UNIVERSITI KEBANGSAAN MALAYSIA

The National University of Malaysia

TTTK 3163 Compiler Construction

Assignment 2- Lexical Analysis using JFlex

Name: Oh Shi Chew

Matric No : A190476

Instructor: Dr. Bahari Idrus

Content

Introduction

Lexical analysis or tokenization in computer science is the process of transforming a sequence of characters (programming language) into a sequence of lexical tokens. There are many types of tokens exist in programming languages. The common tokens are identifier, keyword, operator, delimiter, string literal and number literal. To identify the token type, regular expression is applied to identify the token type. Regular expression is also known as a rational expression is a sequence of characters that describes a match pattern in text. For example, the identifier is started with letters or underscore then followed by digits or letters or underscore.Therefore,the string can be generalized as a regular regression which is (letters_)(letters_|digit)*.For the compiler to recognize the pattern of the token, the regular expression needs to be converted to the Deterministic Finite Automata (DFA). DFA is a finite-state machine that accepts or rejects a given string of symbols by traversing a state sequence that is unique to the string.

JFlex is a Java-based lexical analyzer generator. The lexical analyzer generator will receive a specification containing a set of regular expressions and actions as input. JFlex lexers are based on deterministic finite automata (DFA). They are fast, without expensive backtracking.As output, the Java program will be generated by JFlex based on the lexical specifications.

Ruby is a high-level, general-purpose, interpreted programming language that supports a variety of paradigms. The programming language is created by Yukihiro "Matz" Matsumoto in Japan in the mid-1990s. Although Ruby is open-source and freely accessible on the Internet, it is bound by a licence. Ruby has similar syntax to that of many programming languages such as C++ and Perl. Ruby on Rails is a full-stack open-source framework based on Ruby language and designed primarily for the development of various web applications. For this assignment,the goal is to implement the JFlex to create a lexical analyzer to identify the token of the Ruby on Rails.

Methodology

Before we start to apply the Jflex, we need to download the Jflex and setup the Jflex settings. The Jflex that I used is JFlex 1.9.1 which is the latest version. After we have downloaded our JFlex,we need set PATH variable to the bin file of the JFlex. Also, we need to edit the batch file of the JFlex. Next, we need to ensure that the Java Development Kit (JDK) is installed to compile the Java program.

Next, we need to write a lexical specification in flex file. First, we need to study the format of the flex file to create lexical specification. We also need to study the lexical structure of Ruby language. For me, I import java_cup class in the lexical specification. Therefore, I create a file called sym.java which contains all the token type to be matched.

After the flex file is completed, we need to find at least 3 input which is the sample ruby code from internet. Next, we need to compile the java program in command prompt or terminal. Lastly, we test the input and observe the output. The output must list all the tokens correctly.

Basic Lexical Structure of Ruby

Keywords/Reserve words

begin	def	ensure	self	when	nil
end	defined	false	not	super	while
alias	do	for	or	then	yield
and	else	if	redo	true	
class	in	rescue	undef		

Operators

Symbol	Token Type	Output show in program
=	Assignment operator	EQ
==	Equal operator	EQEQ
!=	Not equal operator	NEQ
>	Greater than operator	GT
<	Less than operator	LT
>=	Greater than or equal to operator	GE
<=	Less than or equal to operator	LE
<=>	Combined comparison operator	C_COMPARISON
&&	AND operator	AND_OP
!	NOT operator	NOT_OP
||	OR operator	OR_OP
+	Plus sign	PLUS
-	Minus sign	MINUS
*	Multiply sign	MULTIPLY

Symbol	Token Type	Output show in program
/	Divide sign	DIVIDE
**	Exponent	EXP
%	Modulus	MOD
+=	Plus assignment	PLUS_AS
-=	Minus assignment	MINUS_AS
*=	Multiply assignment	MULTIPLY_AS
/=	Divide assignment	DIVIDE_AS
%=	Modulus assignment	MOD_AS
**=	Exponent assignment	EXP_AS
&	Bitwise AND	AND_BIT
|	Bitwise OR	OR_BIT
^	Bitwise XOR	XOR_BIT
~	Bitwise NOT	NOT_BIT
?	Ternary operator	QUESTION

