grammar that was given in the statement of this project :

Fortr-S Grammar			
1	<Program>	→	BEGINPROG [ProgName] [EndLine] <Code> ENDPROG
2	<Code>	→	<Instruction> [EndLine] <Code>
3		→	ϵ
4	<Instruction>	→	<Assign>
5		→	<If>
6		→	<While>
7		→	<Print>
8		→	<Read>
9	<Assign>	→	[VarName]:=<ExprArith>
10	<ExprArith>	→	[VarName]
11		→	[Number]
12		→	(<ExprArith>)
13		→	-<ExprArith>
14		→	<ExprArith> <Op> <ExprArith>
15	<Op>	→	+
16		→	-
17		→	*
18		→	/
19	<If>	→	IF (<Cond>) THEN [EndLine] <Code> ENDIF
20		→	IF (<Cond>) THEN [EndLine] <Code> ELSE [EndLine] <Code> ENDIF
21	<Cond>	→	<ExprArith> <Comp> <ExprArith>
22	<Comp>	→	=
23		→	>
24	<While>	→	WHILE (<Cond>) DO[EndLine] <Code>ENDWHILE
25	<Print>	→	PRINT([VarName])
26	<Read>	→	READ([VarName])

But some rules had to change in order to make the grammar LL(1) such as factoring, removing left-recursion, removing useless symbols and taking into account the priority and the associativity of the operators.

3.1 Factoring

In a grammar, if a set of rules are in the following forms:

$$\begin{gathered} V \to \alpha\beta_1 \\ V \to \alpha\beta_2 \\ \vdots \\ V \to \alpha\beta_n \end{gathered}$$

They can be replace by :

$$\begin{gathered} V \to \alpha V' \\ V' \to \beta_1 \\ V' \to \beta_2 \\ \vdots \\ V' \to \beta_n \end{gathered}$$

In this grammar this set of two rules :

<If>	→	IF (<Cond>) THEN [EndLine] <Code> ENDIF
	→	IF (<Cond>) THEN [EndLine] <Code> ELSE [EndLine] <Code> ENDIF

can be factored into :

<If>	→	IF (<Cond>) THEN [EndLine] <Code> <IfTail>
<IfTail>	→	ELSE [EndLine] <Code> ENDIF
	→	ENDIF

3.2 Priority and the associativity

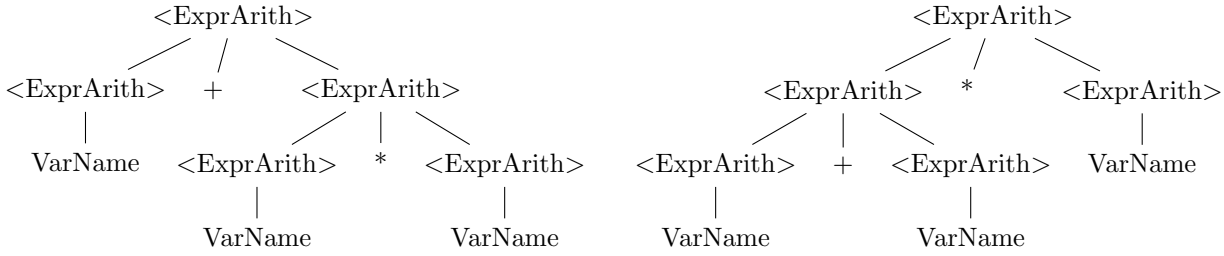
Here are all the rule about arithmetic operators of the current grammar :

<ExprArith>	→	[VarName]
	→	[Number]
	→	(<ExprArith>)
	→	-<ExprArith>
	→	<ExprArith> <Op> <ExprArith>
<Op>	→	+
	→	-
	→	*
	→	/

Here is the list of the operators and their respective associativity sorted in decreasing order of priority :

Operator	**Associativity**
- (unary)	right
*, /	left
+, - (binary)	left

The rules listed above are clearly ambiguous as for example the word VarName + VarName * VarName can produce two different trees :



The tree on the left is obviously the correct one so the grammar needs to be modified in order to accept this and only this tree.
To respect the priority of the + and * operators it is needed that an expression is a sum of product of atoms where an atom is either a VarName or a Number.
For example [Number] * [Number] + [Number] * [Number] should give the sum of the two product [Number] * [Number].

Here are the rules that have been added :

<Expr>	→	<Expr> + <Prod>
	→	<Prod>
<Prod>	→	<Prod> * <Atom>
	→	<Atom>
<Atom>	→	[Number]
	→	[VarName]

The only things left to consider are the unary minus, the parenthesis and the two operator - and /.
An expression like -[Number] + [Number] should be considered as (-[Number]) + [Number] and not as -([Number] + [Number]). So the unary minus should be added in the definition of an atom.
Similarly the parenthesis are also added in the definition of an atom.

