ALGOL-0 Introduction to language theory and compiling Project – Part 2

Mounir Bounamar Mustapha Cherrabi

November 2019

1 Introduction

As part of this course, we were asked to implement a compiler for the ALGOL-0 language. This project is divided into 3 parts and the second part was to implement a parser for the ALGOL-0 language.
The parser has to check if a token sequence respects the rules of the grammar using recursive decent LL(1). The grammar has been given to us and we had to transform it in order to match the demand. At the end of the parsing, the program must display the rules used and optionally the derivation tree.

2 Grammar

We modified the grammar (fig 7) in order to make it non-ambiguous. We also had to factorize some rules like this one :

```
[22]  <If>    → if <Cond> then <Code> endif
[23]          → if <Cond> then <Code> else <Code> endif
```

Figure 1: *if* rules before factorization

using this algorithm :

LeftFactor(Grammar $G = \langle V, T, P, S \rangle$**) begin**
  while *G has at least two rules with the same left-hand side and a common prefix* **do**
    Let $R = \{A \to \alpha\beta, \ldots, A \to \alpha\zeta\}$ be such a set of rules ;
    Let $\mathcal{V}$ be a new variable;
    $V = V \cup \mathcal{V}$;
    $P = P \setminus R$;
    $P = P \cup \{A \to \alpha\mathcal{V}, \mathcal{V} \to \beta, \ldots, \mathcal{V} \to \zeta\}$;

Figure 2: Left factoring

The result of such factorization is the addition of a new rule allowing us to go either in a *else* condition or ending the *if* statement.

```
/* 28 */"<If> -> if <cond> then <code> <ifPrim> "
/* 29 */"<ifPrim> -> endif"
/* 30 */"<ifPrim> -> else <Code> endif"
```

Figure 3: *if* rules after factorization

There was no unproductive nor unreachable variables in the given grammar. Indeed, to prove it we only had to apply the algorithms seen in course. The next two figures are one of them.

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 4: Removal of unproductive symbols

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 5: Removal of inaccessible symbols

3 Choice of implementation and hypotheses

We decided to build a recursive descent LL(1) parser because we just had to rely on the modified grammar in order to implement it. It was easier for us and more intuitive to do it like that. Indeed, each state of the grammar will match

a function in our **parser.java** file. Thus, we will have the possibility to know which rule is applied in which function by simply calling the *getRule* function in the **Grammar.java** file.

3.1 Match symbol

In our **Parser** class we take the next token in order to match it with what we are supposed to have in our grammar. We will keep on moving until we reach the last token represented by the lexical unit *END_OF_STREAM*. It is important to note that we manually add this token right after our lexical analyzer finished matching each token in order to have an end in our symbol reading. We made the hypothesis that we can not have multiple programs in one file (this may be implemented in the next part). Thus, our program is supposed to stop right after the token end. This is tested in the *18_IncorrectSeveralPrograms.alg* example, which returns an error.

3.2 Handling exception

In order to build a compact parser, we need to be able to throw an error message to the user each time we received a token that was not supposed to be there. Indeed, in this message we specify which token(s) was expected.
We could have simply throw an error without saying anything more but we decided to have a proper message because it does not make any sens to only throw an error without any specification.

In addition, it facilitates the work of the user who will know directly what is the error made. All of that is handled by our class **HandleException.java**.

3.3 Modified File

Symbol : We had to add 4 functions that allow us to check which lexical unit the symbol is and a function that will be used for the creation of the tree. Indeed, we assumed that for each function in the parser, we would create a sub-tree with the state in which we are as root and a list of child corresponding to the rest of the grammar.
ParseTree : We added a function that gives us the possibility to transform our *ParseTree* into a latex file with the right coding convention because latex does not handle certain types of symbol like "_". We do that in the function *Tolatex*.