Delimiters

Symbol	Token Type	Output show in program
[	Left bracket	LEFT_BRACKET
]	Right bracket	RIGHT_BRACKET
(	Left parenthesis	LEFT_PAREN
)	Right parenthesis	RIGHT_PAREN
{	Left brace	LEFT_BRACE
}	Right brace	RIGHT_BRACE
.	Dot	DOT
;	Semicolon	SEMICOLON
,	Comma	COMMA
:	Colon	COLON

Sigil

Symbol	Token Type	Output show in program
$	Global variable	DOLLAR
@	Instance variable	AT
@@	Class variable	ATAT

Others

Symbol	
1234	Number literal
“String”	String Literal
#This is a comment	Single line comment

Implementation of JFlex as lexical analyzer

Edit flex File

User code

```
import java_cup.runtime.Symbol;

/** Modified as a Ruby lexical analyzer */

%%

%public
%class Lexer
%unicode
%cup
%line
%column
%debug
%throws UnknownCharacterException

%{
  StringBuffer string = new StringBuffer();

  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }
%}
```

In this part, we specify what should be generated in our java output.

Code segment	Function
%public	Make the generated class public
%class Lexer	Set the class name as Lexer
%unicode	generated scanner will use the full Unicode input character set
%cup	enables CUP compatibility mode
%line	Turns line counting on
%column	Turns column counting on
%debug	Create main function
%throws UnknownCharacterExecption	Throws an UnknownCharacterExecption which is user defined when there is an error

The below code declare a StringBuffer string in which we will store parts of string literals and two helper functions symbol that create java_cup.runtime.Symbol objects with position information of the current token.

```
ALPHA=[A-Za-z]
DIGIT=[0-9]
NONNEWLINE_WHITE_SPACE_CHAR=[\ \t\b\012]
NEWLINE=\r|\n|\r\n
WHITE_SPACE_CHAR=[\n\r\ \t\b\012]

Identifier = {ALPHA}({ALPHA}|{DIGIT}|_)*
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]
DecIntegerLiteral = 0 | [1-9][0-9]*
/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}

TraditionalComment   = "/*" [^*] ~"*/" | "/*" "*"+ "/"
// Comment can be the last line of the file, without line terminator.
EndOfLineComment     = "#" {InputCharacter}* {LineTerminator}?
```

In the next part, we enter the regular expression and the actions when the regular expression is matched.

From the figure above, we have declared some important regular expression such as identifier, number literal and single line comment.

```
/* keywords */
<YYINITIAL> "begin"          { return symbol(sym.BEGIN); }
<YYINITIAL> "end"           { return symbol(sym.END); }
<YYINITIAL> "alias"             { return symbol(sym.ALIAS); }
<YYINITIAL> "and"             { return symbol(sym.AND); }
<YYINITIAL> "break"             { return symbol(sym.BREAK); }
<YYINITIAL> "case"             { return symbol(sym.CASE); }
<YYINITIAL> "class"             { return symbol(sym.CLASS); }
<YYINITIAL> "def"             { return symbol(sym.DEF); }
<YYINITIAL> "defined"             { return symbol(sym.DEFINED); }
```

```
/* operators and delimiters */
"="                             { return symbol(sym.EQ); }
"=="                            { return symbol(sym.EQEQ); }
"+"                             { return symbol(sym.PLUS); }
"{"                             { return symbol(sym.LEFT_BRACE); }
"}"                             { return symbol(sym.RIGHT_BRACE); }
"("                             { return symbol(sym.LEFT_PAREN); }
")"                             { return symbol(sym.RIGHT_PAREN); }
"."                             { return symbol(sym.DOT); }
";"                             { return symbol(sym.SEMICOLON); }
"["                             { return symbol(sym.LEFT_BRACKET); }
"]"                             { return symbol(sym.RIGHT_BRACKET); }
"%"                             { return symbol(sym.MOD); }
```

