Université Libre de Bruxelles

ECOLE
POLYTECHNIQUE
DE BRUXELLES

INFO-F403 : Introduction to language theory and compiling

Computer Science Section

Building a FORTR-S compiler : Part 2

Professor:
Gilles Geeraerts

Students :
Miró-Manuel Matagne
Linh Tran Ngoc

Academic Year 2020 - 2021

Contents

1 Introduction

The goal of this project, as part of the course Introduction to language theory and compiling, is to build a compiler for the FORTR-S language. The first part consisted in coding the scanner of this compiler. This second part aims to code the parser for this compiler with the scanner implemented. First of all, the modifications of the grammar are explained. Then, after describing the creation of the action table, an explanation on the utilization of the parser and the different implementations is given. Lastly, the testing implementation and different test files that our group has chosen will be explained.

2 Transforming the grammar

In order to code the parser, the grammar that we use must be LL(1). Thus, the initial grammar has to be transformed. The goal is to transfer the semantics intended by the initial grammar onto the new one. Note that the accepted language of the initial grammar has to be accepted as well in the transformed one. The processing steps are explained in the subsections below.

2.1 Removing unproductive and unreachable variables

The first step to modify the grammar is to remove unproductive and unreachable variables. In order to remove the unproductive variables, we check if there exists a derivation starting from each variable that leads to a word containing only terminals. On the other hand, to remove unreachable variables, we check whether there exists a derivation starting from the start symbol that leads to this variable. These 2 removal methods are illustrated by the algorithms (taken from the exercise sessions) below :

```
Grammar RemoveInaccessible(Grammar $G = \langle V,T,P,S \rangle$) begin
    $V_0 \leftarrow \{S\}$ ; $i \leftarrow 0$ ;
    repeat
        $i \leftarrow i+1$ ;
        $V_i \leftarrow \{X \mid \exists A \rightarrow \alpha X \beta$ in $P \wedge A \in V_{i-1}\} \cup V_{i-1}$ ;
    until $V_i = V_{i-1}$;
    $V' \leftarrow V_i \cap V$ ; $T' \leftarrow V_i \cap T$ ;
    $P' \leftarrow$ set of rules of $P$ that only contain variables from $V_i$ ;
    return($G' = \langle V',T',P',S \rangle$) ;
```

```
Grammar RemoveUnproductive(Grammar $G = \langle V,T,P,S \rangle$) begin
    $V_0 \leftarrow \emptyset$ ;
    $i \leftarrow 0$ ;
    repeat
        $i \leftarrow i+1$ ;
        $V_i \leftarrow \{A \mid A \rightarrow \alpha \in P \wedge \alpha \in (V_{i-1} \cup T)^*\} \cup V_{i-1}$ ;
    until $V_i = V_{i-1}$;
    $V' \leftarrow V_i$ ;
    $P' \leftarrow$ set of rules of $P$ that do not contain variables in $V \setminus V'$ ;
    return($G' = \langle V',T,P',S \rangle$) ;
```

After applying the 2 algorithms, it results that the initial grammar has neither unproductive nor unreachable variables, therefore we do not need to remove any rule or any variable.

2.2 Priority and associativity of operators

In order to make the grammar non-ambiguous by taking into account the associativity and the priority of the operators, a table of priority and associativity of the FORTR-S operators is used (as shown in table 1).

Operators	Associativity
- (unary)	right
*, /	left
+, - (binary)	left
>, =	left

Table 1: Priority and associativity of the FORTR-S operators (operators are sorted in decreasing order of priority). Note the difference between unary and binary minus (-).

Indeed, to put priority and associativity in the current grammar, new variables are created to fulfill these requirements. Note that our group has chosen to take into account multiple [EndLine] terminals in a row (which is a slight modification from the initial grammar), as a result a **<Endline>** variable is created.

Here down below is the modified grammar :

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

This grammar makes sure that the priority and associativity of operators is respected. For example if an addition is followed by a multiplication, the multiplication will be computed in priority, as the rule 13 concerning multiplication is located at a "lower level" than the addition (rule 10). The variable <Op> became unreachable and was, as a result, removed.