4 Javadoc and tests files desciption

We generated the javadoc automatically thanks to the javadoc library made available by java. Javadoc is very useful for monitoring our project and can be consulted to stay updated about the project. In addition, it gives a good support for someone outside the project wanting to understand our code. It is important to note that we are using *lualatex 1.0.4* for the creation of the **pdf**

from the tree deviation and *java 11 (11.0.4)*.
In order to launch the program, you can run the makefile on the terminal by using this command **make testing** who will launch all the tests available in our folder test. It is also possible to launch it via the command **java -jar part2.jar sourceFile.alg**. The test file must be an **.alg** file otherwise the program will not be able to execute it and will signal it to the user. We have a test folder to facilitate the testing. In this folder we find different **.alg** files and their output/graph derivation tree when there is no error. Those file are used to test specific rules (nested if,pending while,..). File names refers to what they do, here a some examples of our test folder :

1. **03_euclide.alg and 15_complexComparaison.alg** these two files allow us the test the basics of our program. The derivation tree of these files are available in the test folder.

2. **08_Not.alg** this example is interesting in the sens that the "not" can only apply to an simple condition meaning that not $a > 0$ and $b > 0$ will be evaluated to true if and only if $a > 0$ is equal to false and $b > 0$ is true. We could have add parentheses to make the "not" apply to the whole condition but we find it more interesting to do it for part 3.

3. **11_testMinus.alg** this file will be used to test the case where we have multiple minus for a variable. Our lexical analyzer consider the minus and the number as two different and distinct token (*token minus and number*). Now, thanks to our modified grammar, they will be consider as an *atom* (rule 16 of fig 7). We can then add as many minus in front of a number, it will still be considered as a negative/positive number (depending on the number of minus). Here is the derivation tree of the file showing that the minus is taken as a sign and not as an arithmetic operator.

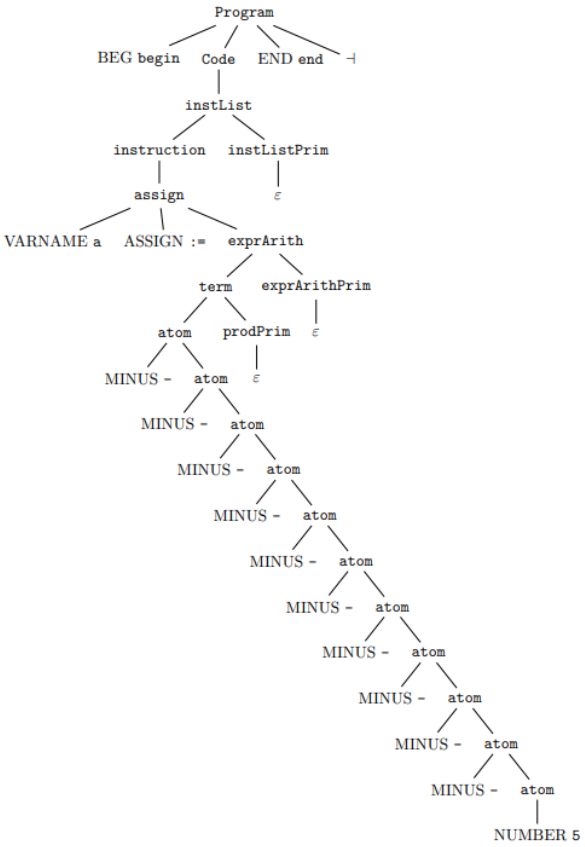

Figure 6: Derivation Tree of 11_testMinus.alg

4. **09_IncorrectSemicolon.alg** this file allows us to test the case where we put a semicolon before an *end*. This case is a bit tricky because it's not written explicitly in the given grammar that we can not have a semicolon before an *end*, *endwhile* , *endif* or *else*. Once the grammar has been modified, it was pretty obvious thanks to the rule 6 of the fig 7.

   As previously said, we show to the user where and what is the token creating the error. Indeed, because of the semicolon it means that we need to have an instruction (rule 5 fig 7).

```
mbounama@socrate24 ~/Compiler part 2/part2 $ java -jar dist/part2.jar test/SemicolonError.alg
Error at line : 3 column : 0 Received token  : END but expected  on of these : VARNAME/IF/WHILE/FOR/PRINT/READ/
```

The output is the leftmost derivation represented by a sequence of number corresponding to the used rule. The user also has the possibility to choose options for the compilation by using the command **java -jar part2.jar [OPTION(S)] [FILE]** where **OPTION(S)** is either:

1. **-v** gives us more informations about the grammar used rather than just having number referencing the rules.