Next, we will declare all the keywords in the Ruby language. In the <YYINITIAL>, if the token is matched, the token type will be the return value. For the operators, delimiters, sigils and identifiers the process is same as the figure shown.

```
%state STRING
```

```
\"                              { string.setLength(0); yybegin(STRING); }
```

```
<STRING> {
  \"                             { yybegin(YYINITIAL);
                                   return symbol(sym.STRING_LITERAL,
                                   string.toString()); }
  [^\n\r\"\\]+                   { string.append( yytext() ); }
  \\t                            { string.append('\t'); }
  \\n                            { string.append('\n'); }

  \\r                            { string.append('\r'); }
  \\\"                           { string.append('\"'); }
  \\                             { string.append('\\'); }
}
```

These code segment take place in determining whether the token is string literal. When the scanner read left quote (“), it will start checking the <STRING> code snippet. After the checking and the scanner read the right quote (”),the scanner can determine that it is a string literal and return the value of the string.

```
/* comments */
{Comment}                        { /* ignore */ }

/* whitespace */
{WhiteSpace}                     { /* ignore */ }
```

The comments and the whitespace will be ignored in the program.

```
/* error fallback */
[^]                              { throw new UnknownCharacterException(yytext()); }
```

When there is an error, the expection will be thrown based on the user defined class.

```
class UnknownCharacterException extends Exception {
  UnknownCharacterException(String unknownInput) {
    super("Unknown character « " + unknownInput + " »");
  }
}
```

Apply JFlex

```
C:\Users\acer>cd C:\jflex-1.9.1\examples\cup-java-minijava\src\main\jflex

C:\jflex-1.9.1\examples\cup-java-minijava\src\main\jflex>jflex ruby.flex
Reading "ruby.flex"

Warning: Macro "WHITE_SPACE_CHAR" has been declared but never used.
Constructing NFA : 442 states in NFA
Converting NFA to DFA :
..........................................................................
..........................................................................
208 states before minimization, 188 states in minimized DFA
Writing code to "Lexer.java"

C:\jflex-1.9.1\examples\cup-java-minijava\src\main\jflex>
```

In the command prompt, we move the current working directory to the folder which contains the flex file. After that, we apply JFlex to convert the flex file to the Java program.

Generated java file

Lexer.java - Notepad

File Edit Format View Help

```
// DO NOT EDIT
// Generated by JFlex 1.9.1 http://jflex.de/
// source: ruby.flex

/*
 * JFlex example from the user Manual
 *
 * Copyright 2020, Gerwin Klein, Régis Décamps, Steve Rowe
 * SPDX-License-Identifier: GPL-2.0-only
 */

import java_cup.runtime.Symbol;

/** Modified as a Ruby lexical analyzer */


@SuppressWarnings("fallthrough")
public class Lexer implements java_cup.runtime.Scanner {

  /** This character denotes the end of file. */
  public static final int YYEOF = -1;

  /** Initial size of the lookahead buffer. */
  private static final int ZZ_BUFFERSIZE = 16384;
  private static final String ZZ_NL = System.getProperty("line.separator");

  // Lexical states.
  public static final int YYINITIAL = 0;
  public static final int STRING = 2;

  /**
   * ZZ_LEXSTATE[l] is the state in the DFA for the lexical state l
   * ZZ_LEXSTATE[l+1] is the state in the DFA for the lexical state l
   *                  at the beginning of a line
   * l is of the form l = 2*k, k a non negative integer
   */
  private static final int ZZ_LEXSTATE[] = {
     0,  0,  1, 1
  };

  /**
   * Top-level table for translating characters to character classes
```

Results

Compile the Java program

```
C:\Users\acer>cd C:\jflex-1.9.1\examples\JFLEX_Assignment\Assignment2

C:\jflex-1.9.1\examples\JFLEX_Assignment\Assignment2>javac Lexer.java
```

