Amirkabir University of Technology
(Tehran Polytechnic)

Department of Computer Engineering and Information Technology

Course Project

Principles of Compiler Design
Phase one
Lexical Analyzer

By
Nima Davari

Professor
Dr. Mohammadreza Razzazi

Fall 2019

Overall description

As for the first phase of the project, the students are expected to implement a fully functional Lexical Analyzer that is able to separate tokens and return corresponding values to the detected patterns. The Analyzer should be able to distinguish between correct or unknown lexemes and continue the tokenization even after an error is reported.

Goals

1. Setup a Lex project.
2. Come up with Regular Expressions for all symbols used in the Language.
3. Enter the Regular Expressions in the Lex file.
4. Create a representation method for the output of the Analyzer by generating a table of tokens, their corresponding token values, and their actual values if they are numbers or strings.

Language

The language comprises the following symbols. For every entry of the table, a Regular Expression must be entered in the Lex file.
The token value is an integer number that determines what type of token is detected.

Lexeme	Token Value
Whitespace	-
Identifier	TOKEN_ID
Integer number	TOKEN_INTEGER
Real number	TOKEN_REAL
String	TOKEN_STRING
Comment	-
“class”	TOKEN_CLASS
“reference”	TOKEN_REFERENCE
“static”	TOKEN_STATIC
“int”	TOKEN_INT_TYPE
“real”	TOKEN_REAL_TYPE
“bool”	TOKEN_BOOL_TYPE
“string”	TOKEN_STRING_TYPE
“void”	TOKEN_VOID
“true”	TOKEN_TRUE
“false”	TOKEN_FALSE
“print”	TOKEN_PRINT
“return”	TOKEN_RETURN

“break”	TOKEN_BREAK
“continue”	TOKEN_CONTINUE
“if”	TOKEN_IF
“else”	TOKEN_ELSE
“elseif”	TOKEN_ELSEIF
“while”	TOKEN_WHILE
“for”	TOKEN_FOR
“to”	TOKEN_TO
“in”	TOKEN_IN
“steps”	TOKEN_STEPS
“&”	TOKEN_BITWISE_AND
“&&”	TOKEN_AND
“|”	TOKEN_BITWISE_OR
“||”	TOKEN_OR
“!”	TOKEN_NOT
“~”	TOKEN_BITWISE_NOT
“>>”	TOKEN_SHIFT_RIGHT
“<<”	TOKEN_SHIFT_LEFT
“=”	TOKEN_ASSIGNMENT
“+”	TOKEN_ADDITION
“-“	TOKEN_SUBTRACTION
“*”	TOKEN_MULTIPLICATION
“/”	TOKEN_DIVISION
“%”	TOKEN_MODULO
“^”	TOKEN_POWER
“>”	TOKEN_GT
“>=”	TOKEN_GE
“<”	TOKEN_LT
“<=”	TOKEN_LE
“==”	TOKEN_EQ
“!=”	TOKEN_NE
“{”	TOKEN_LCB
“}”	TOKEN_RCB
“(”	TOKEN_LP
“)”	TOKEN_RP
“.”	TOKEN_DOT
“;”	TOKEN_SEMICOLON
“,”	TOKEN_COMMA
Error	-

Special Symbols

- **Whitespace:** A whitespace is any sequence of characters that are spaces, tabs or line feeds.

- **Identifier:** An identifier must have an odd number of characters. They can contain alphabetical characters, numerical characters or underscores; however, an ID cannot start with a numerical character. A second condition is that the distance between each underscore and the next one, must be exactly two characters.

  The following characters are accepted:
  - Celeste
  - _C_eleste
  - _1_2_3_

  The following characters are not accepted:
  - _Celeste (Even number of characters)
  - 2oothpaste_ (Starting with a digit)
  - _12345_ (The gap between underscores exceeds the limit)

- **Integer number:** An integer does not have leading zeros. These numbers come in three different representations.
  - Binary: +0b110110 Real value = 54
  - Decimal: -22 Real value = -22
  - Hexadecimal: 0x44Aa5 Real value = 281253

  The existence of a sign is not mandatory.

- **Real number:** A real number does not have leading zeros or trailing zeros.

- **String:** A string is any sequence of characters between two double quotation symbols. Concatenation of strings must also be supported and the result should be returned as a single String token.
  “Gotta go” + “ fast!” Real value = “Gotta go fast!”

- **Comment:** There are two types of comments.
  - **Single line comments**

    num = a + b; // This is a comment.

  - **Multiline comments:**

    /* This is a comment
    as well. */

- **Error:** Any meaningless token that is not matched with any other Regular Expression, must match the Error Regular Expression.