2.3 Removing left-recursion

Left recursion is considered to be a problematic situation for top-down parsers. As a result, to avoid left-recursion, we convert the current grammar into a right-recursive grammar. Every time a left-recursion is detected in a rule, a new variable is created, as well as the corresponding new rules. This is why variables like <ExprArith'>, <Prod'> and <EndLine'> appear in the modified grammar below, free of left-recursion :

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

2.4 Applying factorisation

Finally, it remains the factorisation to apply on the grammar to have a LL(1) grammar. Indeed, if factorisation was not applied, the rules 22 and 23 would have the same First sets (these will be detailed later), and have the same variable on the left (<If>). This means that by knowing the variable on the top of the stack is <If> and knowing the next symbol on the input is `IF`, the parser has no way to figure out weather to apply rule 22 or rule 23. This is a typical example where factorisation is necessary, and it is the only case where it is applied in this grammar. Here is the final modified grammar with an applied factorisation :

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

As a result, the variable <IfTail> was created. The grammar is now LL(1), as it will be shown in the next section.

3 Creation of the action table

Before creating the action table of our grammar, and in order to ensure it will contain no conflicts, we have to make sure our grammar is LL(1). Therefore, we need to make sure there are no First/First conflicts, and that there are no First/Follow conflicts. To ensure we have no First/First conflicts, we must make sure there are no duplicates in the First sets of any variable. Then, to ensure we have no First/Follow conflicts, we must make sure that for every variable A for which a rule $A \rightarrow \epsilon$ exists, there are no elements in common in the First and Follow sets of A.

In order to prove the grammar is LL(1), and to build the action table, we must therefore compute the First and Follow sets of all variables present in our grammar.

3.1 Computation of First and Follow sets

- *First*(Program) = {BEGINPROG}

- *First*(Code) = *First*(Instruction) ∪ {ϵ} = {[VarName],IF,WHILE,PRINT,READ,ϵ}
  Follow(Code) = {ENDPROG,ENDIF,ELSE,ENDWHILE}
- *First*(Instruction) = {[VarName],IF,WHILE,PRINT,READ}
  Follow(Instruction) = *First*(EndLine) = {[EndLine]}
- *First*(Assign) = {[VarName]}
  Follow(Assign) = *Follow*(Instruction) = {[EndLine]}
- *First*(ExprArith) = *First*(Prod) = *First*(Atom) = {-,(,[VarName],[Number]}
  Follow(ExprArith) = *Follow*(Assign) ∪ {)} ∪ *First*(Comp) = {[EndLine],),=,>}
- *First*(ExprArith') = {+,-,ϵ}
  Follow(ExprArith') = *Follow*(ExprArith) = {[EndLine],),=,>}
- *First*(Prod) = *First*(Atom) = {-,(,[VarName],[Number]}
  Follow(Prod) = *First*(ExprArith') ∪ *Follow*(ExprArith') = {+,-,[EndLine],),=,>}
- *First*(Prod') = {*,/,ϵ}
  Follow(Prod') = *Follow*(Prod) = {+,-,[EndLine],),=,>}
- *First*(Atom) = {-,(,[VarName],[Number]}
  Follow(Atom) = *First*(Prod') ∪ *Follow*(Prod') = {*,/,+,-,[EndLine],),=,>}
- *First*(If) = {IF}
  Follow(If) = *Follow*(Instruction) = {[EndLine]}
- *First*(IfTail) = {ENDIF,ELSE}
  Follow(IfTail) = *Follow*(If) = *Follow*(Instruction) = {[EndLine]}
- *First*(Cond) = *First*(ExprArith) = {-,(,[VarName],[Number]}
  Follow(Cond) = {)}
- *First*(Comp) = {=,>}
  Follow(Comp) = *First*(ExprArith) = {-,(,[VarName],[Number]}
- *First*(While) = {WHILE}
  Follow(While) = *Follow*(Instruction) = {[EndLine]}
- *First*(Print) = {PRINT}
  Follow(Print) = *Follow*(Instruction) = {[EndLine]}
- *First*(Read) = {READ}
  Follow(Read) = *Follow*(Instruction) = {[EndLine]}
- *First*(EndLine) = {[EndLine]}
  Follow(EndLine) = {VarName,IF,WHILE,PRINT,READ,ENDPROG,ENDIF,ELSE,ENDWHILE}
- *First*(EndLine') = {[EndLine], ϵ}
  Follow(EndLine') = *Follow*(EndLine) = *First*(Code) ∪ *Follow*(Code) =
  {VarName,IF,WHILE,PRINT,READ,ENDPROG,ENDIF,ELSE,ENDWHILE}

By analyzing these results, we see that there are no duplicates in any First sets for any of the variables, which means the grammar has no First/First conflicts. Moreover, the only variables that contain a rule of the type $A \to \epsilon$ are Code, ExprArith', Prod' and EndLine'. For these 4 variables, we see that there are no common elements between the First and Final sets, which means there are no First/Follow conflicts.

