INFO-F-403 Introduction to Language Theory and Compilation

Project Part 2 : Parser

Assistants :
Raphael Berton
Sarah Winter

Salma El Gueddari
Naim Sassine

23 Octobre 2019

1 Introduction

The aim of this project is to design and write a compiler for ALGOL-0, a variant of the classical Algol68 language. This second part of the project consisted in implementing a recursive descent, LL(1) parser for ALGOL-0 and write the correspondent derivation tree.

2 Grammar

First, we had to modify the grammar to get a LL(1) grammar. In order to obtain this modified grammar, we had to remove all the unproductive and unreachable variables, remove ambiguity, left factorize all the variables and finally remove all the left recursion.

2.1 Unproductive variables

$$S_0 = \{\emptyset\}$$

$$S_1 = \{[Print, Read, Code, ExprArith, Op, BinOp, Comp\}$$

$$S_2 = S_1\ U\ \{Program, Assign, Instruction, SimpleCond, For\}$$

$$S_3 = S_2\ U\ \{Instlist, Cond\}$$

$$S_4 = S_3\ U\ \{if, While\}$$

We can see that there are no unproductive symbols in this grammar.

2.2 Unreachable variables

$$S_1 = \{Program\}$$

$$S_2 = S_1\ U\ \{Code\}$$

$$S_3 = S_2\ U\ \{Instlist\}$$

$$S_4 = S_3\ U\ \{Instruction\}$$

$$S_5 = S_4\ U\ \{if, Assign, While, For, Print, Read\}$$

$$S_6 = S_5\ U\ \{ExprArith, Cond\}$$

$$S_7 = S_6\ U\ \{op, Binop, SimpleCond\}$$

$$S_8 = S_7\ U\ \{Comp\}$$

As you can see every variable is reachable.

2.3 Left factoring

We simply apply the algorithm presented in the course. The rules that have to change are :

if :
<if> –> if (<Cond>) then [endLine] <Code> <ifTail>
<ifTail> –> endif
<ifTail> –> else [endLine] <Code> endif

InstList :
<InstList> –> <Instruction> <Ints>
<Ints> –> ; <InstList>
<Ints> –> EPSILON

2.4 Left recursion

There is two cases where we had to use the left recursion algorithm for this grammar. On the one hand, there is the *ExprArith* variable, and on the other hand, the *Cond* variable.For this part, we had to be careful. But only applying the algorithm might cause some problems. We had to first remove the ambiguity by taking into account the priority and the associativity of the operators and after that, apply the left recursion algorithm.

Previous grammar for *ExprArith* :
<ExprArith> –> [VarName]
<ExprArith> –> [Number]
<ExprArith> –> (<ExprArith>)
<ExprArith> –> -<ExprArith>
<ExprArith> –> <ExprArith> <Op> <ExprArith>
<Op> –> +
<Op> –> -
<Op> –> *
<Op> –> /

After ambiguity removal :
<ExprArith> –> <ExprArith> + <ProdEx>
<ExprArith> –> <ExprArith> - <ProdEX>
<ExprArith> –> <Prod>
<ProdEX> –> <ProdEX> * <ProdAtom>
<ProdEx> –> <ProdEX> / <ProdAtom>
<ProdEx> –> <ProdAtom>
<ProdAtom> –> [VarName]

<ProdAtom> –> [Number]

<ProdAtom> –> (<ExprArith>)

<ProdAtom> –> - <ProdAtom>

After left recursion removal :

<ExprArith> –> <ProdEx> <ExprTail>

<ExprTail> –> <AddSous> <ProdEx> <ExprTail>

<ExprTail> –> EPSILON

<ProdEx> –> <ProdAtom> <ProdTail>

<ProdTail> –> <ProdDiv> <ProdAtom> <ProdTail>

<ProdTail> –> EPSILON

<ProdAtom> –> (<ExprArith)

<ProdAtom> –> - <ProdAtom>

<ProdAtom> –> [VARNAME]

<ProdAtom> –> [NUMBER]

<AddSous> –> +

<AddSous> –> -

<ProdDiv> –> *

<ProdDiv> –> /

Previous grammar for *Cond* <Cond> –> <Cond> <BinOp> <Cond>

<Cond> –> not <SimpleCond>

<Cond> –> <SimpleCond>

<SimpleCond> –> <ExprArith> <Comp> <ExprArith>

<BinOp> –> and

<BinOp> –> or

After ambiguity removal :

