INFO-F-403

PART 1:
LEXICAL ANALYSIS

EL MIRI Hamza
000479603

24/10/2022

Contents

1 Introduction

Part 1[1] of this project aims to design and write a lexical analyzer that will be part of the compiler for the language Fortress. The tool *JFlex* was used and thus, the working language will be *Java*.
The lexical analyzer will parse the input file into meaningful tokens and return them to the parser, the latter being the subject of the next part of this project.
This document will go over the lexical structure of Fortress, some of the challenges faced and finally, some test examples.

2 Regular Expressions

The order in which the regular expressions are listed in the *JFlex* file is important. The first expression that matches will trigger its action. This is important to note because, for example, the invalid tokens expression will interfere with the valid numbers rule and therefore, the former must be situated below the latter to give it matching priority.

2.1 Variables and numbers

The language supports alphanumeric variable names in lowercase only (denoted in the specification below as "Identifier"), integers and decimal number notation. *JFlex* macros were used for ease of readability.

```
Lowercase       = [a-z]
Uppercase       = [A-Z]
Letter          = {Lowercase}|{Uppercase}
Numeric         = [0-9]
Integer         = ([1-9][0-9]*)|0
Identifier      = {Lowercase}({Lowercase}|{Numeric})*

ProgName        = {Uppercase}+({Numeric}|{Letter})*
```

Thus, the variable name can consist of only a single lowercase letter or a lowercase letter followed by an arbitrary number of alphanumeric lowercase characters while the program name must start with an uppercase character. Fortress will support integer numbers only.

[1] This is my second time doing the project so the first report will be mostly identical to last year's (since I got a good grade) with a few adjustments related to Fortress.

2.2 Reserved keywords and operators

Matching the following reserved keywords is trivial.

```
    "BEGIN", "END", "WHILE", "IF",
    "ELSE", "PRINT", "THEN", "DO", "READ"

"(", ")"           = Parentheses
"+", "-", "/", "*" = Addition, Subtraction,
                     Division and Multiplication respectively
":=", "="          = Assignment and Comparison operators
">", "<"           = "Greater than" and "Lesser than"
","                = Instruction break
```

To handle the case where the user wants to write completely unreadable code without whitespaces when they want to make life more difficult to whoever is going to read it, `/.*` is appended to the regular expression to tell the lexer to match the keyword even if it is part of a longer token. This correctly parses cases such as `IFcondition THENWHILEcondition DOvariable:=1,ENDEND`. Note that a VARNAME or NUMBER cannot be immediately followed by a reserved keyword without a whitespace because the grammar does not allow it (only operator symbols, comma or closed parenthesis) so this case will not be handled by the lexer.

The only problematic keyword is `END` because it is used to mark the end of `BEGIN`, `WHILE` and `IF` blocks so we must find a way to differentiate between LexicalUnits `END` and `ENDIF`.
The solution was to use a Stack to keep track of whether we entered a BEGIN/WHILE block or an IF block. When the Lexical Analyzer enters a BEGIN or WHILE block, a boolean value `true` is pushed onto the stack while `false` is for IF blocks. When a END is encountered, we pop the value on top of the stack which allows us to know to which block this unit belongs to. This is a simple but effective way of handling nested WHILE/IF blocks.

2.3 Invalid token handling

The analyzer uses a combination of regular expressions and states to determine invalid tokens.

```
Decimal          = \.[0-9]*
Real             = {Numeric}{Decimal}?
InvalidTokens    = {Real}\w* | {Letter}\w*{Uppercase}*\w*
Any              = .
```

`InvalidTokens` defines which tokens are to be discarded:

- `{Real}\w*` Matches any token starting with a number followed by an arbitrary number of any other character and decimal numbers. This rule also matches valid numbers but since it is applied *after* the valid number rule (`Real`), it has lower priority, thus, only non-valid numbers will be matched.
- `{Letter}\w*{Uppercase}*\w*` This matches any token containing an uppercase letter to identify invalid variable names. This also matches PROGNAME but it will be handled through states.
  - Starting state YYINITIAL: In this state, only the keyword BEGIN and PROGNAME will be matched while everything else will cause an exception to be thrown. Once PROGNAME is matched, the analyzer transitions to the PROGBODY state.
  - PROGBODY: In this state, everything else will be matched. This allows the rule `InvalidTokens` to *not* match PROGNAME but catch any variable name that contains an uppercase letter in addition to non-alphanumeric characters.
- `Any`: The last rule in the file, it will match any character not covered by the previous rules and throw an exception.

2.3.1 Exceptions

When an invalid token is encountered, a `LexicalException` is thrown by the analyzer which, although limited in functionality, it informs the user of which token caused it and suggests a fix.

2.4 Comment handling

Fortress handles two types of comments: Single-line comments that must be situated at the end of a line or in a line by themselves and multi-line comments which are anything found between two "%%". Multi-line comments can co-exist on the same line as the code as long as the first "%%" is situated after a line break "," and the second "%%" is situated before the next line of code.
The expression %%~%% will match everything in between "%%" and "%%" and will be discarded by the analyzer.

```
SingleComment    = %%.*
MultiLineComment = %%~%%
Comment          = {SingleComment}|{MultiLineComment}
```

Finally, single-line comments can be nested inside a multi-line comment but multi-line comments cannot be nested into each other. This is due to the fact that the analyzer cannot tell the difference between the "%%" marking the start of a comment and the "%%" marking the end. Let's look at an an example:

```
%%
    This is a comment example
    %% this is a nested comment %%
%%
```

If the analyzer were to come across this, it would recognize and discard the following as a comment:

```
 %%
    This is a comment example
    %%
```

Because it is delimited by two "%%". Then, it would match each token in `this is a nested comment` as a variable identifier and finally,

```
                                        %%
%%
```

Would be discarded as a comment.
To fix this, we can differentiate between the start and end of a comment in the specification of the language. For example:

```
/%
    This is a comment example
    /% this is a nested comment %/
%/
```

But the RegExp = /%~%/ alone would not be able to handle them as applying the rule of the longest match, the analyzer would match:

```
/%
    This is a comment example
    /% this is a nested comment %/
```

Leaving a trailing "%/" that would throw a lexical error.
The fix is to add more functionality to the analyzer. We could for instance, handle the comments the same way a parser would handle open and closed parentheses: First, match a "/%" and increase a counter by 1 then transition to

a COMMENT state. In this state, characters are matched 1 by 1. Encountering a "/%" increases the counter by 1 and a "/%" decreases it by 1. We reach the end of a comment when the counter is equal to 0. Everything in between was correctly discarded and we transition back to the previous state. If we reach the end of the file, it means that the nested comment wasn't closed properly and an exception could be thrown.

2.5 Other

The line terminator expression `\r|\n|\r\n` will be able to handle both LF and CRLF line terminators. This was needed to explicitly tell *JFlex* to match and consume blank lines because it does not do it by default. Without it, the analyzer would sometimes get stuck in a loop when encountering a line terminator. The expressions `{Blank}` and `{LineTerminator}` fixe the problem by matching and consuming blank spaces.

3 Test files

You'll find in the test folder multiple files written to test specific cases. For example, the `NestedWHILEIF.fs` file tests whether the stack structure properly tracks END statements. On the other hand, `AdditionalEND.fs` tests for a trailing END without a matching BEGIN, WHILE or IF before it among other specific test cases for each expression.

```
BEGIN NestedWhileIf
    WHILE condition DO
        IF condition2 THEN
            WHILE condition3 DO
                something := somethingelse ,
            END
        END
    END
END
```

Figure 1: NestedWHILEIF.fs contents