Here are the rules that have been added :

<Expr>	→	<Expr> - <Prod>
<Prod>	→	<Prod> / <Atom>
<Atom>	→	- <Atom>
	→	(<Expr>)

The resulting grammar after removing all theses ambiguities :

<Expr>	→	<Expr> + <Prod>
	→	<Expr> - <Prod>
	→	<Prod>
<Prod>	→	<Prod> * <Atom>
	→	<Prod> / <Atom>
	→	<Atom>
<Atom>	→	- <Atom>
	→	[Number]
	→	[VarName]
	→	(<Expr>)

3.3 Removing left-recursion

A recursion refers to the occurrence if the left-hand side of a rule in its right-hand side. A left-recursion is when this recursive variable occurs as the first symbol of the right-hand side. The recursion can be direct or indirect as the following table shows :

Direct			Indirect		
S	→	Sa	S	→	Aa
S	→	ϵ	A	→	Sb
			A	→	ϵ

In order to remove the left-recursion, all the indirect left-recursion have to be transformed into direct left-recursion and then to transform these left-recursion into right-recursion.
Here are all the left-recursions of the current grammar :

<Expr>	→	<Expr> + <Prod>
	→	<Expr> - <Prod>
	→	<Prod>
<Prod>	→	<Prod> * <Atom>
	→	<Prod> / <Atom>
	→	<Atom>

And after removing the left-recursions :

<Expr>	→	<Prod> <Expr'>
<Expr'>	→	+<Prod> <Expr'>
	→	- <Prod> <Expr'>
	→	ϵ
<Prod>	→	<Atom> <Prod'>
<Prod'>	→	* <Atom> <Prod'>
	→	/ <Atom> <Prod'>
	→	ϵ

3.4 Removing useless symbols

A variable A is unproductive in a grammar $G =< V, T, P, S >$ iff there is no word $w \in T^*$ s.t. $A \rightarrow_G^* w$.

A symbol $X \in V \cup T$ is unreachable in a grammar $G =< V, T, P, S >$ iff there is no sentential form of G that contains an X, i.e. there is no derivation of the form $S \rightarrow_G^* a_1 X a_2$.

No useless symbol has been detected in the grammar.

3.5 Hypotheses

Two rules have been added to easily handle multiple end lines following each other.

<MultiEndLines>	→	[Endline] <MultiEndLines>
	→	ϵ

The first rule of <Program> rule was modified because the code (not the variable) can start with 0 or more end line and can end with 0 or more end line and the first rule of <Code> was modified because there can be end lines between the instructions.

<Program>	→	<MultiEndLines> BEGINPROG [ProgName] [EndLine] <Code> ENDPROG <MultiEndLines> $
<Code>	→	<MultiEndLines> <Instruction> [EndLine] <Code>

The $ represents the end of the tokens. It is used to tell the program that the end of the file is reached.

3.6 Modified grammar

Here is the grammar after performing all the modifications defined above :

Fortr-S Grammar			
1	<Program>	→	<MultiEndLines> BEGINPROG [ProgName] [EndLine] <Code> ENDPROG <MultiEndLines> $
2	<Code>	→	<Instruction> [EndLine] <Code>
3		→	[EndLine] <Code>
4		→	ϵ
5	<Instruction>	→	<Assign>
6		→	<If>
7		→	<While>
8		→	<Print>
9		→	<Read>
10	<Assign>	→	[VarName]:=<Expr>
11	<Expr>	→	<Prod> <Expr'>
12	<Expr'>	→	+<Prod> <Expr'>
13		→	- <Prod> <Expr'>
14		→	ϵ
15	<Prod>	→	<Atom> <Prod'>
16	<Prod'>	→	* <Atom> <Prod'>
17		→	/ <Atom> <Prod'>
18		→	ϵ
19	<Atom>	→	- <Atom>
20		→	[Number]
21		→	[VarName]
22		→	(<Expr>)
23	<If>	→	IF (<Cond>) THEN [EndLine] <Code> <IfTail>
24	<IfTail>	→	ELSE [EndLine] <Code> ENDIF
25		→	ENDIF
26	<Cond>	→	<Expr> <Comp> <Expr>
27	<Comp>	→	=
28		→	>
29	<While>	→	WHILE (<Cond>) DO[EndLine] <Code>ENDWHILE
30	<Print>	→	PRINT([VarName])
31	<Read>	→	READ([VarName])
32	<MultiEndLines>	→	[Endline] <MultiEndLines>
33		→	ϵ

4 First and Follow

4.1 First algorithm

$First(X)$: it is the set of terminals which can start a string generated from X.
The first set can be found by following the alogrithm below:

