alCOl
Introduction to language theory and compiling
Project – Part 1

Gilles GEERAERTS Raphaël BERTHON Sarah WINTER

October 1, 2021

The more I see of it, the more unhappy I become. E. W. Dijkstra, To the EDITOR ALGOL 68 Mathematische Centrum

1 Introduction

In this project, you are requested to design and write a compiler for alCOl, a simple imperative language. The grammar of the language is given in Figure 1 (page 3), where reserved keywords have been typeset using `typewriter font`. In addition, [VarName] and [Number] are lexical units, which are defined as follows. A [VarName] identifies a variable, which is a string of digits and letters, starting with a letter (this is case sensitive). A [Number] represents a numerical constant, and is made up of a string of digits only. The minus sign can be generated using rule [16].

Finally, comments are allowed in alCOl. There are two kinds of comments: Short comments, which are all the symbols appearing after a lowercase `co` up to the end of the line, and long comments, which are all the symbols occurring between two uppercase `CO` keywords. Nesting comments (`CO CO Comment CO CO`) is forbidden. Observe that comments do not occur in the rules of the grammar: they must be ignored by the scanner, and will not be transmitted to the parser.

Figure 2 shows an example of alCOl program.

2 Assignment - Part 1

In this first part of the assignment, you must produce the *lexical analyzer* of your compiler, using the JFlex tool reviewed during the practicals.

Please adhere strictly to the instructions given below, otherwise we might be unable to grade your project, as automatic testing procedures will be applied.

The lexical analyzer will be implemented in JAVA 1.8. It must recognise the different lexical units of the language, and maintain a symbol table. To help you, several JAVA classes are provided on the UV:

- The `LexicalUnit` class contains an enumeration of all the possible lexical units;
- The `Symbol` class implements the notion of token. Each object of the class can be used to associate a value (a generic Java `Object`) to a `LexicalUnit`, and a line and column number (position in the file). The code should be self-explanatory. If not, do not hesitate to ask questions to the teacher, or to the teaching assistants.

You must hand in:

- A PDF report containing all regular expressions, and presenting your work, with all the necessary justifications, choices and hypotheses, as well as descriptions of your example files. Such report will be particularly useful to get you partial credit if your tool has bugs.
  - *Bonus:* Everyday programming languages can handle nested comments. Explain what technical difficulties arise if you are to handle those. More generally, personal initiative is encouraged for this project: do not hesitate to explain why some features would be hard to add at this point of the project, or to add relevant features to alCOl. Ask us before, just to check that the feature in question is both relevant and doable.
- The source code of your lexical analyzer in a JFlex source file called `LexicalAnalyzer.flex`;
- The alCOl example files you have used to test your analyzer;
- All required files to evaluate and compile your work (like a `Main.java` file calling the lexical analyzer, etc, but also files we provided like `LexicalAnalyzer.java`, `LexicalUnit.java`, etc).

You must structure your files in five folders:

- `doc` contains the JAVADOC and the PDF report. Note that the documentation of your code will be taken into account for the grading, especially for Parts 2 and 3, so Part 1 is a good occasion to setup JAVADOC for your project;
- `test` contains all your example alCOl files. It is necessary to provide relevant example files of your own;
- `dist` contains an executable JAR called `part1.jar`;
- `src` contains your source files;
- `more` contains all other files.

Your implementation must contain:

1. the provided classes `LexicalUnit` and `Symbol`, *without modification*;
2. an executable public class `Main` that reads the file given as argument and writes on the standard output stream the sequence of matched lexical units and the content of the symbol table. More precisely, the format of the output must be:
   (a) First, the sequence of matched lexical units. You must use the *toString()* method of the provided Symbol class to print individual tokens;
   (b) Then, the word *Variables*, to clearly separate the symbol table from the sequence of tokens;
   (c) Finally, the content of the symbol table, formatted as the sequence of all recognised variables, in lexicographical (alphabetical) order. There must be one variable per line, together with the number of the line of the input file where this variable has been encountered for the first time (the variable and the line number must be separated by at least one space).

The command for running your executable must be:

```
java -jar part1.jar sourceFile
```

[1] <Program> → `begin` <Code> `end`
[2] <Code> → ε
[3] → <InstList>
[4] <InstList> → <Instruction>
[5] → <Instruction> `;` <InstList>
[6] <Instruction> → <Assign>
[7] → <If>
[8] → <While>
[9] → <For>
[10] → <Print>
[11] → <Read>
[12] <Assign> → [VarName] `:=` <ExprArith>
[13] <ExprArith> → [VarName]
[14] → [Number]
[15] → `(` <ExprArith> `)`
[16] → `-` <ExprArith>
[17] → <ExprArith> <Op> <ExprArith>
[18] <Op> → `+`
[19] → `-`
[20] → `*`
[21] → `/`
[22] <If> → `if` <Cond> `then` <Code> `endif`
[23] → `if` <Cond> `then` <Code> `else` <Code> `endif`
[24] <Cond> → `not` <Cond>
[25] → <SimpleCond>
[26] <SimpleCond> → <ExprArith> <Comp> <ExprArith>
[27] <Comp> → `=`
[28] → `>`
[29] → `<`
[30] <While> → `while` <Cond> `do` <Code> `endwhile`
[31] <For> → `for` [VarName] `from` <ExprArith> `by` <ExprArith> `to` <ExprArith> `do` <Code> `endfor`
[32] <Print> → `print(`[VarName]`)`
[33] <Read> → `read(`[VarName]`)`

Figure 1: The alCOI grammar.

```
CO Euclid's algorithm CO

begin
  read(a) ;
  read(b) ;
  while not b = 0 do
    c := b ;
    while a+1 > b do     co computation of modulo
      a := a-b
    endwhile ;
    b := a ;
    a := c
  endwhile ;
  print(a)
end
```

Figure 2: An example alCOl program.

For instance, on the following input:

```
read(b)
```

your executable must produce exactly, using the `toString()` method of the `Symbol` class, the following output for the sequence of tokens (an example for the symbol table is given hereunder):

```
token: read           lexical unit: READ
token: (              lexical unit: LPAREN
token: b              lexical unit: VARNAME
token: )              lexical unit: RPAREN
```

Note that the *token* is the matched input string (for instance `b` for the third token) while the *lexical unit* is the name of the matched element from the `LexicalUnit` enumeration (`VARNAME` for the third token).

Also, for the example in Figure 2, the symbol table must be displayed as:

```
Variables
a 4
b 5
c 7
```

An example input alCOl file with the expected output is available on Université Virtuelle to test your program. You will compress your folder (in the *zip* format—no *rar* or other format), **which is named according to the following regexp**: `Part1_Surname1(_Surname2)?.zip`, where `Surname1` and, if you are in a group, `Surname2` are the last names of the student(s) (in alphabetical order), and you will submit it on Université Virtuelle before **October, 23rd**. You are allowed to work in a group of maximum two students.