INFO-F403 - Project : Part1

Julien Baudru - N°000460130

October 2020

1 Introduction

The first part of this project consisted in building a lexical analyzer of a future compiler for the FORTR-S language.

2 Structure

The structure of the project is the following :

```
part1:
|
|---dist
|     |---part1.jar
|
|---doc
|     |---Javadoc
|     |     |--- ...
|     |
|     |---Rapport.pdf
|
|---more
|
|---src
|     |---LexicalAnalizer.java
|     |---LexicalAnalyzer.flex
|     |---LexicalUnit.java
|     |---Main.java
|     |---Symbol.java
|     |---SymbolTable.java
|
|---test
      |---Factorial_3.fs
      |---Fibonacci.fs
      |---Test.fs
```

Note : The file index.html as well as all other files of the type html constitute the JavaDoc of this project. To compile and execute this project the following instructions must be followed:

```
jflex src/LexicalAnalyzer.flex
javac -d bin -cp src/ src/Main.java
jar cfe dist/Part1.jar Main -C bin
java -jar dist/Part1.jar test/Factorial.fs
```

When the file *Main.java* is executed, a loop will iterate through all the tokens in the file until it encounters the symbol **EOS** (the symbol for the end of the file). The symbols are obtained thanks to the *nextToken()* function of the *LexicalAnalyzer.java* class, this class is generated thanks to the file *LexicalAnalyzer.flex* by executing the command below :

```
C:Part1> jflex LexicalAnalyzer.flex
```

In the class *LexicalAnalyzer.java*, the symbols are extracted from the tokens via a set of regular expressions detailed in the point 3 of this document. From the set of symbols received by the function *nextToken()*, a table of symbols will be created via the class *SymbolTable.java*

However, this symbol table still contains the line and block comments of the initial file, as these have to be ignored by the scanner, the function *DeleteComment()* from the class *SymbolTable.java* is called. This function scrolls through the symbol table and deletes everything between a starting X character and an ending Y character, for the line comments the signature of this function is :

```
st.DeleteComment("//", "\\n");
```

and for the block comments his signature is : .

```
st.DeleteComment("/*", "*/");
```

The choice to delete comments after the construction of the symbol table has been made to simplify the writing of regular expressions.

Moreover, this technique would allow to handle nested comments, in fact, as explained above, the function *DeleteComment()* browses the entries of the symbol table looking for the part to delete, so if we have a comment of the following type :

```
/*
Frist line
// Second line
Last line
/*
```

So we could simply call this function with the same signature as for block comments and the nested comment will be ignored as it should be.

3 Regular expressions

Lexical Unit	Regular expression
BEGINPROG	"BEGINPROG"
PROGNAME	({AlphaUpperCase}({AlphaUpperCase}*{AlphaLowerCase}*{Numeric}*)+)
ENDLINE	"\n"
ENDPROG	"ENDPROG"
COMMA	","
VARNAME	({AlphaLowerCase}({AlphaLowerCase}*{Numeric}*)+)
ASSIGN	":="
NUMBER	{Integer} $\|\|\{Decimal\}$
LPAREN	"("
RPAREN	")"
MINUS	"-"
PLUS	"+"
TIMES	"*"
DIVIDE	"/"
IF	"IF"
THEN	"THEN"
ENDIF	"ENDIF"
ELSE	"ELSE)"
EQ	"=="
GT	"¿"
WHILE	"WHILE"
DO	"DO"
ENDWHILE	"ENDWHILE"
PRINT	"PRINT"
READ	"READ"
EOS	*null*

Note : To allow more clarity during the reading, other regular expressions have been used to build the expressions above, here is how they were defined :

Alias for regular expression	Regular expression
{Numeric}	[0-9]
{Zero}	0
{ZeroPlus}	0[0-9]
{Integer}	([1-9][0-9]*) $\|\|\{Zero\}$
{Decimal}	{Integer}"."([0-9]+) $\|\|\{Zero\}$"."$([0-9]+)$
{AlphaUpperCase}	[A-Z]
{AlphaLowerCase}	[a-z]
{OnlyUpperCase}	{AlphaUpperCase}+

The expression {ZeroPlus} is used to detect cases where numbers contain unnecessary zeros such as **002** or **047**. If such a case is recognized then it is changed from *YYINITIAL* to *ZEROCASE* state in the *LexicalAnalyzer.flex* file, so these numbers are ignored by the scanner.

