ALCO1 COMPILER

Glibert Aloïs - 393192 — QUIVRON Loic - 526415

2021-2022

Contents

1 Introduction

The purpose of this whole project is to implement a compiler for this alco1 language. But for this first part, we only had to implement the scanner.

2 Regular Expressions

To implement this scanner we had to create four main regular expressions, one each for:

- the variables
- the numbers
- the one-line comments
- the multi-line comments

We'll describe them here a little further.

2.1 Variables

A variable begins with a letter and is then followed by any number of alpha-numeric character.

```
1 Digit = [0-9]
2
3 AlphaLower = [a-z]
4 AlphaUpper = [A-Z]
5 Alpha = {AlphaLower} | {AlphaUpper}
6 AlphaNum = {Alpha} | {Digit}
7
8 VarName = {Alpha} {AlphaNum}*
```

We defined the intermediary RegEx's as follow :

- Digit is self explanatory
- AlphaLower is a lower alphabetical character

- AlphaUpper is an upper alphabetical character
- Alpha is either an AlphaLower or an AlphaUpper character
- AlphaNum is either an Alpha or a Digit

Remark that we could have used Alpha = [a-Z] but by using two intermediary RegEx's, we can use either one of them if needed (but this is not necessary for this language).

We finally defined a variable name (VarName) as a token beginning with an Alpha, and then, a possible infinite succession of AlphaNum's.

2.2 Numbers

A number contains only digits. The regular expression built for the digits is the same as described above. We then obtain the regular expression for a number which is :

```
Number  = {Digit}+
```

As we need at least one digit for a token to be a number.

2.3 Comments

2.3.1 One-line

A one-line comment is defined as a token which begins by the letters 'co' and ends with the line. The regular expression associated with it is :

```
SmallCom  =  "co" .*
```

2.3.2 Multi-line

A multi line comment is defined as a token beginning with 'CO' and goes up to (using the "∼ " RegEx symbol) the first next 'CO'. The RegEx associated with the multi-lines comment is :

```
BigCom  =  "CO"~"CO"
```

3 Choices and Hypothesis

We chose to halt the program as soon as we identify an unexpected token and in the error message, only to prompt the first character which was responsible for the unexpected token error.
As the scanner scope is local, when we encounter such strings as "98Alexia", our parser takes it as two tokens, 98 is a NUMBER and Alexia is a VARNAME.
This also explains why we only prompt one character for unexpected token errors.

4 Description of example files

We use the term "keyword" to talk about a reserved word of the language such as "begin", "read", "end", etc.

4.1 File test1.co

```
1    beginb
2    begin
3    source3 co can run
4    soUr33ce
5    Something
6    4blah = blih34
7    beg(in
8    432
9    -34
```

This example file is used for many things, for each line :

1. extending a keyword makes it unrecognized as a keyword, and becomes a variable
2. a correct keyword is correctly recognized
3. a variable can end with a digit and a one-line commentary is ignored
4. a variable can contain a digits and upper-case letters
5. a variable can start with upper-case letters

6. when we find a token beginning by a digit and followed by letters, it is recognized as two different tokens

7. dividing a keyword by another token makes it unrecognizable

8. a valid number is recognized

9. a negative number is recognized as a combination of a MINUS and NUMBER lexical units

We can check what we expected here on this output of our program :

```
[...]
List of Lexical Unit
token: beginb        lexical unit: VARNAME
token: begin        lexical unit: BEG
token: source3        lexical unit: VARNAME
token: soUr33ce        lexical unit: VARNAME
token: Something        lexical unit: VARNAME
token: 4        lexical unit: NUMBER
token: blah        lexical unit: VARNAME
token: =        lexical unit: EQUAL
token: blih34        lexical unit: VARNAME
token: beg        lexical unit: VARNAME
token: (        lexical unit: LPAREN
token: in        lexical unit: VARNAME
token: 432        lexical unit: NUMBER
token: -        lexical unit: MINUS
token: 34        lexical unit: NUMBER
token: null        lexical unit: END_OF_STREAM
```

4.2 File crash.co

```
beginb
begin
source3 co can run
soUr33ce
Something
```

```
eazr%bib
4blah = blih34
beg(in
432
-34
```

Here the file is meant to make our parser crash. Indeed, the character "%" is not part of the language. As we said before, we then get an error as shown below :

```
UnexpectedTokenException: Found an unexpected token : % on line : 6
        at LexicalAnalyzer.nextToken(LexicalAnalyzer.java:735)
        at Main.readSourceCode(Main.java:43)
        at Main.main(Main.java:15)

Process finished with exit code 0
```

5 Thoughts on nesting comments

Since comments are most often dealt with in the scanner and the scanner is usually based on regular expressions, we could not detect nested comments. Indeed, we could reduce this problem to the Dyck's language (the language of well parenthesised words), which needs at least a counter to be able to work. As such, it is not part of the regular languages and therefore cannot be expressed as a regular expression.

In the more special case of the commentaries of alco1, since the starting and ending keywords for the multi-line commentaries are the same, we couldn't even detect when there is a start and a beginning and we would need to be able to associate them pairwise after reading the whole text, which again is not the role of the scanner.