<Cond> –> <Cond> or <andExp>

<Cond> –> <andExp>

<andExp> –> <andExp> and <Condis>

<andEXp> –> <Condis>

<Condis> –> not <Condis>

<Condis> –> <ExprArith> <Comp> <ExprArith>

After left recursion removal :

<Cond> –> <andExp> <orExp>

<orExp> –> or <andExp> <orExp>

<orExp> –> EPSILON

<andExp> –> <Condis> <CondTail>

<CondTail>–> and <Condis> <CondTail>

<CondTail>–> EPSILON

<Condis> –> not <Condis>

<Condis> –> <ExprArith> <Comp> <ExprArith>

After these steps we finally get our LL(1) grammar.

2.5 Modified grammar

1. <Program> –> begin <Code> end
2. <Code> –> EPSILON
3. <Code> –> <InstList>
4. <InstList> –> <Instruction> <Ints>
5. <Ints> –> ; <InstList>
6. <Ints> –> 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> –> <ProdEx> <ExprTail>
15. <ExprTail> –> <AddSous> <ProdEx> <ExprTail>
16. <ExprTail> –> EPSILON
17. <ProdEx> –> <ProdAtom> <ProdTail>
18. <ProdTail> –> <ProdDiv> <ProdAtom> <ProdTail>
19. <ProdTail> –> EPSILON
20. <ProdAtom> –> (<ExprArith)
21. <ProdAtom> –> - <ProdAtom>
22. <ProdAtom> –> [VARNAME]
23. <ProdAtom> –> [NUMBER]

24. <AddSous> –> +
25. <AddSous> –> -
26. <ProdDiv> –> *
27. <ProdDiv> –> /
28. <if> –> if <Cond> then <Code> <ifTail>
29. <ifTail> –> endif
30. <ifTail> –> else <Code> endif
31. <Cond> –> <andExp> <orExp>
32. <orExp>–> or <andExp> <orExp>
33. <orExp>–> EPSILON
34. <andExp> –> <Condis> <CondTail>
35. <CondTail>–> and <Condis> <CondTail>
36. <CondTail>–> EPSILON
37. <Condis> –> not <Condis>
38. <Condis> –> <ExprArith> <Comp> <ExprArith>
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])

You cand find the grammar in doc/grammar/grammar.txt.

3 Action Table, First and Follow sets

We were asked to compute the action table. to compute this table we needed first to compute the First and Follow sets.

3.1 First sets

The First(X) consists of all the terminals that begin the strings derivable from a grammar symbol X .

First(Condis)=[not, NUMBER, (, -, VARNAME]

First(Instruction)=[if, print, VARNAME, read,For, while]

First(AddSous)=[+, -]

First(Comp)=[>, >=, =, <,<=, /=]

First(InstList)=[if, print, VARNAME, read, For, while]

First(Read)=[read]

First(Code)=[if, print, VARNAME, read,For, while,EPSILON]

First(ExprArith)=[NUMBER, (, -, VARNAME]

First(While)=[while]

First(if)=[if]

First(andExp)=[not, NUMBER,(, -, VARNAME]

First(For)=[For]

First(Cond)=[not, NUMBER, (, -, VARNAME]

First(ProdTail)=[/,*,EPSILON]

First(Program)=[begin]

First(ProdDiv)=[/, *]

First(orExp)=[or,EPSILON]

First(Assign)=[VARNAME]

First(ifTail)=[else, endif]

First(Print)=[print]

First(ProdEx)=[NUMBER, (, -, VARNAME]

First(CondTail)=[and, EPSILON]

First(ExprTail)=[+, -, EPSILON]

First(ProdAtom)=[NUMBER, (, -, VARNAME]

First(Ints)=[;, EPSILON]

3.2 Follows sets

The Follow(X) consist of all the terminals that can appear immediately to the right of Non-Terminal X.

