Introduction to language theory and compiling

INFO-F-403

Project : Part 1

27 October 2020

BAKKALI Yahya : 000445166
HAUWAERT Maxime : 000461714

Université Libre de Bruxelles (ULB)

Contents

1 Introduction

In this project it has been decided to design and write a compiler for a new programming language named "Fortr-S". Fortr-S is a simple imperative language created to replace and named after the old programming language "Fortran". Fortran's first appearance dates back to 1957 and is now inadequate for today's programming needs. Fortr-S has the ability to determine if a word is whether a keyword, the name of the program or the name of a variable just by looking at the letters that compose the word. In this first part of the project, the lexical analyzer will be implemented.

2 Implementation

2.1 Tools

JFlex is used to generate a lexical analyzer, based on deterministic finite automata (DFAs), with defined rules. This lexical analyzer gives an easy way to extract every token of code. Each time a token is detected JFlex execute a Java code. The rules are expressed with regular expressions.
Java is used to code and execute the compiler with all classes needed for the compiler to work.

2.2 Regular Expressions

2.2.1 Definition

In order to improve the readability aliases have been used.

Regular expression	Description	Alias
$[A-Z]$	Accept any uppercase letter	AlphaUpperCase
$[a-z]$	Accept any lowercase letter	AlphaLowerCase
$[0-9]$	Accept any digit	Numeric
$[A-Za-z0-9]$	Accept any digit or letter	AlphaNumeric

2.2.2 Program names

The names of the programs should contains only letters and digits starting with an uppercase letter and should not contain only uppercase letters.

$$\{AlphaUpperCase\}+[a-z0-9]\{AlphaNumeric\}*$$

2.2.3 Variable names

The names of the variables should contains only lowercase letters and digits starting with a lowercase letter.

$$\{AlphaLowerCase\}[a-z0-9]*$$

2.2.4 Numbers

The numbers should not have leading zeroes.

$$\{[1-9]\{Numeric\}*\}|0$$

2.2.5 Line terminators

There are three types of line terminator for Unix, Windows and Mac OS

$$\backslash n|\backslash n\backslash r|\backslash r$$

2.2.6 Comments

Short comments The short comments start with // and end automatically before the end of the line.

$$"//".*$$

Long comments To handle long comment, two rules and an exclusive state "COMMENT_STATE" were added. When a new long comment begins the lexer switch to the state COMMENT_STATE. In the COMMENT_STATE if a new long comment is detected an exception is thrown and if the end of a long comment is detected the lexer switch back to the initial state.

$$OpenLongComment : "/*"$$

$$CloseLongComment : "*/"$$

2.2.7 Perfect match

And there is also some tokens that need to be match as they are defined in the FORTR-S syntax:

- Keywords : BEGINPROG, ENDPROG, IF, THEN, ENDIF, ELSE, WHILE, DO, ENDWHILE, PRINT, READ
- Arithmetical operators : -, +, *, /
- Utility operator : := (the assign operator), =, >
- Parenthesis : (,)

2.2.8 Spacing

To ignore spacing characters, the space and the tab character the following rule was added with a void code to be executed.

$$"\ "|\backslash t$$

2.2.9 Other

To catch lexical errors the following rule was added and the code associated with it throws a *LexicalException*. Because everything that falls in this category doesn't exist in the Fortr-S lexicon.

$$[\hat{}]$$

2.3 Explanation and hypothesis

2.3.1 Explanation

- Each time a token is detected the analyzer returns a Symbol object with a LexicalUnit type and with value of the token, the only exception is for the comments as their content is discarded.
- The lexical analyzer takes as an argument the file containing the code to be analyzed. The program prints every token of the code then print the name of all the variables followed by the line of its first occurrence.
- The symbol table contains all recognised variables, in lexicographical (alphabetical) order. To get such a table a TreeMap has been used. Each time a VarName is encountered it is put in the map if it is the first time the variable has been encountered. The treeMap keeps the variables in lexicographical order at all times.

2.3.2 Hypothesis

Syntax errors were decided not to be handled as the next part, the syntax analyzer, is more appropriate.

For example numbers: if 042 is written in the code, the lexer will split it into two parts 0 and 42. The future syntax analyzer will detect two numbers following each other and then will throw an error.

The only exception is the long comment because comments are not part of the set of tokens returned so it's the only way to catch these errors.

2.3.3 Errors

Lexical errors are detected with the latest added rule and *LexicalException* is thrown.

Two syntax errors were handled:

- When a long comment is closed without it being opened first. It is detected when the lexer is in the initial state and a *CloseLongComment* is detected.
- When the end of the file is reached and a long comment wasn't closed. It is detected in the `%eofval` statement if the lexer was in the `COMMENT_STATE`.

In both cases the program throws a *SyntaxException*.

2.4 Classes

The main code verify that the path of the file containing the source code to compile has been passed. Then it creates an instance of the *Compiler* class then runs the *compile* method and pass the path.

The class *Compiler* has two important methods:

- The method *lexicalAnalyse* return the list of tokens of the code of the source file.
- The other method *compile* is the main one, it calls *lexicalAnalyse* and for the moment it just prints the tokens and the symbol table.

The two types of exception are placed in the *exceptions* package.

3 Nested comments (bonus)

The problem of nested comments is similar to the problem of the well-parenthesised words with the open parenthesis replaced by "/*" and the closed one by "*/". The language that accept "well-commented" $L_{/**/}$ words is not regular. The use of a counter is needed to count the number of current open comments.

3.1 Example

$$L = \{/*, */\}^* \neq L_{/**/}$$

$$\left.\begin{array}{r} /**/*/ \in L \\ *//**/ \in L \\ /*/**/ \in L \end{array}\right\} \notin L_{/**/}$$

Note: the language of Fortr-S is more complicated but in this problem it is useless to describe to whole language as the things in and out of the comments do not matter.

3.2 Implementation

A new counter variable was added and several things had to be changed. When a new long comment is detected the counter is set to 1 before switching to the `COMMENT_STATE`.

When the lexer is in the `COMMENT_STATE` :

- When a new long comment is detected, instead of throwing an exception, the counter is increased.

- When the end of a long comment is detected the counter is decreased and when the counter is equal to 0 the lexer switch back to the initial state.

4 Description of example files

Three example files have been added:

- Maximum: a simple code that ask the user two numbers and prints the maximum. It doesn't contain any error.
- Minimum: a simple code that ask the user two numbers and prints the minimum. It doesn't contain any error with well-nested comments.
- NestedComment: a code with comments but without instructions. It contains an error because there is an extra "*/" at the 18th line.

For each source code example there is a ".output" with the expected result from the lexical analyzer.

5 Conclusion

This first part of the project is essential for the compiler to be working properly. The lexical analyzer must be working flawlessly, otherwise the syntax analyzer will not work as expected.