Test first input

input1.txt

```
#!/usr/bin/ruby
x = 1
unless x>=2
   puts "x is less than 2"
 else
   puts "x is greater than 2"
end
```

Output:

```
C:\jflex-1.9.1\examples\JFLEX_Assignment\Assignment2>java Lexer input1.txt
line: 1 col: 1 match: --#!/usr/bin/ruby\u000D\u000A--
action [156] { /* ignore */ }
line: 2 col: 1 match: --x--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 2 col: 2 match: -- --
action [156] { /* ignore */ }
line: 2 col: 3 match: --=--
action [109] { return symbol(sym.EQ); }
#8
line: 2 col: 4 match: -- --
action [156] { /* ignore */ }
line: 2 col: 5 match: --1--
action [105] { return symbol(sym.INTEGER_LITERAL); }
#6
line: 2 col: 6 match: -- --
action [156] { /* ignore */ }
line: 2 col: 7 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 3 col: 1 match: --unless--
action [93] { return symbol(sym.UNLESS); }
```

```
line: 3 col: 7 match: -- --
action [156] { /* ignore */ }
line: 3 col: 8 match: --x--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 3 col: 9 match: -->=--
action [124] { return symbol(sym.GE); }
#56
line: 3 col: 11 match: --2--
action [105] { return symbol(sym.INTEGER_LITERAL); }
```

```
#6
line: 3 col: 12 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 4 col: 1 match: --    --
action [157] {   }
line: 4 col: 4 match: --puts--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 4 col: 8 match: -- --
action [156] { /* ignore */ }
line: 4 col: 9 match: --"--
action [106] { string.setLength(0); yybegin(STRING); }
line: 4 col: 10 match: --x is less than 2--
action [166] { string.append( yytext() ); }
line: 4 col: 26 match: --"--
action [163] { yybegin(YYINITIAL);
                                   return symbol(sym.STRING_LITERAL,
                                   string.toString()); }
#7
line: 4 col: 27 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 5 col: 1 match: -- --
action [156] { /* ignore */ }
line: 5 col: 2 match: --else--
action [72] { return symbol(sym.ELSE); }
```

```
#26
line: 5 col: 6 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 6 col: 1 match: --    --
action [157] {   }
line: 6 col: 4 match: --puts--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 6 col: 8 match: -- --
action [156] { /* ignore */ }
line: 6 col: 9 match: --"--
action [106] { string.setLength(0); yybegin(STRING); }
line: 6 col: 10 match: --x is greater than 2--
action [166] { string.append( yytext() ); }
line: 6 col: 29 match: --"--
action [163] { yybegin(YYINITIAL);
                                   return symbol(sym.STRING_LITERAL,
                                   string.toString()); }
#7
line: 6 col: 30 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 7 col: 1 match: --end--
action [63] { return symbol(sym.END); }
#3
#0
```

Lexeme	Token type
x	Identifier
=	Assignment operator
1	Number literal
unless	Keyword
>=	Relational operator
2	Number literal
puts	Identifier (Built in function)
"x is less than 2"	String literal
else	Keyword
"x is greater than 2"	String literal
end	Keyword

Test second input

input2.txt

```
def add_three(number)
  number + 3
end

returned_value = add_three(4)
puts returned_value
```

