ECOLE
POLYTECHNIQUE
DE BRUXELLES

INFO-F403 - Introduction to Language Theory and Compiling

Compiling Project Part 1

DE VOS Sébastien

KALAI Tarik

GEERAERTS Gilles

WINTER Sarah

BERTHON Raphael

M1-IRIFS - 2021-2022

Table des matières

1 Introduction

As part of the course INFO-F403, a compiler needs to be computed. This compiler is specifically made for the alCOl language. This report explains and justifies the choices made during the implementation of the program.

There will be two main sections in this report : the first one is about how the program works and the description of the different implementations realised. The second part is about the bonus, a proposed answered is detailed in that section.

2 Work process of the program

Since the analyser will check every character and compare it in the order we indicated in the code [1] (by putting first in line what we want to detect first) we will present them in the same order.

2.1 Identification of tokens

We will first check if it falls in the category of relational operators which is composed of :

- Relational operators (e.g. : "+", "/", " :=", ...)
- if-for-while keywords
- others : ("(", ")", ";")

Here the order between all of the 3 different types of operators does not matter because they are precise tokens [2], this is also why the comparison begins with them.

2.2 States

In the case of CO and co, we had to implement states [3] so that we do not take into account what is between (CO *text* CO) or after them (co *text*). To do so, three simple states have been created :

- *YYINITIAL*
- CO_STATE
- co_STATE

1. We discovered this by trial and error
2. Such that they will not be included in the scope of a less restrictive language such as the Identifer language in the Regular Language section
3. The use of states is justified here because we are going to do the same thing each time we detect Co or co

First we begin in the YYINITIAL State, if the token "CO" or "co" is detected then we go into the respective states where we do nothing or wait for another "CO" in the case of the CO_STATE or an EndOfLine character for the co_STATE.

When these characters are detected we don't take into account all the previous ones thanks to the "^" command that allows to not accept anything from the beginning of the state to the character next to "^". After that we return into the YYINITIAL State.

```
<CO_STATE> {
    [^"CO"] {}
    "CO" {yybegin(YYINITIAL);}
}
```

FIGURE 2.1: CO_STATE

```
<co_STATE> {
    [^"\r"?"\n"] {}
    {EndOfLine} {yybegin(YYINITIAL);}
}
```

FIGURE 2.2: co_STATE

2.3 Extended Regular Expression

Finaly, in our code we have expressed different regular expressions (these being the least restrictive, they come last in the program) :

- AlphaUpperCase : all the uppercase letters.
- AlphaLowerCase : all the lowercase letters.
- Alpha : all the upper and lower case letters.
- Numeric : all numbers from 0 to 9.
- AlphaNumeric : the combination of Alpha and Numeric.
- EndOfLine : characters that end a line.
- Integer : all integer numbers.

- Identifier : all combinations of integers and letters starting with a letter.

These are used to detect all numbers and variables in the code we need to compile. Integer will be tagged as NUMBER and Identifier as VARNAME.

2.4 Java Code

Each of these tokens and regular expressions, will call either the *addToSymbol* method and only the *Indentifier* tokens will call the *addToVariableAndSymbol* method. When we call these methods we create a Symbol object with the given *Symbol.java* Class which receives 4 arguments [4] : The LexicalUnit associated, the column, the line and the value.

At the end of the program we have an end of program command that just returns the end of stream to finish properly.

2.5 Test

We created a test [5] specifically made to verify if our program could recognise all the different symbols [6]. As it was only to test the capabilities of our program the test itself does not do anything, it is just a series of commands. However it implements all the possible situations we could have thought off.

4. Actually we don't need the 4 arguments. In the case of the VARNAME we need only 3 : the lexical unit, the line and the value(yytext()). For all the others we actually only need 2 : the LexicalUnit, and the value(yytext()). But Since we didn't know whether they would be useful or not in the future parts of the project we just safely took all of them.
5. It is named : *test_1.co*
6. The output is kept under the file :*text_1.out*

3 Answer to the Bonus

3.1 Description of the problem

The compiler cannot handle nested comments. What may be an idea of implementation that will allow the compiler to do such thing? (Example : CO CO Hello world CO CO)

3.2 Resolution

Our idea to detect and handle nested comments is to implement a counter. We would add 1 to the counter in the case where the beginning of a comment is detected and we would remove 1 in the case where it is the end of a comment that is detected.

However, in this project we cannot make a distinction between the opening and ending of a comment. As such we will try to answer to the question by simplifying it.

We will consider a particular type of nested comments : *CO CO Hello world! CO CO*. This nested comment has his two openings and endings separated by a simple space [1] whereas the text itself is put between the nested CO. In this case we can handle nested characters using states if we consider that we have a regular language (which is another simplification of the question because we are not in the case of a regular language as we can have an infinite amount of nested comments) here is an illustration :

1. We also take into account the end of line

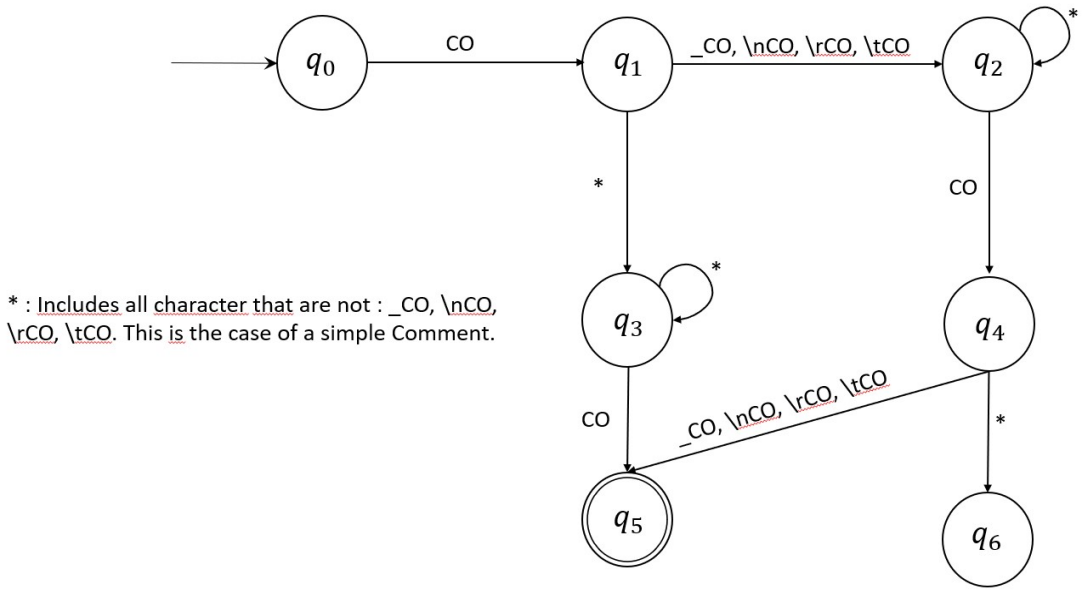

FIGURE 3.1: Automaton

However, if we have to take into consideration the whole possibilities of nested comments (e.g : *CO Hello CO World CO! CO* We think it is impossible to handle because we simply cannot make a distinction between the end and the beginning of a comment.