Algorithm 1 First

for all $a \in T$ **do**
  $First^k(a) \leftarrow \{a\}$
end for
for all $A \in V$ **do**
  $First^k(A) \leftarrow \{\emptyset\}$
end for
repeat
  for all $A \in V$ **do**
    $First^k(A) \leftarrow First^k(A) \cup \{x \in T^* | A \rightarrow Y_1 Y_2 ... Y_n \wedge x \in$
    $First^k(Y_1) \oplus^k First^k(Y_2) \oplus^k ... \oplus^k First^k(Y_n)\}$
  end for
until stability

4.2 Follow algorithm

$Follow(X)$: it is the set of terminals which can follow a string generated from X.
The follow set can be found by following the algorithm below:

Algorithm 2 Follow

for all $A \in V \setminus \{S\}$ **do**
  $Follow^k(A) \leftarrow \{\emptyset\}$
end for
$Follow^k(S) \leftarrow \{\epsilon\}$
repeat
  if $B \rightarrow \alpha A \beta \in P$ $(with\ B \in V\ and\ \alpha, \beta \in (V \cup T)^*)$ **then**
    $Follow^k(A) \leftarrow Follow^k(A) \cup \{First^k(\beta) \oplus^k Follow^k(B)\}$
  end if
until stability

Here is the First and Follow table at the first step :

Symbol	First	Follow
<Program>	first(MultiEndLines)	
<Code>	first(<Instruction>), [EndLine], ϵ	ENDPROG, ENDWHILE, first(<IfTail>)
<Instruction>	first(<Assign>), first(<If>), first(<While>), first(<Print>), first(<Read>)	[EndLine]
<Assign>	[VarName]	follow(<Instruction>)
<Expr>	first(<Prod>)	), first(<Comp>), follow(<Cond>), follow(<Assign>)
<Expr'>	+, -, ϵ	follow(<Expr>)
<Prod>	first(<Atom>)	first(<Expr'>)
<Prod'>	*, /, ϵ	follow(<Prod>)
<Atom>	[VarName], [Number], -, (	first(<Prod'>)
<If>	IF	follow(<Instruction>)
<IfTail>	ELSE, ENDIF	follow(<If>)
<Cond>	first(<Expr>)	)
<Comp>	=, >	first(<Expr>)
<While>	WHILE	follow(<Instruction>)
<Print>	PRINT	follow(<Instruction>)
<Read>	READ	follow(<Instruction>)
<MultiEndLines>	[EndLine], ϵ	BEGINPROG, $

Here is the First and Follow table after finishing all the calculations :

Symbol	First	Follow
<Program>	[EndLine], BEGINPROG	
<Code>	[EndLine], [VarName], IF, WHILE, PRINT, READ, ϵ	ENDPROG, ENDWHILE, ENDIF, ELSE
<Instruction>	[VarName], IF, WHILE, PRINT, READ	[EndLine]
<Assign>	[VarName]	[EndLine]
<Expr>	[VarName], [Number], -, (	[EndLine], =, >,)
<Expr'>	+, -, ϵ	[EndLine], =, >,)
<Prod>	[VarName], [Number], -, (	[EndLine], +, -, =, >,)
<Prod'>	*, /, ϵ	[EndLine], +, -, =, >,)
<Atom>	[VarName], [Number], -, (	[EndLine], +, -, *, /, =, >,)
<If>	IF	[EndLine]
<IfTail>	ELSE, ENDIF	[EndLine]
<Cond>	[VarName], [Number], -, (	)
<Comp>	=, >	[VarName], [Number], -, (
<While>	WHILE	[EndLine]
<Print>	PRINT	[EndLine]
<Read>	READ	[EndLine]
<MultiEndLines>	[EndLine], ϵ	BEGINPROG, $

5 Action Table

It gives the rule to produce for each possible symbol in the top of the stack, and each possible first character on the input.

Algorithm 3 Action table

$M \leftarrow \times$
for all $A \rightarrow \alpha$ **do**
  for all $A \in First^1(\alpha)$ **do**
    $M[A, a] \leftarrow M[A, a] \cup Produce(A \rightarrow \alpha)$
  end for
  if $\epsilon \in First^1(\alpha)$ **then**
    for all $A \in Follow^1(A)$ **do**
      $M[A, a] \leftarrow M[A, a] \cup Produce(A \rightarrow \alpha)$
    end for
  end if
end for
for all $a \in T$ **do**
  $M[a, a] \leftarrow Match$
end for
$M[\$, \epsilon] \leftarrow Accept$

The action table is at the appendix A.

6 Implementation

The lexical analyser implementation of the first part of the project was reused. So in this section only the additions and the modifications will be shown.

6.1 Command line

Two options were added :

- -v : Indicates if the user wants a more verbose output. If it present the program prints the produce transitions otherwise it prints only the sequence of the numbers of the rules that were used.
- -wt : Indicates the latex file to write the parse tree in.