As the grammar does not present any First/First conflict nor any First/Follow conflict, it is an LL(1) grammar. Note that we could have deduced that the grammar is LL(1) by building the action table and noticing that there are no conflicts (only one rule per cell of the action table).

3.2 Action table

In order to build the action table, the following algorithm (taken from the exercise sessions) was used :

```
begin
    $M \leftarrow \times$ ;
    foreach $A \rightarrow \alpha$ do
        foreach $a \in First^1(\alpha)$ do
            $M[A,a] \leftarrow M[A,a] \cup \text{Produce}(A \rightarrow \alpha)$ ;
        if $\varepsilon \in First^1(\alpha)$ then
            foreach $a \in Follow^1(A)$ do
                $M[A,a] \leftarrow M[A,a] \cup \text{Produce}(A \rightarrow \alpha)$ ;
    foreach $a \in T$ do $M[a,a] \leftarrow$ Match ;
    $M[\$,\varepsilon] \leftarrow$ Accept ;
```

The First and Follow sets computed at the previous subsection were used to compute the action table. In this table, the numbers represent the number of the rule that has to be applied when the symbol on the top of the stack is the variable in the first column, and the next symbol on the input is the terminal in the first line.

For most of the lines, the completion of the action table is pretty straightforward : for each rule of our grammar, we have a look at the variable on the left hand side, and at the first terminal on the right hand side (using the previously computed First sets). We mark the cell having the same line as the variable and the same column as that first terminal by the number of the corresponding rule. For example : the rule number 4 is $Instruction \rightarrow Assign$, and we know that *First*(Assign) = {[VarName]}, so we put a 4 in the case at the intersection of Instruction and [VarName].

For rules of the type $A \rightarrow \epsilon$ however, the situation is a bit different. The next symbol on the input is not trivial, and we need to use the Follow set of A to find it. For example, the rule 13 is $ExprArith' \rightarrow \epsilon$, and we know that *Follow*(ExprArith') = {[EndLine],),=,>}, so whenever one of those terminals is the next symbol on the input and that ExprArith' is on top of the stack we must use rule 13. This is why cells (ExprArith',[EndLine]), (ExprArith',)), (ExprArith',=) and (ExprArith',>) all contain 17.

Note that the following action table only contains the Produces, and not the Matches, which are trivial (every terminal matches itself). There is also an Accept that is not represented here, which is the Match of the terminal ENDPROG with itself.

	BEGINPROG	ENDPROG	ProgName	EndLine	:=	VarName	+	-
Program	1							
Code		3				2		
Instruction						4		
Assign						9		
ExpArith						10		10
ExpArith'				13			11	12
Prod						14		14
Prod'				17			17	17
Atom						20		18
If								
IfTail								
Cond						25		25
Comp								
While								
Print								
Read								
EndLine				31				
EndLine'		33		32		33		

	*	/	(	)	Number	IF	THEN	ENDIF
Program								
Code						2		3
Instruction						5		
Assign								
ExpArith			10		10			
ExpArith'				13				
Prod			14		14			
Prod'	15	16		17				
Atom			19		21			
If						22		
IfTail								23
Cond			25		25			
Comp								
While								
Print								
Read								
EndLine								
EndLine'						33		33

	ELSE	=	>	WHILE	DO	ENDWHILE	PRINT	READ
Program								
Code	3			2		3	2	2
Instruction				6			7	8
Assign								
ExpArith								
ExpArith'		13	13					
Prod								
Prod'		17	17					
Atom								
If								
IfTail	24							
Cond								
Comp		26	27					
While				28				
Print							29	
Read								30
EndLine								
EndLine'	33			33		33	33	33

4 Utilization

In order to write on the standard output stream the leftmost derivation of a given FORTR-S code, one should navigate to the *dist* folder and run the command :

```
java -jar part2.jar sourceFile.fs
```

where `sourceFile.fs` is the file we want to parse.

Options can be introduced in the command line, if the user wishes to make the output more verbose, he can run the following command with the flag -v :

```
java -jar part2.jar -v sourceFile.fs
```

This will show the user details of every match, as well as the rules used in a more "written" format (the rules are written such as in the report at section 2.4).

In addition, the user can store in a specified .tex file a parse tree provided by the parser, the flag -wt has to be used in the following command :

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

where `tree.tex` is the file we want to store the parse tree into.

Note that the user can also quickly check the project is functional by using the Makefile in the root directory. One can run `make` to compile, and `make normal`, `make verbose` or `make tree` in order to test the corresponding features.

