



UNIVERSITÉ LIBRE DE BRUXELLES

REPORT INFO-F403: PART 1

Abderrahmane Faher, Eden Aroch

School year 2024-2025

1 Regular expressions

In this section, we will present and explain the regular expressions used throughout our implementation.

1.1 Basic regular expressions

Here are the "basic" regular expressions that we defined:

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

Theses macros are self explanatory. They constitute a base for all the other regular expressions that we defined.

1.2 EOF and Space

We also implemented regular expressions to recognize a space or a end of line (EOF), because these are key elements in order to tokenize input properly:

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

A space in a text that the lexer has to tokenize can be represented by a "tabulation" ("\t") or a blank space (" "). This is why the regular expression is the union of both. The EOF varies between different OS's. Some use a combination of "\r" and "\n", some others simply use "\n", that's why we made the "\r" optional by using the "?" character.

1.3 Program name, variable name and number

We implemented regular expressions to match a program name, a variable name and a number (these were the only lexical units that required a more complex regex to be matched, the other lexical units like LET could just be matched by specifying a simple regex "LET"). Here are the regular expressions:

- ProgName = {AlphaUpperCase}({Alpha} | "._")*
- VarName = {AlphaLowerCase}{AlphaNumeric}*
- Number = (0 | [1-9]{Numeric}*)

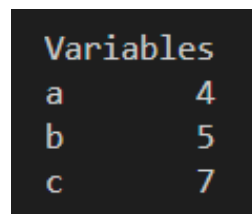
The regular expressions above have been constructed in order to respect all the constraints that they need to respect. Per instance, the 'ProgName' regular expression makes sure to only recognize a string of characters that starts with an uppercase letter and is followed by whichever letter or the character ".": The 'VarName' regex only matches with strings that starts with a lower case letter, like mentioned in the project instructions. The 'Number' regex matches with '0' or with numbers that don't begin by 0 (like 129438943). The reason of this implementation is that we do not want to accept numbers like '0329857', because it starts with a '0'.

2 Project's structure

In this section, we will discuss and explain the structure of our project.

At the root of the project, there is a Makefile that can compile the 'LexicalAnalyzer.flex' file, create the '.jar' and run tests on all the example files that are in the '/test' folder.

The core logic of the project is located in the '/src' folder. In this folder, there is the 'LexicalAnalyzer.flex' that contains all the instructions that the lexer has to follow. There is also all the '.java' files that were provided on the 'UV' and a 'Main.java' file that calls the lexer (obtained by compiling 'LexicalAnalyzer.flex') and parses all its output to print it out clean. The 'Main.java' file, while reading the lexer's output, computes a variable table that is outputted at the end of the program (see example of a variable table 1).



Variables	
a	4
b	5
c	7

Figure 1: Example of a variable table

3 Justification and choices

In the lexer, we decided to use exclusive states so that comments can be handled in a totally separate way by the lexer. In the 'COMMENT' state, we simply ignore everything that the lexer reads, and when the lexer reads a string indicating the end of the comment, it returns to the 'YYINITIAL' (initial state) state, that analyzes the lexer's input, instead of ignoring it, like in the 'COMMENT' state.

The Main.java file reads the output from the lexer, parses it, and prints everything in a clear way (ignoring white spaces and other useless information). This is done because the lexer's output isn't perfectly formatted, so we decided that the parsing had to be done in the 'Main.java' file.

4 Descriptions of example files

The '/test' folder contains all the example files that we can test the lexer on. There are several files that test if the lexer successfully ignores short and long comments. There are also files that test if the 'Main.java' parses the lexer's output successfully, by ignoring white spaces. Additionally, the '/test/' folder contains files that test the general functioning of the lexer (if it recognizes variable names, program names, ...).

5 Difficulties when handling nested comments

The main difficulty when handling nested comments in our case, is that we are already in the '**COMMENT** state' when the nested comment is encountered. That means that when a nested comment is found, it will be ignored, because we are already in the '**COMMENT** state'. The problem with this is that when the nested comment will be closes (using "!!"), the lexer will see that as the end of the "parent" comment (comment that contains the nested comment) and will enter the '**YYINITIAL** state' (the default state) and will treat the remaining of the comment as actual code.

Below is an image of the 'COMMENT' state for better understanding.

```
<LONG_COMMENTS_STATE> {  
    "!!" {yybegin(YYINITIAL);}   
    . {} // Ignore every character in the comment  
}
```