Follow(Condis)=[or, and, do,then]

Follow(Instruction)=[endwhile, ;,else, endif, end]

Follow(AddSous)=[NUMBER, (, -, VARNAME]

Follow(Comp)=[NUMBER, (, -, VARNAME]

Follow(InstList)=[endwhile, else, endif, end]

Follow(Read)=[endwhile, ;,else, endif, end]

Follow(Code)=[endwhile, else, endif, end]

Follow(While)=[endwhile, ;,else, endif, end]

Follow(if)=[endwhile, ;,else, endif, end]

Follow(andExp)=[or, do,then]

Follow(For)=[endwhile, ;,else, endif, end]

Follow(Cond)=[do, then]

Follow(ProdDiv)=[NUMBER, (, -, VARNAME]

Follow(orExp)=[do, then]

Follow(Assign)=[endwhile, ;,else, endif, end]

Follow(ifTail)=[endwhile, ;,else, endif, end]

Follow(Print)=[endwhile, ;,else, endif, end]

Follow(CondTail)=[or, do,then]

Follow(Ints)=[endwhile, else, endif, end]

Follow(ExprArith)=[or, endwhile, and, ;, <,),then, <=, endif, >, >=, =, to, do, else, /=, end, by]

Follow(ExprTail)=[or, endwhile, and, ;, <,),then, <=, endif, >, >=, =, to, do, else, /=, end, by]

Follow(ProdAtom)=[/, or, endwhile, and, ;, -, <,),then, <=, endif, >, >=, =, to, +, do, *, else, /=, end, by]

Follow(ProdTail)=[or, endwhile, and, ;, -, <,),then, <=, endif, >, >=, =, to, +, do, else, /=, end, by]

Follow(ProdEx)=[or, endwhile, and, ;, -, <,), EPSILON, then, <=, endif, >, >=, =, to, +, do, else, /=, end, by]

3.3 Action table

Please consult the Action table in the corespondent directory /doc/grammar/ActionTable.xlsx

4 Code

In this part of the report we will provide an explanation of our code, how he thought it through and how it works.

4.1 Parser

We coded a recursive descent parser in this class. Based on our action table, we implemented a function for each and every line of the action table, where the function tries to match the next token with the possible symbols and derives the rules accordingly. For example, when trying to derive <ProdAtom>, the parser examines the next token and derives the rule according to the action table. The derived rule is then inserted in a ArrayList that will contain the list of rules that were used through the deviations.

At the end of the class, we implemented different methods to print out the rules, or print out a more explicit definition of the rules.

4.2 Parse Tree computation

Our parse tree is represented using the ParseTree class, which was given by the assistants, and represents a tree as a root label and the list of its children which are themselves parse trees.

We used the parser class to implement the represented parse tree. In fact the parse tree is computed during the parsing. As we said before, we implemented a function for each non-terminal of the grammar, and each function returns the parse tree rooted in the corresponding non-terminal.

At the end of the parsing, the program has built the entire parse tree and the main function (in Main.java) outputs it to a .tex file given as argument.

4.3 Main

Our Main class implements the main function that will output the requested information. The main method is also in charge of analyzing the options given by the user.
In fact, if the user does not add any option, the main function will only output the set of rules (the leftmost derivation of the input string).
If the user types "-v" before the input file, the main will output the rules with more details explicitly describing the rules which are used.
If the user types "-wt" and adds a .tex file to the arguments (-wt needs to be followed by a .tex file to work, order here matters), the function will not only print out the rules that were used, but it will also write down the parse tree generated by the program in the .tex file passed in argument.

4.4 ParserException

This class is used to handle an exception that can occur when the current token does not match the expected token or the rules of the grammar are not respected. We decided to create a customized exception, because there is no exception that can handle these two cases and it's more user-friendly. It outputs the read token and the token it expected when a problem occurs while parsing.

4.5 Rules

This class is used to store the rules. It contains all the "verbose" rules we print when the user asks for them by passing the -v parameter. We stored the rules in an HashMap.

4.6 Test

We created several tests to pass through all the rules of the grammar in order to check if the associativity and the priorities were respected. There is also some failed test to check which error is printed.

5 Conclusion

To conclude the second part of the project, we can say that we first applied different transformations to our grammar to turn it into an LL(1) grammar that we verified by building a conflict free action table. Then we coded a recursive descent parser using java that will parse the input file after it has been scanned (by the program built in the first part of the project) and will build a parse tree that it will write down in a .tex file given as argument to the program.