INFOF403 Project

Part 2

Submitted to:

Gilles GEERAERTS
Léonard BRICE
Sarah WINTER

Submitted by:

Bermudez Arroyo Loïc

Nabbout Fadi

Submitted on:

28/11/2022

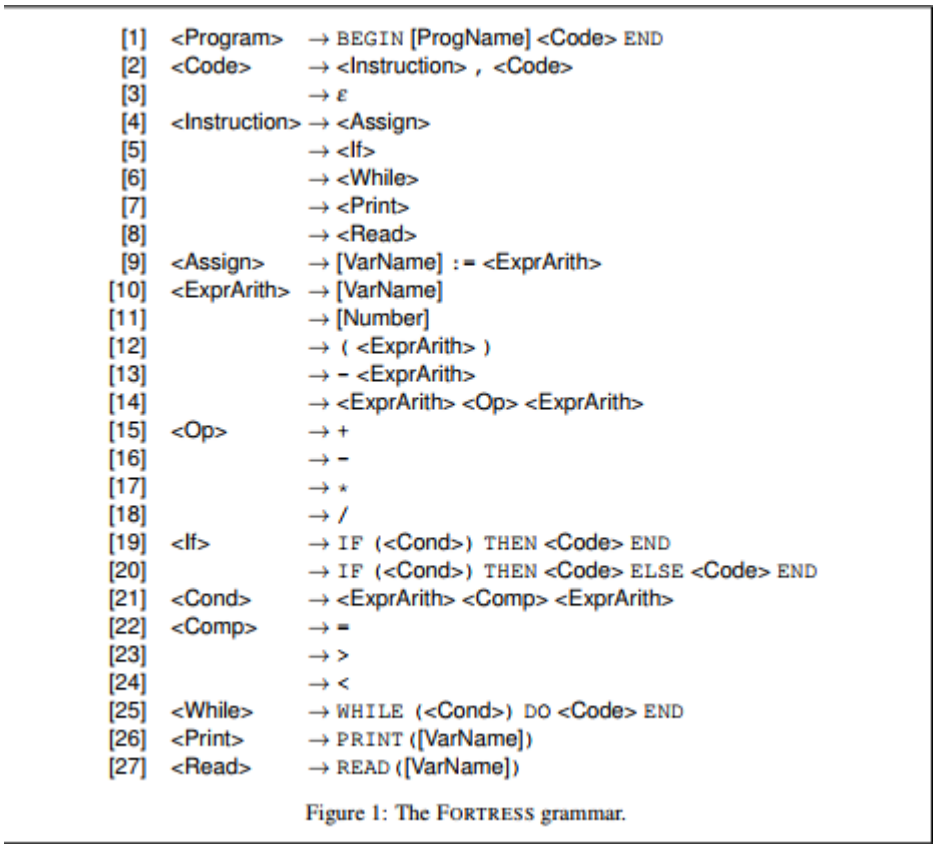
1. Introduction:

The second part of the project is about creating the parser, which will analyze the input file and build the derivation tree from it. But first we need to build an LL(1), standing for Left scanning, Left parsing grammar as the input string is read (scanned) from left to right and the parser builds a leftmost derivation when correctly recognizing the input words/tokens.

For this we will start form the original given grammar and transform it by removing the unproductive/unreachables variables if there are any.

Then we make the grammar non-ambiguous through setting a hierarchy by taking account of the priority and associativity of the operators.

Then we make it LL(1) by removing left recursion and finally factoring the grammar.
We then build the first and follow table from the obtained LL(1) grammar and build from those tables the action table, which will help to prove the grammar is well LL(1).



[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]	<Op>	→ +
[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])

Figure 1: The FORTRESS grammar.

Original given Fortress grammar

2. Unproductive variables removal

To remove all the unproductive variables, we need to check which variable (non-terminals) produce terminals through the rules, within a certain number of iterations. For this we start from the top of the grammar, and we build the sets iteratively.

i	Vi
0	{Ø}
1	{Code, ExprArith, Op, Comp, Print, Read}
2	{Code, ExprArith, Op, Comp, Print, Read, Program, Instruction, Assign, Cond}
3	{Code, ExprArith, Op, Comp, Print, Read, Program, Instruction, Assign, Cond, If, While}

We now, know that every variable that are not present in the third set, is considered unproductive and is to be deleted from the grammar. After comparing with the original grammar, we noted that all variables are contained in the thirds set, then we cannot reduce the grammar.

3. Unreachable variables removal

This step of the development is about removing every variable that cannot be reached through derivation after a certain number of iterations. As the unproductive variable removal, this step is done recursively going from top to bottom, building the table iteration after iteration

i	Vi
0	{Program}
1	{Program, Code}
2	{Program, Code, Instruction}
3	{Program, Code, Instruction, Assign, If, While, Print, Read}
4	{Program, Code, Instruction, Assign, If, While, Print, Read, ExprArith, Cond}
5	{Program, Code, Instruction, Assign, If, While, Print, Read, ExprArith, Cond, Op, Comp}

As we can see, every variable which are not in the fifth set, won't be kept in the grammar. After comparison, every variable from the original grammar is present in the set, there are no reduction here.

4. Non-ambiguous grammar

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

Figure 2: Non-ambiguous grammar

We adapted the original grammar into this configuration, this way we consider and respect the priority of operations (figure 3). The priority is hierarchized from top, the least priority to bottom the most priority.

