Université Libre de Bruxelles

INFO-F403 : Introduction to language theory and compiling

Computer Science Section

Building a FORTR-S compiler : Part 1

Professor:
Gilles Geeraerts

Students :
Miró-Manuel Matagne
Linh Tran Ngoc

Academic Year 2020 - 2021

Contents

1 Introduction

The goal of this project, as part of the course Introduction to language theory and compiling, is to build a compiler for the FORTR-S language. This first part consists in coding the scanner of this compiler. After describing the utilization of the scanner, an explanation on the different regular expressions and all the test files that our group chose is given.

2 Utilization

In order to get the tokens and symbol table related to a certain file, one should run the command :

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

inside the *dist* folder, where `sourceFile` is the file we want to scan.

If the user wishes to recompile the project, he can run the *make* command (if on a UNIX environment) in the root repository.

In order to be sure our lexical analyzer worked properly, we put in place a testing system in order to rapidly and automatically make sure each regular expression is still properly working after we made some changes. More details on this is described in section 4 of this report.

2.1 Main file

The Main.java file is located in the *src* folder. This Java class checks that the number of arguments entered is correct, and then creates an instance of the LexicalAnalyzer Java class, which is generated by our LexicalAnalyzer.flex file. The Main class receives each token one by one, using the *nextToken* function, as we inserted the line `%function nextToken` in the lexer file which replaces the initial *yylex* function provided by *JFlex* by *nextToken*. It then prints them out using the provided `toString()` function of the Symbol class.

Additionally, it also keeps a symbol table up do date : everytime a new variable is detected, it is stocked in a TreeMap with the number of the line it was read in. We chose the TreeMap to store this symbol table as we need to print out the variables in alphabetical order, and the TreeMap sorts alphabetically on the keys automatically. The key of each element in the symbol table TreeMap is the name of the variable, and the value is the corresponding line number. After printing out all the tokens detected in the file, the Main class prints out the symbol table.

Our group made the hypothesis that the entered code can contain errors, and we have chosen to detect them as much as possible (more details are given on this during the explanations given on the error regular expressions in section 3). When an error is detected, we print out an error message for this token, with a very brief description of the error. All the non erroneous tokens, before or after this error, are still outputted normally.

2.2 Lexical Analyzer file

The lexer is based on two states. The first one is by default the Initial state (called `YYINITIAL`) followed by the Comment state (called `COMMENT`). The initial state detects regular expressions that will be explained down below, and whenever it detects `/*`, it enters the Comment state. Note that the lexer detects nested comments by introducing a counter `commentDepth` that increments every time it meets `/*`, and inversely, it decrements if `*/` is met. Thus, `commentDepth` will indicate on which "level of comment" the lexer will be, and in consequence, it permits to ignore the nested comments. However, if a comment or a nested comment is not closed by a `*/` (which means `commentDepth` is not equal to 0 at the end of the file), an error will occur, a string "COMMENT ERROR" will be printed at the end of the program.

Furthermore, for certain detected regular expressions, the lexical analyzer will create a Symbol object (with the *LexicalUnit* it corresponds to, the position in the file and the string representing the symbol), that will be received by the Main class, and then printed like explained in the previous section. At the end of the program, a last Symbol is created, with the `EOS` *LexicalUnit*, so that the Main class knows when the end of the file is reached.

3 Regular expressions

Right below are listed the regular expressions in the scanner :

```
AlphaUpperCase  = [A-Z]
AlphaLowerCase  = [a-z]
Alpha           = {AlphaUpperCase}|{AlphaLowerCase}
Numeric         = [0-9]
AlphaNumeric    = {Alpha}|{Numeric}
AlphaLowNumeric = {AlphaLowerCase}|{Numeric}

Space           = "\t"|" "
EndOfLine       = "\r"?"\n"

Number          = (([1-9][0-9]*)|0)

VarName         = {AlphaLowerCase}{AlphaLowNumeric}*
ProgName        = {AlphaUpperCase}+{AlphaLowNumeric}+{AlphaNumeric}*

InlineComment   = "//".*

VarnameError    = {AlphaLowerCase}+{AlphaLowNumeric}*{AlphaUpperCase}+{AlphaNumeric}*
ProgNameError   = {AlphaUpperCase}+
NumError        = {Numeric}+{Alpha}+{AlphaNumeric}*
ZeroError       = (0){Numeric}+
.

"BEGINPROG"
"ENDPROG"

":="

"+"
"-"
"*"
"/"

"("
")"
","

"=="
">"

"PRINT"
"READ"

"IF"
"THEN"
"ENDIF"
"ELSE"
"WHILE"
"DO"
"ENDWHILE"

"*/"
"/*"
```

For this first part of the project, we have to build regular expressions to detect the existing expressions of FORTR-S.

The first regular expressions we need for this lexer are related to numbers and letters, as well as detecting if the letters are lowercase or uppercase. The regular expressions are pretty self explanatory.

Next, we detect spaces and end of lines with two very basic regular expressions.

The regular expression `Number` detects numbers that do not start with a zero (not to be confused with the regular expressions `Numeric`, which detects a single numeric character).

