ALGOL-0 Introduction to language theory and compiling Project – Part 1

Mounir Bounamar Mustapha Cherrabi

October 2019

1 Introduction

As part of this course, we were asked to implement a compiler for the ALGOL-0 language. This project is divided into 3 parts and the first part is to implement a lexical analyzer using the JFlex tool.

Jflex is a tool that takes as input a file (.jflex) that contains regular expressions and will generate a program that reads input (.java). Each regular expression will execute their corresponding java code if there is a match during the analysis. ALGOL-0 is a simple language that use the essential and basics constructors of a language like loops, conditions, assignation and mathematical operations.

2 Implementation

2.1 Regular expressions

We have at fig.2 all the regular expression that we had to implement. We decide to use macros as much as possible in order to keep a more readable flex document.

In addition to all the obvious regular expressions like the mathematical operation or the comparison operators, we were asked the add 2 more regex named **Varname** and **Number**. One will identifies the variables, which is a string of digits and letters, starting with a letter and the other one represent a numerical constant, and is made up of a string of digits only.

2.2 Macro

In this subsection, we will explain the use of some macros and their syntax. We defined a macro every time a regular expression was too long or too complex to write in one line. At fig.1 you can see all the macros that we used.

`AlphaLowerCase`	=	[a − z]
`Numeric`	=	[0 − 9]
`AlphaLowerNumeric`	=	{AlphaLowerCase}|{Numeric}
`Varname`	=	(AlphaLowerCase)(AlphaLowerNumeric)*
`Number`	=	([1 − 9][0 − 9]*)|0
`ShortComment`	=	co
`LongComment`	=	CO
`EndLine`	=	”\n”
`CarriageReturn`	=	”\r”
`ReturnLine`	=	(EndLineCarriageReturn?)|(CarriageReturnEndLine?)
`Space`	=	(\t|””)
`Ignore`	=	{Space}|{ReturnLine}

Figure 1: Extended regular expressions

Some expressions are very intuitive like for **AlphaLowerCase** where *[a-z]* means that we will take every letter starting from *a* to *z*.

We have a regex **Ignore** that will get match each time we have either a space, a carriage return or `\n` meaning the end of the line. For the moment, we ignore these information but they may be important in the rest of the project. Indeed, spaces in some places may be a mistake and we will need to handle these cases if need it. In addition, we do not distinguish between tabs and simple spaces.

2.3 Explanation of choices and hypotheses

We made the hypothesis that only lowercase letters were taken into account for **Varname**. It means that capital letters are not recognized by our lexical analyzer. We also made an assumption for **Number**. All number has to be integer.

It is important to have a lexical analyzer who can recognize the different token even those who are not supposed to be there. Indeed, unexpected tokens are words that do not match with any regular expression despite this we have to be able to handle them. Two possibility were available to us. Either we ignore them and we keep analysing the rest of the code or we throw an error at the right line to show what went wrong to the user. We decided to go with the last option because it does not make any sense to find an error and not report it. In addition, it facilitates the work to the user who will know directly what is the error made

2.4 Comments

Comments is one of the fundamental principles of a language. ALGOL-0 allows comments to be inserted into the code. We have to possibility to add either short comments or long comments.

Regular expression	Token
`\n`	EndLine
`\r`	CarriageReturn
`\t | " "`	Space
{`Varname`}	VARNAME
{`Number`}	NUMBER
`begin`	BEG
`end`	END
;	SEMICOLON
`:=`	ASSIGN
`=`	EQUAL
`/=`	DIFFERENT
`>`	GREATER
`>=`	GREATER_EQUAL
`<`	SMALLER
`<=`	SMALLER_EQUAL
`(`	LEFT_PARENTHESIS
`)`	RIGHT_PARENTHESIS
`while`	WHILE
`do`	DO
`endwhile`	ENDWHILE

Regular expression	Token
`if`	IF
`then`	THEN
`endif`	ENDIF
`else`	ELSE
`print`	PRINT
`read`	READ
`+`	PLUS
`-`	MINUS
`*`	TIMES
`/`	DIVIDE
`for`	FOR
`from`	FROM
`by`	BY
`to`	TO
`and`	AND
`or`	OR
`not`	NOT
`EOF`	EOF
{`Ignore`}	Ignore
{`ReturnLine`}	ReturnLine

Figure 2: Regular Expressions

In the case of short comments, it is limited to one line and must start with **co**. It is important to note that it must be written in lower case. When the lexical analyzer reads a short comment, it goes to **<shortcomment>** state and then ignores its entry until it reads the end of the comment marker. This end of comment marker is designated by the **ReturnLine** regex.

In the case of long comments, the situation is a little more complicated to manage since the comment can extend over several lines. You have to be able to detect the beginning and the ending of the comment. A long comment begins with a **CO** and ends with a **CO** both of them in capital case. Like for the short comment, we have a **<longcomment>** state. We will jump back to **<yyinitial>** from **<longcomment>** as soon as we have a closing **CO**. In case the comment is not closed, we have an exception that is raised and the error is reported to the user.

3 Bonus

The nested comments were not asked to be implemented (forbidden) because that would have made the scanner no longer a finite automaton. Indeed, the language of nested comment **is not regular** and we can demonstrate that by using the Myhill–Nerode theorem or the pumping lemma. There exist different method that allow us to show this. The nested comments problem cannot be recognized with a finite automaton, since a finite automaton has finite memory.

4 Javadoc and tests files desciption

We generated the javadoc automatically thanks to the javadoc library made available by java. It may seem not important for the this first part but it will be very useful as we go further in the project. It is important to note that we are using *jflex 1.7* and *java 11 (11.0.4)*

In order to launch the program, you can run the makefile on the terminal by using this command **make**. It is also possible to launch it via the command **java -jar Part1.jar testFile.alg**. The test file must be an **.alg** file otherwise the program will not be able to execute it and will signal it to the user.
We have a test folder to facilitate testing. In this folder we find different **.alg** files:

1. **euclide.alg and read.alg** these two files allow us the test the basics of our program

2. **testMinus.alg** this file will be used to test the case where we have a negative number. Our lexical analyzer will consider the minus and the number as two different and distinct token (*token minus and number*) because it does not yet take into account the priorities of the operations. Indeed, it should match it with the token **Number**. This situation will surely be managed in the next parts.

3. **testCharacter.alg** this file allows us to test the case where there is a character that is not recognized by our lexical analyzer. This situation may be encountered if the file contains a non-Latin alphabet or a word that does not match any regular expression. In our case, we put as variable ”c5D”. Our analyzer will recognize *c5* as a **Varname** but *D* will not match anything. Indeed, as previously said we made the hypothesis that **Varname** will only have lower case letter (with possibly numbers).

5 Conclusion

This first part of the project allowed us to be confronted for the first time with the way to implement a compiler. It helped us understand how a lexical analyzer work and most importantly how to implement it. The main concern was managing the comments but as soon as jflex's syntax and way of working was acquired, it was easy to deal with it. Through all of this, we have now the necessary tools to continue creating a compiler.