INFO-F403 Project part 2 : Parser

Vincent Gailly 547819 - Oruç Kaplan 540662

November 2022

1 Introduction

This part of the project consists in the creation of a parser for the Fortress language. The parsing comes right after the lexical analysis in the compilation process of an input. Its role is to ensure that the input data respect the syntax specified by a grammar like this :

[1]	<Program>	→ BEGIN[ProgName]<Code>END
[2]	<Code>	→ <Instruction>,<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]		→ +
[16]		→ -
[17]		→ *
[18]		→ /
[19]	<If>	→ IF (<Cond>) THEN <Code> END
[20]		→ IF (<Cond>) THEN <Code> ELSE <Code> END
[21]	<Cond>	→ <ExprArith><Comp><ExprArith>
[22]	<Comp>	→ =
[23]		→ >
[24]		→ <
[25]	<While>	→ WHILE(<Cond>) DO <Code> END
[26]	<Print>	→ PRINT([VarName])
[27]	<Read>	→ READ([VarName])

2 Grammar

Before creating a parser via a programming language, we first modified the grammar to make it usable. This section therefore summarizes the various pre-processing steps performed on the basic grammar. Below you will find our final grammar after our modifications :

[1]	<Program>	→ BEGIN[ProgName]<Code>END
[2]	<Code>	→ <Instruction>,<Code>
[3]		→ ϵ
[4]	<Instruction>	→ <Assign>
[5]		→ <If>
[6]		→ <While>
[7]		→ <Print>
[8]		→ <Read>
[9]	<Assign>	→ [VarName]:=<ExprArith>
[10]	<ExprArith>	→ <MulDiv><ExprArith'>
[11]	<ExprArith'>	→ + <MulDiv><ExprArith'>
[12]		→ - <MulDiv><ExprArith'>
[13]		→ ϵ
[14]	<MulDiv>	→ <Atom><MulDiv'>
[15]	<MulDiv'>	→ * <Atom><MulDiv'>
[16]		→ / <Atom><MulDiv'>
[17]		→ ϵ
[18]	<Atom>	→ -<Atom>
[19]		→ [VarName]
[20]		→ [Number]
[21]		→ (<ExprArith>)
[22]	<If>	→ IF (<Cond>) THEN <Code> <IfSeq>
[23]	<IfSeq>	→ END
[24]		→ ELSE <Code> END
[25]	<Cond>	→ <ExprArith><Comp><ExprArith>
[26]	<Comp>	→ =
[27]		→ >
[28]		→ <
[29]	<While>	→ WHILE(<Cond>) DO <Code> END
[30]	<Print>	→ PRINT([VarName])
[31]	<Read>	→ READ([VarName])

2.1 Unproductive variables

The first step performed is the removal of unproductive variables. A variable is unproductive if any derivation of the grammar passing through this variable recursively recalls this variable and leaves no possibility to stop this recursion.

In order to remove the unproductive symbols, we used the following algorithm from the Figure 1:

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$(G' = \langle V', T, P', S \rangle)$;

Figure 1: unproductive symbol deletion algorithm

In our case, we did not need to apply any changes for this part of the project.

2.2 Unreachable symbols

Another type of symbol that is unnecessary and can be removed are unreachable symbols. These are symbols that no sentential form obtained from the start symbol will ever contain. It is important to remove unproductive variables first before applying the current step, because this previous step may generate new inaccessible symbols.

In order to remove the unreachable symbols, we used the algorithm from the Figure 2:

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$(G' = \langle V', T', P', S \rangle)$;

Figure 2: unreachable symbol deletion algorithm

In our case, we did not need to apply any changes for this part of the project.

2.3 Ambiguity removal

One of the most important operations to perform on a grammar is to check that it is not ambiguous. "A CFG is ambiguous if and only if it generates at least one word that admits two different derivation trees". In other words, there must be only one way to construct the derivation tree.

This case can very easily occur when the grammar contains arithmetic expressions (see Figure 3). For example, when an addition is followed by a multiplication, should the addition be done first or the multiplication ? In short, the grammar must respect the priority of operations.

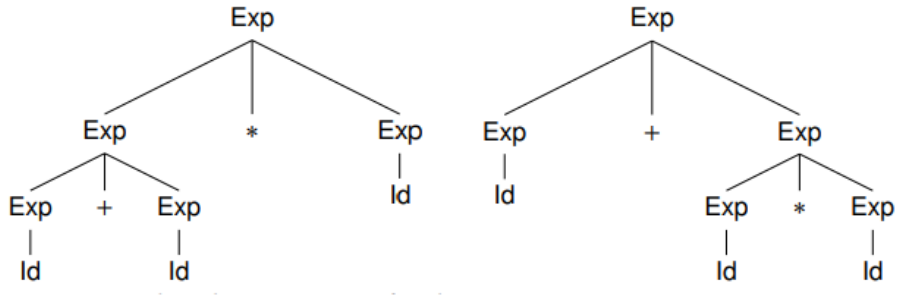

Figure 3: Two possible derivation tree for the word Id + Id * Id.