Output:

```
line: 1 col: 1 match: --def--
action [69] { return symbol(sym.DEF); }
#23
line: 1 col: 4 match: -- --
action [156] { /* ignore */ }
line: 1 col: 5 match: --add_three--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 1 col: 14 match: --(--
action [114] { return symbol(sym.LEFT_PAREN); }
#13
line: 1 col: 15 match: --number--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 1 col: 21 match: --)--
action [115] { return symbol(sym.RIGHT_PAREN); }
```

```
#14
line: 1 col: 22 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 2 col: 1 match: --	--
action [157] {	}
line: 2 col: 3 match: --number--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 2 col: 9 match: -- --
action [156] { /* ignore */ }
line: 2 col: 10 match: --+--
action [111] { return symbol(sym.PLUS); }
#10
line: 2 col: 11 match: -- --
action [156] { /* ignore */ }
line: 2 col: 12 match: --3--
action [105] { return symbol(sym.INTEGER_LITERAL); }
#6
line: 2 col: 13 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 3 col: 1 match: --end--
action [63] { return symbol(sym.END); }
#3
line: 3 col: 4 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 4 col: 1 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 5 col: 1 match: --returned_value--
action [102] { return symbol(sym.IDENTIFIER); }
```

```
#5
line: 5 col: 15 match: -- --
action [156] { /* ignore */ }
line: 5 col: 16 match: --=--
action [109] { return symbol(sym.EQ); }
#8
line: 5 col: 17 match: -- --
action [156] { /* ignore */ }
line: 5 col: 18 match: --add_three--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 5 col: 27 match: --(--
action [114] { return symbol(sym.LEFT_PAREN); }
#13
line: 5 col: 28 match: --4--
action [105] { return symbol(sym.INTEGER_LITERAL); }
#6
line: 5 col: 29 match: --)--
action [115] { return symbol(sym.RIGHT_PAREN); }
#14
line: 5 col: 30 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 6 col: 1 match: --puts--
action [102] { return symbol(sym.IDENTIFIER); }
```

```
#5
line: 6 col: 5 match: -- --
action [156] { /* ignore */ }
line: 6 col: 6 match: --returned_value--
action [102] { return symbol(sym.IDENTIFIER); }
#5
#0
```

Lexeme	Token type
def	Keyword
add_three	Identifier
(	Left parenthesis
number	Identifier
)	Right parenthesis
+	Plus
3	Number literal
end	Keyword
returned_value	Identifier
4	Number literal
puts	Identifier(Built in function)

Test third input

input3.txt (Example input given in the task)

```
# !/usr/bin/ruby –w
$i = 0
$num = 0
time1 = Time.new
while $i < $num do
        puts ("Inside the loop i = #i")
end
puts "Current Time : " + time1.inspect
```