2. **-wt** allow us to generate a latex file for the derivation tree. The user must specify the file name where the latex will be written. In order to obtain the **pdf** file, the user can either use the **make pdf** command or **lualatex some-file.tex**. It is important to notice that in order for **make pdf** to work, the latex file must be in the **more** folder.

3. **-v** and **-wt** can be used at the same time.

5 Firsts and Follows

In order to justify the action table, we made the First and Follow sets. We could have implemented it but it was easier to directly display it in our report.

States	First
<Program>	begin
<Code>	VarName If While For Print Read EPSILON
<InstList>	VarName If While For Print Read
<InstListPrim>	Semicolon EPSILON
<Instruction>	VarName If While For Print Read
<Assign>	VarName
<ExprArith>	VarName - Number (
<Term>	VarName - Number (
<Atom>	VarName - Number (
<ExprArith_prim>	+ - EPSILON
<prodPrim>	* / EPSILON
<addOp>	+ -
<multOp>	* /
<If>	if
<ifPrim>	endif else
<Cond>	VarName - Number (not
<condPrim>	or EPSILON
<andCond>	VarName - Number (not
<andCondPrim>	and EPSILON
<SimpleCond>	VarName - Number (not
<comp>	= <= >= < > / =
<While>	while
<For>	for
<Print>	print
<Read>	read

States	Follows
<Program>	
<Code>	endif endwhile else end
<InstList>	endif endwhile else end
<InstListPrim>	endif endwhile else end
<Instruction>	Semicolon endif endwhile else end
<Assign>	Semicolon endif endwhile else end
<ExprArith>	= <= >= < > / = Semicolon endif endwhile else end) by to do OR then and
<Term>	+ - Semicolon endif endwhile else end) by to do= <= >= < > / = OR then and
<Atom>	* / + - Semicolon endif endwhile else end) by to do = <= >= < > / = OR then and
<ExprArith_prim>	= <= >= < > / = Semicolon endif endwhile else end) by to do OR then and
<prodPrim>	+ - Semicolon endif endwhile else end) by to do = <= >= < > / = OR then and
<addOp>	- Varname Number (
<multOp>	- Varname Number (
<If>	Semicolon endif endwhile else end
<ifPrim>	Semicolon endif endwhile else end
<Cond>	then do
<condPrim>	then do
<andCond>	or then do
<andCondPrim>	or then do
<SimpleCond>	and or then do
<comp>	- Varname Number (
<While>	Semicolon endif endwhile else end
<For>	Semicolon endif endwhile else end
<Print>	Semicolon endif endwhile else end
<Read>	Semicolon endif endwhile else end

6 Action table

The action table describes what actions the parser must perform (either produce or match),depending on the look-ahead and the top of the stack. Each number is the rule used represented in our modified grammar (7). We used First and Follow set in order to produce the action table. In order to facilitate the visibility, we only put the actions possible for each state.

State	begin
<Program>	1

State	Varname	if	while	for	print	read	endif	endwhile	else	end
<Code>	3	3	3	3	3	3	2	2	2	2

State	Varname	if	while	for	print	read
<InstList>	4	4	4	4	4	4

State	Semicolon	endif	endwhile	else	end
<InstListPrim>	5	6	6	6	6

State	Varname	if	while	for	print	read
<Instruction>	7	8	9	10	11	12

State	Varname
<Assign>	13

State	Varname	-	Number	(
<ExprArith>	14	14	14	14

State	Varname	-	Number	(
<Term>	15	15	15	15

State	Varname	-	Number	(
<Atom>	16	17	18	19

State	+	-	=	<=	>=	<	>	/ =	by	to	do	)
<ExprArth_prim>	20	20	21	21	21	21	21	21	21	21	21	21

State	and	or	then	Semicolon	endif	endwhile	else	end
<ExprArth_prim>	21	21	21	21	21	21	21	21

State	*	/	=	<=	>=	<	>	/ =	by	to	do	)	end
<prodPrim>	22	22	23	23	23	23	23	23	23	23	23	23	23

State	+	-	not	and	or	then	Semicolon	endif	endwhile	else
<prodPrim>	23	23	23	23	23	23	23	23	23	23

