ECOLE
POLYTECHNIQUE
DE BRUXELLES

INFO-F403 | Introduction to language theory and compiling

Project Report
Part 1

Darius Couchard
Stefano Donne

27th October 2020

1 Introduction

The project consists in the design and writing of a compiler for FORTR-S. This report aims on the work done on the first part of this project : the scanner, and thus will describes how the lexical analyzer was implemented.

2 Regular Expressions

Different regular expressions were implemented, in respect of the JFlex syntax, to allow pattern matching of the lexical units by the lexical analyzer.

-Variables Name = $[a-z][a-z0-9]*$, starts with a lowercase character and can be followed by any combination of lowercase character or digits.
-Program Name = $[A-Z][A-Za-z0-9]*$, starts with a uppercase character and can be followed by any combination of lowercase/uppercase character or digits.

-Wrong Numbers = $[0][0-9]+*$, matches any combination of digits starting with 0.
-Numbers = $[1-9][0-9]*||[0]$, matches any combination of digits starting by something else than 0, or 0 itself.
The idea for matching numbers is straightforward. Indeed, the Wrong Number RegEx has a higher priority in the lexical analyzer than the Number RegEx. By doing that we insure that an error is thrown immediately after passing by a wrong number pattern and that no partial matching of a wrong number is done.

-LineTerminator = $\backslash r|\backslash n|\backslash r\backslash n$, corresponds to carriage return, newline or the two. On Window *Enter* uses carriage return and newline, while on Unix only newline.
-InputCharacter = $[\wedge\backslash r\backslash n]$, corresponds to any pattern which doesn't include carriage return or newline.

3 Comments handling

Two types of comment exists :

- **Inline comment**, starting with //, until the end of line.

- **Block comment** , starting with /* and ending with */.

The inline comments are handled by simply ignoring // and any **InputCharacter** following.

For the block comments it is a bit more tricky because we have to throw an error in case of comment nesting.

This is why another meta state called BLOCK_COMMENT was defined for managing this situation. This state is set with JFlex's yybegin() method when the **BlockCommentStart** = $/*$ regular expression is matched in YYINITIAL state.

In this state only two behaviors are possible :

- **BockCommentStart** RegEx is matched again, thus throwing an error because nested comments aren't supported.

- **BlockCommentEnd** = $*/$ RegEx is matched, making the lexical analyze return to YYINITIAL state.

Nested comments are more difficult to handle because it's needed to keep track of the several start and end patterns that already matched. A solution for supporting them would be to implement a counter in the BLOCK_COMMENT state which is incremented each time **BlockCommentStart** is matched and decremented everytime **BlockCommentEnd** is found. When the counter is at 0 it returns at YYINITIAL state.

4 Example

Here's a code example and the output given by the lexical analyzer.

```
1  BEGINPROG Test
2
3  /* Test */
4
5  //this is a test
6  a := 0
7  b := 1
8  READ(input)
9  WHILE b > a DO //TEST
10     c := b
11     b := b + input
12     IF (b > c) THEN
13         /* test
14
15         */
16         PRINT(b)
17     ELSE
18         //test
19         input := input + 1
20     ENDIF
21 ENDWHILE
22 ENDPROG
```

Test.fs

```
token: BEGINPROG        lexical unit: BEGINPROG
token: Test             lexical unit: PROGNAME
token: \n               lexical unit: ENDLINE
token: \n               lexical unit: ENDLINE
token: \n               lexical unit: ENDLINE
token: \n               lexical unit: ENDLINE
token: \n               lexical unit: ENDLINE
token: a                lexical unit: VARNAME
token: :=               lexical unit: ASSIGN
token: 0                lexical unit: NUMBER
token: \n               lexical unit: ENDLINE
token: b                lexical unit: VARNAME
token: :=               lexical unit: ASSIGN
token: 1                lexical unit: NUMBER
token: \n               lexical unit: ENDLINE
token: READ             lexical unit: READ
token: (                lexical unit: LPAREN
token: input            lexical unit: VARNAME
token: )                lexical unit: RPAREN
token: \n               lexical unit: ENDLINE
token: WHILE            lexical unit: WHILE
token: b                lexical unit: VARNAME
token: >                lexical unit: GT
token: a                lexical unit: VARNAME
token: DO               lexical unit: DO
token: \n               lexical unit: ENDLINE
token: c                lexical unit: VARNAME
token: :=               lexical unit: ASSIGN
token: b                lexical unit: VARNAME
token: \n               lexical unit: ENDLINE
token: b                lexical unit: VARNAME
token: :=               lexical unit: ASSIGN
token: b                lexical unit: VARNAME
token: +                lexical unit: PLUS
token: input            lexical unit: VARNAME
token: \n               lexical unit: ENDLINE
token: IF               lexical unit: IF
token: (                lexical unit: LPAREN
token: b                lexical unit: VARNAME
token: >                lexical unit: GT
```

```
token: c              lexical unit: VARNAME
token: )              lexical unit: RPAREN
token: THEN           lexical unit: THEN
token: \n             lexical unit: ENDLINE
token: \n             lexical unit: ENDLINE
token: PRINT          lexical unit: PRINT
token: (              lexical unit: LPAREN
token: b              lexical unit: VARNAME
token: )              lexical unit: RPAREN
token: \n             lexical unit: ENDLINE
token: ELSE           lexical unit: ELSE
token: \n             lexical unit: ENDLINE
token: \n             lexical unit: ENDLINE
token: input          lexical unit: VARNAME
token: :=             lexical unit: ASSIGN
token: input          lexical unit: VARNAME
token: +              lexical unit: PLUS
token: 1              lexical unit: NUMBER
token: \n             lexical unit: ENDLINE
token: ENDIF          lexical unit: ENDIF
token: \n             lexical unit: ENDLINE
token: ENDWHILE       lexical unit: ENDWHILE
token: \n             lexical unit: ENDLINE
token: ENDPROG        lexical unit: ENDPROG
token: \n             lexical unit: ENDLINE

Variables
a 6
b 7
c 10
input 8
```

Test.output