Université Libre de Bruxelles
École polytechnique de Bruxelles
Master of science in Computer Science and Engineering

ECOLE
POLYTECHNIQUE
DE BRUXELLES

INFO-F403 - Introduction to language theory and compiling

Report of the Compiling Project - Part 1

Authors:
Nikita FAESCH
Maxime THONAR

Professor:
Gilles Geeraerts

2021-2022

Contents

1 Regular expressions

First of all, let us present the short list of regular expressions we used for this part of the project:

VarName = $[A-Z]|[a-z]\ ([0-9]|[A-Z]|[a-z])^*$
Number = $[1-9][0-9]*|0$
ShortComments = ”co” .*
LongComments = ”CO” .* ”CO”

In fact, a [VarName] identifies a variable, which is a string of digits and letters, starting with a letter (this is case sensitive). A [Number] represents a numerical constant, and is made up of a string of digits only. ShortComments start with the string ”co”, followed by arbitrary characters, while LongComments start with the string ”CO”, followed by arbitrary characters and end with the same string ”CO”.

2 Presentation of the work

For this part of the project, only 2 files were coded: Main.java and LexicalAnalyzer.flex.

The LexicalAnalyzer.flex file is separated into 3 parts: the first consists of options for the scanner to be prepended in the output Java scanner program, the second contains all the regular expressions and the third consists of defining the action to be done when a token is encountered in the input file. The tokens consisting of special characters such as ”\n”, ”\t” or a simple space are simply being ignored. Other special characters such as ”\r” or ”\r\n” exist but are not relevant for this project, as our files were coded under a Linux system.

The Main.java file contains two methods. The method main(String[] args) calls the lexical analyzer with args[0] (which is supposed to be the input file) as an argument. Each token encountered will then be printed on the standard output, using the method toString() provided in the file Symbol.java. If the token is a [VarName], the method varL will be called (this method is described hereafter). If the token is the end of the file, the word ”Variables” is printed on the standard output, followed by each [VarName] encountered in the input file as well as the line where they were first encountered. The method varL(String var, int line) allows to create a list in which each element (the lines) is associated to a key (the variable name). The variable names are stored in the lexicographical order.

3 Example files

Five example files have been added, in addition to the provided euclid.co file. ”test.co” is the most basic alCOl file possible: it is only composed of begin followed by end. This file was used to debug our code. ”special_cases.co” contains all the special cases we thought of:

- The sequence 00000 is recognised as 5 different [Number], as we made the choice that a number composed of multiple digits should not start with a zero.
- The sequence CO CO CO is recognised as a long comment, as a comment should start with CO and end with CO, whatever the characters in between may be.
- The sequence abraca1234dabra is recognised as a [VarName], as it should according to the definition of our regular expression.
- The sequence co co is recognised as a short comment because we have defined that all strings written after a co is considered as a comment.
- The sequence 1abc is recognised as a [Number] (1) followed by a [VarName] (abc), as a [VarName] must start with a letter.
- The sequence 10E4 is recognised as a [Number] (10) followed by a [VarName] (E) followed by a [Number] (4), as a number must consist of a string of digits only.
- The sequence is recognised as two unidentified tokens, as they are not included in the LexicalUnit.java and LexicalAnalyzer.flex files.

”test_tab.co” and ”test_newline.co” both have a very similar interest: they were used to test the handling of, respectively, ”\t” and ”\n” characters. ”not_in_euclid.co” was used to test the recognition of all the tokens that are not present in the ”euclid.co” file.
All these tests were successful.

4 Nested comments

Nested comments can't be handled by our lexical analyzer because as a comment is written between two "CO", if we write CO CO *comment* CO CO, the compiler will interpret the space between the two COs as the comment and will see *comment* as a [VarName]. A possible solution to handle those nested comments is to change the definition of how a comment can be written. Indeed we could say that for a string between two COs to be considered as a comment, it must contain at least one char. We could therefore treat nested comments by saying that when between two COs there aren't any char (there's just a space or nothing), we don't consider what's between them but what will be written after those two COs.