Output:

```
C:\jflex-1.9.1\examples\JFLEX_Assignment\Assignment2>java Lexer input3.txt
line: 1 col: 1 match: --# !/usr/bin/ruby \u2013w\u000D\u000A--
action [156] { /* ignore */ }
line: 2 col: 1 match: --$--
action [147] { return symbol(sym.DOLLAR); }
#79
line: 2 col: 2 match: --i--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 2 col: 3 match: -- --
action [156] { /* ignore */ }
line: 2 col: 4 match: --=--
action [109] { return symbol(sym.EQ); }
#8
line: 2 col: 5 match: -- --
action [156] { /* ignore */ }
line: 2 col: 6 match: --0--
action [105] { return symbol(sym.INTEGER_LITERAL); }
```

```
#6
line: 2 col: 7 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 3 col: 1 match: --$--
action [147] { return symbol(sym.DOLLAR); }
#79
line: 3 col: 2 match: --num--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 3 col: 5 match: -- --
action [156] { /* ignore */ }
line: 3 col: 6 match: --=--
action [109] { return symbol(sym.EQ); }
#8
line: 3 col: 7 match: -- --
action [156] { /* ignore */ }
line: 3 col: 8 match: --0--
action [105] { return symbol(sym.INTEGER_LITERAL); }
#6
line: 3 col: 9 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 4 col: 1 match: --time1--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 4 col: 6 match: -- --
action [156] { /* ignore */ }
```

```
line: 4 col: 7 match: --=--
action [109] { return symbol(sym.EQ); }
#8
line: 4 col: 8 match: -- --
action [156] { /* ignore */ }
line: 4 col: 9 match: --Time--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 4 col: 13 match: --.--
action [116] { return symbol(sym.DOT); }
#15
line: 4 col: 14 match: --new--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 4 col: 17 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 5 col: 1 match: --while--
action [96] { return symbol(sym.WHILE); }
#50
line: 5 col: 6 match: -- --
action [156] { /* ignore */ }
line: 5 col: 7 match: --$--
action [147] { return symbol(sym.DOLLAR); }
#79
line: 5 col: 8 match: --i--
action [102] { return symbol(sym.IDENTIFIER); }
```

```
#5
line: 5 col: 9 match: -- --
action [156] { /* ignore */ }
line: 5 col: 10 match: --<--
action [123] { return symbol(sym.LT); }
#55
line: 5 col: 11 match: -- --
action [156] { /* ignore */ }
line: 5 col: 12 match: --$--
action [147] { return symbol(sym.DOLLAR); }
#79
line: 5 col: 13 match: --num--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 5 col: 16 match: -- --
action [156] { /* ignore */ }
line: 5 col: 17 match: --do--
action [71] { return symbol(sym.DO); }
#25
line: 5 col: 19 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 6 col: 1 match: --\u0009--
action [156] { /* ignore */ }
line: 6 col: 2 match: --puts--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 6 col: 6 match: -- --
action [156] { /* ignore */ }
```

```
line: 6 col: 7 match: --(--
action [114] { return symbol(sym.LEFT_PAREN); }
#13
line: 6 col: 8 match: --"--
action [106] { string.setLength(0); yybegin(STRING); }
line: 6 col: 9 match: --Inside the loop i = #i--
action [166] { string.append( yytext() ); }
line: 6 col: 31 match: --"--
action [163] { yybegin(YYINITIAL);
                                  return symbol(sym.STRING_LITERAL,
                                  string.toString()); }
#7
line: 6 col: 32 match: --)--
action [115] { return symbol(sym.RIGHT_PAREN); }
#14
line: 6 col: 33 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 7 col: 1 match: --end--
action [63] { return symbol(sym.END); }
#3
line: 7 col: 4 match: --\u000D\u000A--
action [156] { /* ignore */ }
line: 8 col: 1 match: --puts--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 8 col: 5 match: -- --
action [156] { /* ignore */ }
line: 8 col: 6 match: --"--
action [106] { string.setLength(0); yybegin(STRING); }
line: 8 col: 7 match: --Current Time : --
```

```
action [166] { string.append( yytext() ); }
line: 8 col: 22 match: --"--
action [163] { yybegin(YYINITIAL);
                                   return symbol(sym.STRING_LITERAL,
                                   string.toString()); }
#7
line: 8 col: 23 match: -- --
action [156] { /* ignore */ }
line: 8 col: 24 match: --+--
action [111] { return symbol(sym.PLUS); }
#10
line: 8 col: 25 match: -- --
action [156] { /* ignore */ }
line: 8 col: 26 match: --time1--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 8 col: 31 match: --.--
action [116] { return symbol(sym.DOT); }
#15
line: 8 col: 32 match: --inspect--
action [102] { return symbol(sym.IDENTIFIER); }
#5
line: 8 col: 39 match: --\u000D\u000A--
action [156] { /* ignore */ }
#0
```

Lexeme	Token Type
$	Sigil(global variable)
i	Identifier
=	Assignment operator
0	Number literal
num	Identifier
time1	Identifier
Time	Identifier
.	Delimiter(dot)
new	Identifier
while	Keyword
<	Relational operator (Less than)

do	Keyword
puts	Identifier(Built in function)
(	Left parenthesis
"Inside the loop i = #i"	String Literal
)	Right parenthesis
end	Keyword
"Current Time : "	String Literal
+	Plus sign
inspect	Identifier

Conclusion

To summarise, learning JFlex for lexical analysis can be a very useful way to get knowledge about tokenizing and scanning source code. JFlex is an effective tool for creating effective regular expression-based lexical analyzers. By defining lexical specification in flex files, we can create scanners that effectively tokenize input streams. Furthermore, the study of JFlex emphasises the significance of understanding a programming language's lexical structure. By studying the lexical structure of programming languages,so that we are able to create an exact and precise lexical analyzer.

References:

1. Ruby syntax

   https://ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/syntax.html

2. JFlex User’s Manual

   http://web.cs.unlv.edu/CSC460/docs/JFlex/manual.html#SECTION00052500000000000000