State	+	-
<addOp>	24	25

State	*	/
<multOp>	26	27

State	if
<If>	28

State	endif	else
<IfPrim>	29	30

State	Varname	-	(	not	Number
<Cond>	31	31	31	31	31

State	or	then	do
<CondPrim>	32	33	33

State	Varname	-	(	not	Number
<andCond>	34	34	34	34	34

State	and	or	then	do
<andCondPrim>	35	36	36	36

State	Varname	-	(	not	Number
<SimpleCond>	37	37	37	38	37

State	=	>=	>	<=	<	/ =
<comp>	39	40	41	42	43	44

State	while
<While>	45

State	for
<For>	46

State	print
<Print>	47

State	read
<Read>	48

7 Conclusion

In this second part, we learned more about the implementation of a compiler and how it analyzes and detects grammar errors. One of the difficulties was to transform the ambiguous grammar to a non-ambiguous one even if the algorithm was given in classes. Some implicit rules had to be understood in order to have a functional grammar (like the rule 5/6 fig 7). We also could not check if our grammar was correctly define. However, tests and derivation trees gives us confirmation about our implementation. This part helped us to have a better understanding of how a parser works.

```
/*  1 */"<Program> -> begin <Code> end",
/*  2 */"<Code> -> EPSILON",
/*  3 */"<Code> -> <InstList>",
/*  4 */"<InstList> -> <Instruction> <InstListPrim>",
/*  5 */"<InstListPrim> -> SEMICOLON <instList>",
/*  6 */"<InstListPrim> -> EPSILON",
/*  7 */"<Instruction> -> <Assign>",
/*  8 */"<Instruction> -> <If>",
/*  9 */"<Instruction> -> <While>",
/*  10 */"<Instruction> -> <For>",
/* 11 */"<Instruction> -> <Print>",
/* 12 */"<Instruction> -> <Read>",
/* 13 */"<Assign> -> [VarName] := <ExprArith>",
/* 14 */"<ExprArith> -> <Term> <ExprArith_prim>",
/* 15 */"<Term> -> <Atom> <prodPrim>",
/* 16 */"<Atom> -> <MINUS> <Atom>",
/* 17 */"<Atom> -> <VarName>",
/* 18 */"<Atom> -> <NUMBER>",
/* 19 */"<Atom> -> <LEFT_PARENTHESIS> <ExprArith> <RIGHT_PARENTHESIS>",
/* 20 */"<ExprArith_prim> -> <AddOp> <Term> <ExprArith_prim>",
/* 21 */"<ExprArith_prim> -> EPSILON",
/* 22 */"<prodPrim> -> <MultOp> <Atom> <prodPrim>",
/* 23 */"<prodPrim> -> EPSILON",
/* 24 */"<addOp> -> +",
/* 25 */"<addOp> -> -",
/* 26 */"<multOp> -> *",
/* 27 */"<multOp> -> /",
/* 28 */"<If> -> if <cond> then <code> <ifPrim> ",
/* 29 */"<ifPrim> -> endif",
/* 30 */"<ifPrim> -> else <Code> endif",
/* 31 */"<Cond> -> <andCond> <condPrim>",
/* 32 */"<condPrim> -> or <andCond> <condPrim>",
/* 33 */"<condPrim> -> EPSILON",
/* 34 */"<andCond> -> <simpleCond> <andCondPrim>",
/* 35 */"<andCondPrim> -> and <SimpleCond> <andCondPrim>",
/* 36 */"<andCondPrim> -> EPSILON",
/* 37 */"<simpleCond> -> <exprArith> <comp> <exprArith>",
/* 38 */"<simpleCond> -> not <simpleCond>",
/* 39 */"<comp> -> =",
/* 40 */"<comp> -> >=",
/* 41 */"<comp> -> >",
/* 42 */"<comp> -> <=",
/* 43 */"<comp> -> <",
/* 44 */"<comp> -> /=",
/* 45 */"<while> -> while <cond> do <code> endwhile",
/* 46 */"<For> -> for [VarName] from <exprArith> by <exprArith> to <exprArith> do <code> endwhile",
/* 47 */"<Print> -> print ( <VarName> )",
/* 48 */"<Read> -> read ( <VarName> )",
```

Figure 7: Modified Grammar