Université Libre de Bruxelles

INFO-F-403
Introduction to language theory & compiling

Report

Étudiants :
Abderrahmane Faher, 000514457
M1-INFO
Eden Aroch, 000542373
M1-INFO

Enseignants :
G. GEERAERTS
M. BALACHANDER
M. SASSOLAS

24 octobre 2024

Table des matières

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 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 it's 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 below).

```
Variables
a 4
b 5
c 7
```

3 Choices and justifications

3.1 xstates vs states

We opted to use exclusive states (xstate) rather than normal states for handling the various states of the lexer. It allows the lexer to process specific tokens only when it's in a particular state. For instance, in the case of SHORT_COMMENTS_STATE and LONG_COMMENTS_STATE, the lexer needs to ignore characters or handle specific tokens like newlines differently than in the general lexical analysis. By using exclusive states, we ensure that the lexer only processes comment-related tokens while it is in a comment state, and then returns to the initial state (YYINITIAL) once the comment has ended.
This approach prevents the lexer from mistakenly interpreting comment content as part of the main syntax of the language and providing more structure and a easier readability to the lexer.

3.2 ProgName state

As it's indicated in the GILLES grammar table, we have to respect this format :

$$\langle \text{Program} \rangle \rightarrow \text{LET } \textbf{[ProgName]} \text{ BE } <\textbf{Code}> \text{ END}$$

The PROGNAME_STATE role is to interpret the word after the keyword LET as a Program name as specified even if it is another keyword of the project's lexicalUnit. The example below shows the test file illustrating this situation.

```
LET THEN BE
```

```
<PROGNAME_STATE> {
    {Space} {}
    {EndOfLine} {yybegin(YYINITIAL);}
    {ProgName} {Symbol token = new Symbol(LexicalUnit.PROGNAME, yyline,
   yycolumn, yytext()); yybegin(YYINITIAL); return token; }
}
```

4 Description of Test 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, some files that test if the 'Main.java' parses the lexer's output successfully, having the same content but separated by white spaces and end lines differently and another file testing the program's name analogy as described in the section above.

5 Makefile

This project's Makefile is designed to automate the process of *compiling, packaging, running, and testing* a Java-based lexical analyzer project. It has a *default target*, **all**, which depends on three other targets : compile, jar, and run. The **compile** target first uses jflex to generate a lexer from the LexicalAnalyzer.flex file, and then it compiles the Java files in the src directory using javac. The **jar** target ensures that the project is compiled before creating a .jar file, which includes the compiled classes and a manifest file. The **run** target runs the program using the generated .jar file. It iterates over the input files, executing the LexicalAnalyzer class for each one via the java -jar command. The **test** target automates the testing process by running the program for each file listed in TEST_FILES, ensuring that the program is thoroughly tested on multiple inputs. Lastly, the clean target removes all compiled .class files and the generated .jar file, allowing the reset of the build environment.

6 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 except the newline
   character
```

```
4       {EndOfLine} {} // Ignore the newline character
5   }
```