FORTRESS
Introduction to language theory and compiling
Project – Part 1

Gilles GEERAERTS Léonard BRICE Sarah WINTER

October 3, 2022

FORTRAN, "the infantile disorder", by now nearly 20 years old, is hopelessly inadequate for whatever computer application you have in mind today: it is too clumsy, too risky, and too expensive to use. (Edsger W. Dijkstra, SIGPLAN Notices, Volume 17, Number 5 (1982))

1 Introduction

In this project, you are requested to design and write a compiler for FORTRESS, a simple imperative language (similar to FORTRAN). The grammar of the language is given in Figure 1 (page 2), where reserved keywords have been typeset using `typewriter font`. In addition, [VarName], [ProgName] and [Number] are lexical units, which are defined as follows. A [VarName] identifies a variable, which is a string of digits and *lowercase* letters, starting with a letter. A [ProgName] identifies the program name, which is a string of digits and letters, starting with an uppercase letter but *not* entirely uppercase (*e.g.* `FaCTORIAL`, although not very pretty, is accepted, `FactorialPrgm` also, but `FACTORIAL` is not, so that it is not confused with a keyword). Finally, a [Number] represents a numerical constant, and is made up of a string of digits only, without leading zeroes (*e.g.* `12` is accepted, but `012` is rejected). The minus sign can be generated using rule [13].

Comments are allowed in FORTRESS. There are two kinds of comments: Short comments, which are all the symbols appearing after `::` and up to the end of the line. Long comments, which 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 FORTRESS program.

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

Figure 1: The FORTRESS grammar.

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 must be implemented in JAVA. It must recognize 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.

You must hand in:

- A PDF report presenting your work: In particular it must contain your regular expressions as well as descriptions of your example files. Such report will be particularly useful to get you partial credit if your tool has bugs. Do not forget to put your names on the report!
- The code of your lexical analyzer in a JFlex source file called `LexicalAnalyzer.flex`;
- The FORTRESS 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 `LexicalUnit.java`, etc.).

Bonus: Some programming languages can handle nested comments. Explain in your PDF report what technical difficulties arise if you are to handle those, and suggest a way to overcome the problem. Make your answer specific to FORTRESS. 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 FORTRESS. Ask the teaching assistants before, just to check that the feature in question is both relevant and doable.

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 FORTRESS files. It is necessary to provide some relevant example files of your own;
- `dist` contains an executable JAR called `part1.jar`;
- `src` contains your source files;
- `more` contains all other files.

Furthermore, provide a `Makefile` for your project. There is an example Makefile on the UV.

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 recognized 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
```

You must 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 team, `Surname2` are the last names of the student(s) (in alphabetical order), and you must submit it on Université Virtuelle before **24. October**. You are allowed and encouraged to work in a team of maximum two students.

3 Example

```
BEGIN Factorial

%% Compute the factorial of a number.
   If the input number is negative, print -1. %%

  READ(number) ,              :: Read a number from user input
  result := 1 ,

IF (number > -1) THEN
  WHILE number > 0 DO
    result := result * number ,
    number := number - 1 ,    :: decrease number
  END ,
  ELSE                        :: The input number is negative
  result := -1 ,
END ,
PRINT(result) ,
END
```

Figure 2: An example FORTRESS 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
number 6
result 7
```

An example input FORTRESS file with the expected output is available on Université Virtuelle to test your program.

4 Frequently Asked Questions

Here are some questions that we have gotten in the previous years which might help you during your project. If you have further questions please ask in the forum on the UV or via email.

- Q: What should happen if the FORTRESS file can not be correctly tokenized?

  A: Throw a (useful) error message such as “Unknown symbol detected” “Long comment not closed” for example. You can assume that you will not encounter a FORTRESS file that has nested comments.

- Q: What about whitespaces?

  A: Whitespaces are not necessary (but nice for readability) and must be ignored. For example

```
   IF abbb END END DO
```

```
    IFabbbENDENDDO
```

```
    IFabbb


    END
         END DO
```

  must all yield the same output

```
    token: IF           lexical unit: IF
    token: abbb         lexical unit: VARNAME
    token: END          lexical unit: END
    token: END          lexical unit: END
    token: DO           lexical unit: DO
```

- Q: What is the use of a `Makefile`?

  A: You need to provide a `Makefile` because your executable might not run on our machine, because of different JAVA versions (unavoidable because some use Windows, Linux, MacOS). A working `Makefile` allows us to quickly produce an executable from your code. It takes a considerable amount of time if we need to figure out how to turn your code into an executable by hand.