IMP
Introduction to language theory and compiling
Project – Part 1

Gilles GEERAERTS Marie VAN DEN BOGAARD Léo EXIBARD

October 3, 2017

1 Introduction

In this project, you are requested to design and write a compiler for IMP, a simple imperative language. The grammar of the language is given in Figure 1, 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 IMP: they are all the symbols occurring between `(*` and `*)` keywords. Nesting comments (`(* (* Comment *) *)`) 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 IMP program.

2 Assignment - Part 1

In this first part of the assignment, you must produce the *lexical analyser* 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 1.6. 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 Université Virtuelle platform:

- 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.

[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> → <Cond> <BinOp> <Cond>
[25] → `not` <SimpleCond>
[26] → <SimpleCond>
[27] <SimpleCond> → <ExprArith> <Comp> <ExprArith>
[28] <BinOp> → `and`
[29] → `or`
[30] <Comp> → `=`
[31] → `>=`
[32] → `>`
[33] → `<=`
[34] → `<`
[35] → `<>`
[36] <While> → `while` <Cond> `do` <Code> `done`
[37] <For> → `for` [VarName] `from` <ExprArith> `by` <ExprArith> `to` <ExprArith> `do` <Code> `done`
[38] → `for` [VarName] `from` <ExprArith> `to` <ExprArith> `do` <Code> `done`
[39] <Print> → `print(`[VarName]`)`
[40] <Read> → `read(`[VarName]`)`

Figure 1: The IMP grammar.

```
// Euclid's algorithm

begin
  read(a) ;
  read(b) ;
  while b <> 0 do
    c := b ;
    while a >= b do     // computation of modulo
      a := a-b
    done ;
    b := a ;
    a := c
  done ;
  print(a)
end
```

Figure 2: An example IMP program implementing Euclid's algorithm. The inner loop computes `a` modulo `b`. Note the use of semicolons as *instruction separators* and not *instruction terminators* as in C. That is, semicolon *separate consecutive instructions* but they are not present after the last instruction of each block.

You must hand in:

- A PDF report containing all REs, and presenting your work, with all the necessary justifications, choices and hypotheses
  - *Bonus:* Everyday programming languages can handle nested comments. Explain what technical difficulties arise if you are to handle those, and suggest a way to overcome the problem.
- The source code of your lexical analyzer in a JFlex source file called `LexicalAnalyzer.flex`;
- The IMP example files you have used to test your analyser;
- All required files to evaluate your work (like a `Main.java` file calling the lexical analyser, etc).

You must structure your files in four folders:

- `doc` contains the JAVADOC and the PDF report;
- `test` contains all your example files;
- `dist` contains an executable JAR;
- `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 *Identifiers*, 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 identifiers, in lexicographical (alphabetical) order. There must be one identifier per line, together with the number of the line of the input file where this identifier has been encountered for the first time (the identifier and the line number must be separated by at least one space).

The command for running your executable must be:

```
java -jar yourJarFile.jar Main sourceFile
```

For instance, on the following input:

```
read(b)
while(b) do
```

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
token: while     lexical unit: WHILE
token: (         lexical unit: LPAREN
token: b         lexical unit: VARNAME
token: )         lexical unit: RPAREN
token: do        lexical unit: DO
```

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:

```
Identifiers
a 2
b 3
c 5
```

An example input 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) and you will submit it on the Université Virtuelle before **October, 30th**. You are allowed to work in group of maximum two students.