Then, we detect regular expressions corresponding to different types of variables :

- `VarName` : detects the lexical unit `VARNAME` : strings of digits and lowercase letters, starting by a lowercase letter (this condition is ensured by the `{AlphaLowerCase}` regular expression contained in the `VarName` regular expression) and followed by lowercase letters or numbers (hence the `{AlphaLowNumeric}` regular expression contained in the `VarName` regular expression).

- `ProgName` : detects the lexical unit `PROGNAME` : strings of digits and letters, starting by an uppercase letter (this condition is ensured by the + symbol following the `{AlphaUpperCase}` regular expression contained in the `ProgName` regular expression) but not entirely uppercase (hence the + symbol following the `{AlphaLowNumeric}` regular expression contained in the `ProgName` regular expression).

The next one, `InlineComment` detects comments in one line starting by two slashes and ending at the end of the current line. Indeed, using the . regular expression ensures that the inline comment doesn't extend to multiple lines, since it corresponds to every symbol except \n.

Moreover, it was obvious that erroneous inputs could occur. Thus, our group considers that errors are possible. To try and recognize a maximal number of errors, five regular expressions were introduced :

- `VarnameError` : detects words starting with a lowercase letter but containing an uppercase letter inside it. This is considered an error as it cannot either be a `VARNAME` because it contains an uppercase letter, nor a `PROGNAME` because it does not start with an uppercase letter.

- `ProgNameError` : detects a word that only contains uppercase letters, which is an error because it cannot either be a `PROGNAME` nor a `VARNAME`.

- `NumError` : detects every expressions starting with a number (for example : *3test*), which we consider is an error.

- `ZeroError` : detects numbers starting with a zero, which are an error.

- `.` : detects any special character that can not be contained in any of the regular expressions (for example : *?*), which we consider is an error.

As a result, when the scanner detects errors in the input codes, it outputs the value and the type of error mentioned above. There is one more error detected in our project, not related directly to regular expressions : if the end of the file is reached with a multi-line comment opened but not closed, the user is notified with a *Comment error*. This situation occurs when the `commentDepth` variable is not equal to 0 at the end of the file.

The rest of the regular expressions we have used are very simple, they simply correspond to strings that we need to find in the FORTR-S file. These strings each correspond to their own Lexical Unit, except for the last 2 regular expressions (`*/` and `/*`). These are used in the `COMMENT` state, in order to detect the beginning of a multi-line comment or the end of a multi-line comment.

It is also important to talk about the order of appearance of the regular expressions in our code. Indeed, the `ProgNameError` regular expression has to be tested after all the regular expressions containing exclusively uppercase letters. For instance if the `ProgNameError` was before the `"IF"`, an IF in the file would be detected as an error.
In addition, the `.` regular expression in the `YYINITIAL` state has to be the last one to be tested among all the regular expressions of the scanner because for instance if the `.` was before the `VarName` regular expression, a word `x`, for example, in the file would be detected as an error.

It is necessary to keep in mind that the scanner activates the regular expression with the longest match. For example, if we encountered the word `12ab34`, because of the property we just mentioned, it will be detected as a `NumError`. If the scanner didn't work based on longest matching, this could have been detected as `Number : 12, VarName : ab, Number : 34`.

4 Testing

As mentioned previously, our group chose to implement a testing system to make sure all the regular expressions we have written work properly on our test files. In order to do that, we have created folders for each different group of regular expressions that need to be detected by our lexical analyzer and a folder containing entire FORTR-S codes inside the test folder. In each one of those, there is a *.fs* file and a *.output* file. The *.fs* file is the file used to test our lexical analyzer and the *.output* file is the correct output that is expected after analyzing the *.fs* file. Therefore, we have used the *diff* bash command in order to compare the output of our lexical analyzer on the *.fs* file to the *.output* file.
In order to use the testing system, the user can simply go to the *test* folder and enter one of the following commands to test a particular group of regular expressions :

- `make comments`
- `make comparisons`
- `make errors`
- `make instructions`
- `make names`
- `make operations`
- `make parenthesis`
- `make program`
- `make signsnumber`

The output of these commands will be a string. For example for the test related to comments, the output can be either *Comments : Test Successful* or *Comments : Test Failed*. The *diff* command also gives a very brief detail about where the differences between the 2 files are in case of a failure.

If the user wishes to test all the possible regular expressions at once (therefore run all the commands listed above), he can run the command `make test`.

Furthermore, we have also chosen to test complete FORTR-S files, mixing all the different regular expressions tested individually before. Indeed, even though these regular expressions are all detected properly in their individual test files, we thought it was necessary to apply our lexical analyzer to complete FORTR-S codes containing all the regular expressions tested individually. These are grouped in the folder FortrS, and as before, we have an option to compare the output of our lexical analyzer applied to the FORTR-S files with the expected output, using `make fortrs`. This command will notify the user if the outputs are correct for the 5 FORTR-S files we have created.

5 Conclusion

The scanner that we built fulfills the requested tasks for a FORTR-S compiler. Indeed, all expressions and words of the language FORTR-S are well identified. In addition, the hypothesis that errors can occur in the input codes is made, thus the scanner takes them in account due to specified regular expressions, and notifies the user whenever an error is encountered.

Finally, our group chose to implement a testing system to verify that the outputs of our scanner on different *.fs* files are concordant with the *.output* files we have created. It was a very useful way to make sure our scanner was always functional after changes.

Now that we have built a functioning scanner, we have all the cards in hand to start building a parser relying on it.