6.2 Packages

A package `compiler` and a sub-package `exceptions` were added.

The `compiler` package is composed of all the classes except the exceptions which have been put in the sub-package `exceptions`.

Only the `Main` file is in the root of the `src` directory.

6.3 Classes

The `LexicalAnalyser` class was renamed to `Scanner` and the method `lexicalAnalyse` was moved to the `Scanner` and renamed to `scan` to have more coherence. (Everything done via the flex file)

The method `compile` of the `Compiler` class takes another argument, a list of options corresponding to the new options of the command line.

A new constructor of the `Symbol` class has been added and takes only a value (type Object) in order to form a non-terminal symbol and a new method `toTexString` was added, this method return the name of the symbol, if it is a non-terminal it puts its name between < > and if it is a [Number] or a [Varname] it adds its value.

The `CommandLineParser` class was added to be able to easily extract the different options, as `Option` class instances, given in the command line. To handle errors coming from this class the new exception `CommandLineException` was added.

The `Option` class is used to represent an option with 0 or 1 argument.

The `Parser` class was added and it has an important method `parse()`. This method launch the parsing process and builds the left most derivation as well as the parse tree. It has a stack with the first element being the first token. For each variable the rule to produce is determined by either a peek on the top of the stack if there are multiples rules or directly if there is only one rule. Then the method `match` pop the top of the stack and verify if its type is the same as the expected one.

The `ParseTree` class represents parse trees, each node has a symbol and a list of children of type `ParseTree` and was taken as it was given. The only little modifications were in the iterations of the children, as some children can be `null`, it is important not to iterate on them.

6.4 Program manual

NAME
part2.jar – A compiler parser that parses the FORTR-S language.

SYNOPSIS
java -jar part2.jar [-v] [-wt Filename] SourceFile

DESCRIPTION
The SourceFile specify the path of the file to parse, also the following options are available:

-v Specify a more verbose output explicitly describing the rules which are used

-wt Build the parse tree of the input file and write it as a LaTeX file called Filename

EXAMPLES
java -jar part2.jar -v -wt tree.tex sourceFile.fs

7 Description of example files

Three example files were created.

- IFexample.fs : This example contains a syntax error as there are two numbers following each other at line 3 (as numbers cannot have leading zeroes the scanner detects two numbers).
- WHILEexample.fs : This example does not contain any syntax error. It also shows that the multi end lines rules added to the grammar works.

- ExpectedToken.fs : This example contains a syntax error at line 4 as the ENDIF token is unexpected.

For each source code example there is a ".output" with the expected result from the parser.

8 Conclusion

The complete compiler is almost finished, after the first part which was the implementation of the parser (lexical analyser) and now the LL(1) parser (syntax parser), the last step will be the implementation of the semantic analyser. In this part, the focus was on the syntax analyzer where an LL(1) parser was used to recognize the grammar of the input file. The parser took the token stack generated by the scanner and analyzed it following the rules of the action table on which the parser is based. Since the modified grammar was unambiguous and one token of look-ahead was sufficient, LL(1) can be used, if in the future the grammar language is updated, the parser may not work properly and perhaps the use of more k tokens of look-ahead should be necessary to overcome the problem.

A Action Table

Terminal / Variable	BEGINPROG	PROGNAME	[EndLine]	ENDPROG	[VarName]	ASSIGN	[Number]	(	)	-	+	*	/	IF	THEN	ENDIF	ELSE	=	>	WHILE	DO	ENDWHILE	PRINT	READ	$
<Program>	1	x	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
<Code>	x	x	3	4	2	x	x	x	x	x	x	x	x	2	x	4	4	x	x	2	x	4	2	2	x
<Instruction>	x	x	x	x	5	x	x	x	x	x	x	x	x	6	x	x	x	x	x	7	x	x	8	9	x
<Assign>	x	x	x	x	10	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
<Expr>	x	x	x	x	11	x	11	11	x	11	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
<Expr'>	x	x	14	x	x	x	x	x	14	13	12	x	x	x	x	x	x	14	14	x	x	x	x	x	x
<Prod>	x	x	x	x	15	x	15	15	x	15	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
<Prod'>	x	x	18	x	x	x	x	x	18	18	18	16	17	x	x	x	x	18	18	x	x	x	x	x	x
<Atom>	x	x	x	x	21	x	20	22	x	19	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
<If>	x	x	x	x	x	x	x	x	x	x	x	x	x	23	x	x	x	x	x	x	x	x	x	x	x
<IfTail>	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	25	24	x	x	x	x	x	x	x	x
<Cond>	x	x	x	x	26	x	26	26	x	26	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
<Comp>	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	27	28	x	x	x	x	x	x
<While>	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	29	x	x	x	x	x
<Print>	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	30	x	x
<Read>	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	31	x
<MultiEndLines>	33	x	32	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	33