Project Part 1 Compiling

Antoine Bedaton and Pierre Defraene

October 2020

1 Introduction

For this project, we were asked to create a compiler for FORTR-S, a simple imperative language. Since creating a compiler is a difficult task, we were asked to divide this project into three main parts. For the first part of this project, we need to produce the symbol table by creating the lexical analyser of the compiler using `JFlex`, a lexical analyser generator for `Java`. The second part will be syntax analysis and we will conclude this project by doing a Semantic analyser.

2 REGEX

The lexical analyser will parse the input file and will try to match the according regexes. Therefore we need to define multiple regexes in order to match the signs, numbers, and all keywords.

2.1 Basic regexes

First of all, We define simple regular expressions:

- Every letter in upper case: `AlphaUpperCase` = `[A-Z]`
- Every letter in lower case: `AlphaLowerCase` = `[a-z]`
- Every digit: `Num` = `[0-9]`

Then we can combine those expressions to create some new regular expressions:

- Every letter or digit: `AlphaNum` = `[a-z]` | `[A-Z]` | `[0-9]` or if we use the name decided above : {`AlphaUpperCase`} | {`AlphaLowerCase`} | {`Num`}
- Every lower letter or digits: AlphaLowerNum = `[a-z]` | `[0-9]` or if we use the name decided above : {`AlphaLowerCase`} | {`Num`}

2.2 Variable

A variable is a string of digits and lowercase letters, starting with a letter. So the regular expression for these is a concatenation of a letter in lowercase followed by the Kleene closure of a letter in lowercase or digit:

`[a-z][a-z0-9]`* or {`AlphaLowerCase`}{`AlphaLowerNum`}*

2.3 Numbers

For numbers, we defined our rules according to `Python3`. While `12.` is a valid number, `012` is not.

Therefore we will accept all integers that don't start with a `0`. For the floats, we accept all integers followed by a "." and possibly some digits. We define an integer as the Kleene closure of any digits. The decimal value is defined as a point followed by a Kleene closure of the digits. By concatenating both regexes, we obtain all real numbers. So the regular expressions are:

- `Integer` = `(([1-9][0-9]`*`)` | `0)` or `(([1-9]`{`Num`}*`)` | `0`
- `Decimal` = \\`.[0-9]`*
- `Real` = {`Integer`}{`Decimal`}`?`

We also used a special regex to match the invalid number such as `012`:

- `BadNumber` = `0`{`Real`}

Other bad numbers will be catch by the last [$\wedge$] and will throw an error. For example:

- `12.4.2` will match `12.4` and will throw an error when reading the "."
- `12..3` will match `12.` and will throw an error when reading the second "."

2.4 Program Name

A Program name is a string of digits and letters, starting with an uppercase letter but not entirely uppercase. So the regular expression is a concatenation of a letter in upper case followed by the Kleene closure of the regular expression `AlphaNum`, `AlphaLowerNum` and again a Kleene closure of `AlphaNum`.

`[A-Z][a-zA-Z0-9]`*`[a-z0-9][a-zA-Z0-9]`* or
{`AlphaUpperCase`}{`AlphaNum`}*{`AlphaLowerCase`}{`AlphaNum`}*

This regex just says that we need to start with an uppercase letter, and have at least one lowercase or digit in the word.

2.5 Keyword

A keyword is a word totally in Uppercase. The regular expression is quite simple for them:

`[A-Z]+` or `{AlphaUpperCase}+`

3 Java code

3.1 ensureGoodUnit method

This method is used to check if the unit is in the `LexicalUnit` enum provided. If it is not, we raise an error.

4 State

We will now describe the states we used for this part of the project.

4.1 YYINITIAL

The `YYINITIAL` state is the main state, we use it to match all keywords and signs. For all valid sign/keywords, we call the ensureGoodUnit method. The only four exceptions are:

1. `Comment`: When we read a "`//`", we enter the `COMMENT_STATE` state.
2. `CommentBlock`: When we reach a "`/*`", we set the variable openComments to true and enter the `MULTICOMMENT_STATE`.
3. `EndOfBlock`: When we reach a "`*/`", we automatically throw an `IllegalArgumentException` because we know it is not possible to find a end of block comment here.
4. `BadNumber`: Again, if we find a bad number, we throw an error.

4.2 COMMENT_STATE

In this state, we loop until we find a line terminator, and we then jump to `YYINITIAL` to continue.

4.3 MULTICOMMENT_STATE

When we are in this section, it means that we found a "`/*`". If we find a line terminator, we do not create an endline symbol, we assume that one entire block of comment equals one line. Then, if we find a another "`/*`", we throw an error because we cannot have nested block comments. If we find the end of block "`*/`", we go back to `YYINITIAL`.

5 Comments

There are 2 types of comments in this language:

- Short comments which are all the symbols appearing after `//` and up to the end of the line
- Block comments which are all the symbols occurring between `/*` and `*/` keywords

5.1 Nesting block comments

A nesting block comment is a block comment inside a block comment (ex: `/* hello /* inside */ outside */` In this language, nested comments are forbidden and should throw an error. Although, we could have handled them with a counter, starting from `0`, we increment the counter every time we find a new comment and decrements every time we find an end comment. If the counter become 0 we are not in the comment anymore and if it become negative, there is more closing comment than open comment and we throw an error.
Another option is to use a pushdown automata and use a stack as a counter when we see a /∗ we push it on the stack then when we see a ∗/ we remove ∗/ from the stack.

6 Example Files

We created a bunch of test files in order to test if our parser works properly. Each one has a unique goal.

6.1 read.fs

It is the most basic test file, he reads a variable and store it inside the `b` and prints it.

6.2 Factorial.fs

This program finds the factorial number of a number. This is the test file given by the statement.

6.3 pgcd.fs

This program finds the pgcd of two number.

6.4 NestedComments.fs

This file is meant to test the nested comments. It should return an error right at the start.

6.5 BadNumber.fs

In the BadNumber file, it should return an error due to a bad assignation. Indeed, we try to assign `012` to `number`.

7 Conclusion

We successfully managed parse an input file and store the variables into a table. The next part of the project will consist in doing the sementic analysis. We will need to create an Abstract Syntax Tree.

ME
Why can't you just be normal?
regex
^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$