This is why we have transformed the part of the grammar concerning arithmetic expressions, like this :

```
[1]    <Cond>        → <ExprArith> <Comp> <ExprArith>
[2]    <Comp>        → =
[3]                  → >
[4]                  → <
[5]    <ExprArith>   → <ExprArith> + <MulDiv>
[6]                  → <ExprArith> - <MulDiv>
[7]                  → <MulDiv>
[8]    <MulDiv>      → <MulDiv> * <Atom>
[9]                  → <MulDiv> / <Atom>
[10]                 → <Atom>
[11]   <Atom>        → -<Atom>
[12]                 → [VarName]
[13]                 → [Number]
[14]                 → (<ExprArith>)
```

2.4 Left-recursion removal

Another operation to perform in order to make the grammar regular, is to check that there is no mixture of left-recusion and right-recursion.
Following the previous modification, we end up with a grammar that does not respect this rule. To fix this, we applied the left-recursion removal operation. This one consists in removing the left-recursion by creating intermediate rules.

This leads to a left-regular grammar:

[1]	<Cond>	→ <ExprArith> <Comp> <ExprArith>
[2]	<Comp>	→ =
[3]		→ >
[4]		→ <
[5]	<ExprArith>	→ <MulDiv><ExprArith'>
[6]	<ExprArith'>	→ + <MulDiv><ExprArith'>
[7]		→ - <MulDiv><ExprArith'>
[8]		→ ϵ
[9]	<MulDiv>	→ <Atom><MulDiv'>
[10]	<MulDiv'>	→ * <Atom><MulDiv'>
[11]		→ / <Atom><MulDiv'>
[12]		→ ϵ
[13]	<Atom>	→ -<Atom>
[14]		→ [VarName]
[15]		→ [Number]
[16]		→ (<ExprArith>)

2.5 Factoring

The last modification we did to the grammar is the factoring. It can be done when the grammar contains two rule with the same left-hand side and both right-hand side begins in the same way, with the same prefix. It is the case with the `<If>` rule in which the end can be divided into two rules depending if an "ELSE" exist. This is done by adding the rule `<IfSeq>` :

[1]	<If>	→ IF (<Cond>) THEN <Code> <IfSeq>
[2]	<IfSeq>	→ END
[3]		→ ELSE <Code> END

2.6 Checking if the grammar is LL(1)

In order to check if the grammar is LL(1). First we need to construct the list of first(X) and follow(X). Then, it will help us to build the action table for the parser. This table allows us to know if there is no ambiguity about the choice of rules that the parser will make.

2.6.1 First and Follow sets

Here is the list of firsts and follows :

X	**First[1](X)**	**Follow[1](X)**
<Program>	BEGIN	
<Code>	[VarName] IF WHILE PRINT READ ϵ	END ELSE
<Instruction>	[VarName] IF WHILE PRINT READ	,
<Assign>	[VarName]	,
<ExprArith>	- [VarName] [Number] (	,) = > <
<ExprArith'>	+ - ϵ	,) = > <
<MulDiv>	- [VarName] [Number] (	+ - ,) = > <
<MulDiv'>	* / ϵ	+ - ,) = > <
<Atom>	- [VarName] [Number] (	+ - ,) = > < * /
<If>	IF	,
<IfSeq>	END ELSE	,
<Cond>	- [VarName] [Number] (	)
<Comp>	= > <	- [VarName] [Number] (
<While>	WHILE	,
<Print>	PRINT	,
<Read>	READ	,

Thanks to these sets, we can already have a hint of what the action table will look like. If we look at the rules that can become an epsilon. We can already see that the symbols in the first and follow columns are completely different. This means that there will probably be no ambiguity if the symbol is epsilon or not for the corresponding rule. But to be sure that there isn't any ambiguity, it is better to build the action table.

2.6.2 Action table

Using the action table, we can check if the grammar is not ambiguous in LL(1). To do this, after the construction, we have to check that there are not two applicable rules, in the same context for a specific symbol encountered by the parser.

In our action table, the empty cells correspond to grammar errors and the filled cells correspond to the rules to apply in the given context.

As we can see in our action table (divided in two parts). There is no situation where the parser would not know which rule to apply among two rules. To make it simple, there should be at most one rule per cell.