As in the part 1 of the compiler project, a testing system is implemented in order to verify that the parser works properly. This system will be discussed in detail in section 6 below.

5 Implementation

5.1 Main file

First of all, the Main class goes through the options provided in the command line used to run the program, to see if the user wishes to get a more verbose feedback of the parsing (with the flag -v), or if he wishes to get a parse tree relative to the parsing in a LaTex file (with the flag -wt).

Then, if there are no errors, the lexical analyser is called and the list of tokens is retrieved. Next, the parser is called, and the returned list of used rules is stored. Again after checking there were no errors, we print the rules out to the user, each number being separated by a space. If the user wished to get a parse tree, the relevant LaTex code is written to the input file indicated by the user.

5.2 Parser

The implemented parser is a recursive parser. Therefore, to each non-terminal corresponds a different function, which is called when needed, following the rules of the grammar and the action table described previously. To know which rule to apply, the Parser checks the next token as well as the variable on the top of the stack, and then uses the rule computed in the action table.

The Parser class has a constructor taking as arguments the list of tokens returned from the lexical analyzer, as well as a boolean indicating if the user wants a verbose output or not. In addition to these 2, the Parser class contains 4 other static variables : an index indicating what symbols of the token list have we already matched, a list containing all the numbers of the rules used during parsing, in chronological order, a string corresponding to an error message in case an error occurs during parsing, and a ParseTree (explained in the next section).

The *start* method of the Parser class is called to start the parsing, it immediately calls the *program* function as the parsing inevitably starts by the rule number 1. After the whole parsing process is done, it returns the list of rules, in case no error was detected during the parsing.

In practice, during the parsing, every time we encounter a non-terminal, a function associated with it is called. For each function, a ParseTree is created, which will contain all the children of the current node. On the other hand, when a terminal is met, the *match* function is called, which will do two things :

- Check that the next symbol to be matched has the correct type in order to respect the grammar. If it is the case, the index is incremented, and a message is displayed to the user in case the verbose boolean is true. In case the type is incorrect, in means there is an error in the FORTR-S code parsed, and an error message giving details on the error is saved.
- In case there is no error, the *match* function adds the token to the ParseTree of the calling entity.

The verbose mode allows the user to see everytime a rule has been used, and everytime a match occurs. Additionally, at the end of the parsing, a full output of all the used rules in verbose mode is shown to the user.

5.3 Parse Tree

The ParseTree class is used to produce a parse tree in a LaTex file after the parsing. The Parser class recursively adds elements to the parse tree during the parsing, and when the parsing is done, the *toLaTexTree* function is called on that final parse tree. This function will compute the necessary LaTex code, using the *toTexString* function of the Symbol class in order to get a string corresponding to each symbol. For terminals, that strings consists of the type of the symbol, and its value (for example if we have a VarName called *abc*, the output will be "VARNAME abc"). For non-terminal symbols, only the name of the symbol is returned (for instance "ExprArith"). The final string corresponding to the entirety of the LaTex code to represent the parse tree is then returned, and stored in the file specified by the user.

6 Testing

As mentioned previously, our group chose to implement a testing system to make sure that the LL(1) grammar is well applied on the test files. In order to do that, folders are created for the following variables or group of variables :

- Atom
- Comments
- Errors
- FortrS
- Instructions
- Program

In each one of those folders, there is a *.fs* file and a *.output* file. The *.fs* file is the file used to test the parser and the *.output* file is the correct output containing the rules of the leftmost derivation from the *.fs* file that is expected after parsing it. Therefore, we have used the *diff* bash command

in order to compare the output of the parser on the *.fs* file to the *.output* file.
In order to use the testing system, the user can simply go to the *test* folder and enter one of the following commands to test a particular group of regular expressions :

- `make atom`
- `make comments`
- `make errors`
- `make average`
- `make exponent`
- `make factorial`
- `make fibonacci`
- `make maximum`
- `make instructions`
- `make program`

As the part 1, the output of these commands will be a string. For example for the test related to the atoms, the output can be either *Atom : Test Successful* or *Atom : Test Failed.* The *diff* command also gives a very brief detail about where the differences between the 2 files are in case of a failure.

If the user wishes to test all the possible regular expressions at once (therefore run all the commands listed above), he can run the command `make test`.

The user can also run all the FortrS folder tests at once using `make fortrs`.

7 Conclusion

In conclusion, we have built a fully functional parser for the FORTR-S language, which we were able to build by modifying the original grammar in several ways to make it LL(1).