In the same logic, the expression *OnlyUpperCase* is used to detect symbols of type PROGNAME which are only in upper case, these will also be ignored by the scanner.

In addition to this, although it is not specified in the statement, the scanner is also able to recognize comma numbers because almost all programming languages are able to do so. This is why the regular expression allowing to recognize the lexical unit of type NUMBER is composed of the regular expressions {Integer} and {Decimal}.

4 Example of FORTR-S

4.1 Fibonacci program

The file given as an argument to our lexical analyzer is the following, this program print the n first numbers of the Fibonacci suite. This example was chosen because it uses almost all the types of lexical units provided by the FORTR-S language.

```
BEGINPROG Fibonacci
READ(n)
a := 0
olda := a
b := 1
i := 0
WHILE(n > i) DO
    olda := a
    a := b
    b := olda + b
    i := i + 1
    PRINT(a)
ENDWHILE
ENDPROG
```

4.2 Symbol table of Fibonacci program

Below you will find the symbol table returned by the lexical analyzer for the program *Fibonacci.fs*.

Note : To have exactly the same symbol table as below, the line separator of the file must be \n (i.e. LF) or \r (i.e. CR) but not both at the same time (i.e. CRLF).

```
token: BEGINPROG       lexical unit: BEGINPROG
token: Fibonacci       lexical unit: PROGNAME
token: \n              lexical unit: ENDLINE
token: READ            lexical unit: READ
token: (               lexical unit: LPAREN
token: n               lexical unit: VARNAME
token: )               lexical unit: RPAREN
token: \n              lexical unit: ENDLINE
token: a               lexical unit: VARNAME
token: :=              lexical unit: ASSIGN
token: 0               lexical unit: NUMBER
token: \n              lexical unit: ENDLINE
token: olda            lexical unit: VARNAME
token: :=              lexical unit: ASSIGN
token: a               lexical unit: VARNAME
token: \n              lexical unit: ENDLINE
token: b               lexical unit: VARNAME
token: :=              lexical unit: ASSIGN
token: 1               lexical unit: NUMBER
token: \n              lexical unit: ENDLINE
token: i               lexical unit: VARNAME
token: :=              lexical unit: ASSIGN
token: 0               lexical unit: NUMBER
token: \n              lexical unit: ENDLINE
token: WHILE           lexical unit: WHILE
token: (               lexical unit: LPAREN
token: n               lexical unit: VARNAME
token: >               lexical unit: GT
token: i               lexical unit: VARNAME
token: )               lexical unit: RPAREN
token: DO              lexical unit: DO
token: \n              lexical unit: ENDLINE
token: olda            lexical unit: VARNAME
token: :=              lexical unit: ASSIGN
token: a               lexical unit: VARNAME
token: \n              lexical unit: ENDLINE
```

```
token: a               lexical unit: VARNAME
token: :=              lexical unit: ASSIGN
token: b               lexical unit: VARNAME
token: \n              lexical unit: ENDLINE
token: b               lexical unit: VARNAME
token: :=              lexical unit: ASSIGN
token: olda            lexical unit: VARNAME
token: +               lexical unit: PLUS
token: b               lexical unit: VARNAME
token: \n              lexical unit: ENDLINE
token: i               lexical unit: VARNAME
token: :=              lexical unit: ASSIGN
token: i               lexical unit: VARNAME
token: +               lexical unit: PLUS
token: 1               lexical unit: NUMBER
token: \n              lexical unit: ENDLINE
token: PRINT           lexical unit: PRINT
token: (               lexical unit: LPAREN
token: a               lexical unit: VARNAME
token: )               lexical unit: RPAREN
token: \n              lexical unit: ENDLINE
token: ENDWHILE        lexical unit: ENDWHILE
token: \n              lexical unit: ENDLINE
token: ENDPROG         lexical unit: ENDPROG
token: \n              lexical unit: ENDLINE

Variables :
a 3
b 5
i 6
n 2
olda 4
```