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

Figure 3: priority and associativity

5. LL(1) grammar

Here we removed every possible left-recursion. For example, we have the 15th rule which is:

```
[15] <ExprArith>        → <ExprArith> + <Prod>
[16]                    → <ExprArith> - <Prod>
[17]                    → <Prod>
```

Here, we have multiple possibilities, and we want the grammar to be deterministic and the look ahead can only analyze one token at time and cannot go backward. The look ahead in the case upward, cannot know which case to choose. We then transform it that way:

```
[20] <ExprArith>        → <Prod> <ExprArithF>
[21] <ExprArithF>       → + <Prod> <ExprArithF>
[22]                    → - <Prod> <ExprArithF>
[23]                    → ε
```

Furthermore, that configuration respect also the associativity of the operators.

And finally we also applied factorization on some rules, for example:

```
[9] <If>              → IF (<Cond>) THEN <Code> END
[10]                  → IF (<Cond>) THEN <Code> ELSE <Code> END
```

As those two rules have the same prefix in the right hand side, we factored them to make them deterministic, as we got the same problem for the look ahead.

```
[12] <If>               → IF (<Cond>) THEN <Code> <IfSeq>
[13] <IfSeq>            → END
[14]                    → ELSE <Code> END
```

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

Figure 4: LL(1) grammar

6. First and Follow

Non Terminal Symbols	First(X)	Follow(X)
Program	BEGIN	$\mathcal{E}$
Code	VarName, IF, WHILE, PRINT, READ	END, ELSE
CodeF	","	END, ELSE
Instruction	VarName, IF, WHILE, PRINT, READ	"," , END, ELSE
Assign	VarName	"," , END, ELSE
If	IF	"," , END, ELSE
IfSeq	END, ELSE	"," , END, ELSE
While	WHILE	"," , END, ELSE
Cond	-, (, Number, VarName	)
Comp	=, >, <	)
ExprArith	-, (, Number, VarName	",", =, >, <,), ELSE, END
ExprArithF	+, -, $\mathcal{E}$	",", =, >, <, END, ELSE,)
Prod	-, (, Number, VarName	"," ,=, >, <, +, -, END, ELSE,)
ProdF	*,/,$\mathcal{E}$	"," ,=, >, <, +, -, END, ELSE,)
Atom	-, (, Number, VarName	",",), *, /, +, -, =, >, <, END, ELSE
Print	PRINT	"," , END, ELSE
Read	READ	"," , END, ELSE

6.1. First computation

We simply look the element produced by a variable, if it is a terminal element, we add it to the table. But, if the first element is also a non-terminal element (a variable), we derive the non-terminals until reaching a non-terminal, for example:

ExprArith -> Prod ExprArithF, Prod -> Atom ProdF, Atom -> - Atom, (ExprArith), Number, VarName.

We then take { -; (; Number; VarName}.

6.2. Follow computation

Here we look at the terminal element following each variable in the right-hand side of the grammar, for example:

ExprArith -> Prod ExprArithF, where Prod -> Atom ProdF, where ProdF can be derived as * Atom ProdF or /Atom ProdF, we then can take * and / as follow(ExprArith)

If we look at ExprArithF, it can be derived as + Prod ExprArith or – Prod ExprArith, we then can add + and – in the set follow(ExprArith), and so on.

7. Action table

Variables	BEGIN	ProgName	END	,	VarName	:=	IF	THEN	ELSE	WHILE	DO	=	>	<	+	-	*	/	(	)	Number	PRINT	READ
Program	1																						
Code			3	23	2		2		3	2												2	2
CodeF			5	4					5														
Instruction					6		7			8												9	10
Assign					11																		
If							12																
IfSeq			13						14														
While										15													
Cond					16											16			16		16		
Comp												17	18	19									
ExprArith					20											20			20		20		
ExprArithF			23	23					23			23	23	23	21	22				23			
Prod					24											24			24		24		
ProdF			27	27					27			27	27	27	27	27	25	26		27			
Atom					31											28			29		30		
Print																						32	
Read																							33

The action table is filled following this thinking:

For each variable, we look the rules producing each terminal present in the corresponding First(x) table and add them to the corresponding cross path non-terminal/terminal. And for every nonterminal producing an empty word, we then add that rule to the cross-path non-terminal/follow(non-terminal). If there are, for every case, one and only one rule, then the grammar is ll(1).

That's our case here, we can then say our grammar is LL(1).

8. Implementation

We implemented a recursive descent parsing predictive parser, as we use a look ahead to check what should follow each token.

Each non-terminal element has his corresponding function, where if there are multiple possibilities, there are switch cases leading to the corresponding situation thanks to the look ahead.

At each function we return a ParseTree, which correspond to a parent node of a tree, each node corresponding to a non-terminal element has children set in an array list. Every terminal element corresponds to a leaf to the tree, or in another word corresponds to a parent node without children.

We also implemented a match function, which will check if the look ahead corresponds to the good variable, if not it leads to an error function which will print in the stderr which token is bad read and the corresponding line and column and will finally exit the program.

A function named getNextToken was also implemented, it only checks if the look ahead was already analyzed or not, if yes then it takes the following token, if not it stays as it is.

In the main, we implemented a if loop which will check whether the arguments are correctly entered, if it miss any argument or not. If it detects an “-wt” statement as argument then it needs two arguments after, if there is not then it expects only one argument which is the input file.