	BEGIN	[VarName]	[Number]	IF	WHILE	PRINT	READ	ELSE	END
<Program>	[1]								
<Code>		[2]		[2]	[2]	[2]	[2]	[3]	[3]
<Instruction>		[4]		[5]	[6]	[7]	[8]		
<Assign>		[9]							
<ExprArith>		[10]	[10]						
<ExprArith'>									
<MulDiv>		[14]	[14]						
<MulDiv'>									
<Atom>		[19]	[20]						
<If>				[22]					
<IfSeq>								[24]	[23]
<Cond>		[25]	[25]						
<Comp>									
<While>					[29]				
<Print>						[30]			
<Read>							[31]		

Table 1: Action Table

	(	)	+	-	*	/	=	<	>	,
<Program>										
<Code>										
<Instruction>										
<Assign>										
<ExprArith>	[10]			[10]						
<ExprArith'>		[13]	[11]	[12]			[13]	[13]	[13]	[13]
<MulDiv>	[14]			[14]						
<MulDiv'>		[17]	[17]	[17]	[15]	[16]	[17]	[17]	[17]	[17]
<Atom>	[21]			[18]						
<If>										
<IfSeq>										
<Cond>	[25]			[25]						
<Comp>							[26]	[28]	[27]	
<While>										
<Print>										
<Read>										

Table 2: Action Table (continued)

3 Code

3.1 Main class

This class handles the parsing of the arguments given to the program (e.g. : read the input file, create in a tex file the parsing tree or not).

In addition to that, it is this class that launches the lexical analyzer made in the first part of the project (we kept the one we had made) and the parser developed for this part.

3.2 Change in Symbol class

The only change we made to this class is the implementation of the "to-TexString" method. This method is used to return the latex code needed to generate the value of a symbol in a tex file (for the visual representation of the parse tree).

3.3 TexHandler class

This class manages the creation of the latex file which will contain the parse tree if it does not exist. In addition to that, it adds the content of the file (the parse tree in TeX language).

3.4 Parser class

A Parser object takes as parameter a list of tokens (Symbol) in its constructor. This list of tokens is provided by the lexical analyser realized in the previous part.

A Parser object also contains two other attributes :

- **leftMostDerivationArray :** This one is a list of integers filled with the numbers of the rules of the grammar used throughout the parsing. It allows to obtain the left-most derivation of the input data and can be easily displayed in the console thanks to its getter which returns the list as a string.
- **parseTree :** This attribute contains a ParseTree object which is none other than the root node of the parse tree representing the input data once the parse has been performed. This object contains the root node as well as its children which are themselves objects that can also reccursively have children.

Here is also a list of the most important methods that make up this class :

- A method **syntax_error(List<LexicalUnit> expected)** which throws an exception and precise where is the token on which the error occurs. It also precises the list of lexical units that were expected.
- A method **match(LexicalUnit lu)** which verifies that the type of the first token of the list is the same as the one in parameter. It calls the syntax_error method if it is not the case.
- A method **next_token()** which returns the next token of the list of tokens. If the list is empty it returns a null.
- A method **parse()** which launches the parsing on the list of tokens and displays in the console the left-most derivation if everything went well.

As for a grammar, our parser works in a reccursive way. A method has therefore been implemented for each element of the left-hand side of our final grammar.

In general, these methods work as follows:

- It takes the next token to process.
- It compares this token with the different cases given in the action table (depending on the method we are in at the time t).
- If the case exists, we add the rule number to the "leftMostDerivationArray" list and we add a new node to our parse tree. If it does not exist, the syntax_error() method is called.

3.5 Tests

In the test folder you will find a bunch of .fs files which correspond to all the tests we've done. Basically we tested if each rule work properly, so each file correspond to one or multiple rules.

The tests consist of testing all the possible cases of a rule;
For example for an "IF", we check if it work while being empty in the code section in it, with or without an "ELSE";
We did it the same for "WHILE" and also tests all the other rules.

Another example and maybe the most important test is testing the arithmetical expressions. To do so, in *RuleExprArithBasic.fs*, you will find nearly all the possible basic arithmetical expressions and in *RuleExprArithPriority.fs*, you will find the expressions for which the priority has an importance.

Finally, for the rule "PROGRAM", we tested if it crash properly if the code is unfinished or if there is something after the "END". A normal parser should not accept the code if there is something after the end of the code.

3.6 Usage

In order to use our parser, a makefile is provided to be able to compile the source code.

- How to compile:
  You can run the makefile using the command : *make*
- How to run:
  - You can run the makefile using the command : *make basic*
    Will run the parser on the *factorial.fs* file.
  - You can run the makefile using the command : *make testing*
    Will run the parser on all the *.fs* file of the test directory.
  - You can use the command :
    java -jar dist/part2.jar <your code file>
    Will run the parser on your own code.
  - You can use the command :
    java -jar dist/part2.jar <your code file> -wt <your code file>.tex
    Will run the parser on your own code and generate the parse tree in the corresponding tex file.

4 Conclusion

We learned more in depth through this part of the project how the parsing phase of a compiler works. This project allowed us to have a more concrete idea of how to implement this step of compilation for a programming language.