Sathyanarayana Ramesh

Lexical Analyzer

PHASE 1: SPECIFICATION:

1.1 Language Specification:

The self-created language, known as Cminus, has the attributes listed below:

- This is a case-sensitive language that supports ASCII characters.
- Supported data types: standard int and a standard float data type.
- Supported Operators: arithmetic, logical, relational, and assignment operators.
- Programming constructs: if-else, while, and compound statements for control flow.
- Comments: supports single-line and multi-line comments that start with /* and end with */
- Identifiers: starts with a letter and can contain alphanumeric characters.
- Constants: supports literals that are enclosed in single quotes for strings, and supports only decimal notation for floating point numbers.
- Keywords: are the reserved words which have special meaning and cannot used as identifier. The keywords are: int, float, if, else, exit, while, read, write, and return.

1.2 Lexical Analyzer Specification:

Lexical analyzer is the important component of compiler and it's primary function is to break down the source code of a program into a sequence of smaller units called tokens. The lexical analyzer reads the input source code character by character, recognizing and categorizing characters into tokens based on the language's syntax rules. The Supported token types are as follows:

A. KEYWORD: A reserved word in the language.
B. IDENTIFIER: A user-defined name for a variable or a function.
C. CONSTANT: A constant value of a data type.
D. ARITH-OP: A symbol that performs an arithmetic operation on operands.
E. LOGIC-OP: A symbol that performs a logical operation on operands.
F. SEPARATOR: A symbol that separates tokens or groups them together.
G. COMMENT: A text that is ignored by the compiler and used for documentation purposes.

The programme reads the code of c minus written in a file, and the lexical analyzer is built and called which starts reading the input source code character by character, recognizing and categorizing characters into tokens based on the language's syntax rules. It also typically removes whitespace and comments, which are not needed for further processing by the compiler.

Source Code:

```
/* This is a multi-line
        comment*/
int c;
if (c == d) { IF = a; }
```

Tokens List

```
[('COMMENT', '/* This is a multi-line\n       comment*/'), ('KEYWORD', 'int'), ('IDENTIFIER',
'c'), ('SEPARATOR', ';'), ('KEYWORD', 'if'), ('SEPARATOR', '('), ('IDENTIFIER', 'c'), ('LOGICAL_OP',
```

'=='), ('IDENTIFIER', 'd'), ('SEPARATOR', ')'), ('SEPARATOR', '{'), ('IDENTIFIER', 'IF'), ('ARITHMETIC_OP', '='), ('IDENTIFIER', 'a'), ('SEPARATOR', ';'), ('SEPARATOR', '}')]

Testing the lexical analyzer using the following steps:

A. Source code which is free of lexical error, which should produce list of legal tokens.
B. Identifier which starts with digits, should be handled as illegal token.
C. Lexical analyser should handle the missing closing comment.

PHASE 2 DESIGN:

2.1 Functions and its purpose

This program uses PLY to build the lexical analyzer, PLY requires the token class names to be listed in tuple, later defined by the methods which starts with 't_' and the respective token names. Each token class to be defined separately and the order of token class is very much dependent for matching tokens and to catch the lexical error for the illegal character.

A. Comments token class - ***token t_COMMENT(string t)****:*
   This function accepts the string t, returns the token object for the matched regex. Comments are identified by /* and */. Regular expression - r'/*(.|\n)**/

B. Keyword token class - ***token t_KEYWORD(string t)****:*
   This function accepts the string t, returns the token object for the matched regex. Keywords are identified by list of reserved words.
   Regular expression -r'int|float|if|else|exit|while|read|write|return'

C. Identifier token class - ***token t_IDENTIFIER(string t):***
   This function accepts the string t, returns the token object for the matched regex. Identifier are identified by first occurance letter followed by letter or digit or _(under score).
   Regular expression - r'[a-zA-Z_][a-zA-Z0-9_]*'

D. Identifier error - ***token t_identifier_error(string t):***
   This functions accepts the string, identifiers which don't match to the above expression ie the identifiers which contains the special symbol or identifiers which starts with a digit are matched and handled as identifier error.
   Regular expression - r'\d+[a-zA-Z_][a-zA-Z0-9_]*|[a-zA-Z0-9_]*[^a-zA-Z0-9_]^'

E. Constant token class – ***token t_CONSTANT(string t):***
   This function accepts the string t, returns the token object for the matched regex. Keywords are identified by single digit optionally followed decimal point and other digits.
   Regular expression r'\d+\.\d+|\d+'

F. Arithmetic operator token class – ***t_AIRTHMETCI_OP*** matches for the arithmetic operators
   Regular expression r'\+|-|*|/|='

G. Logical operator token class – ***t_LOGICAL_OP*** matches for the logical operators
   Regular expression r'==|!=|<|>|<=|>=|&&|\|\|'

H. Separator token class – these are the character which separates tokens – ***t_SEPARATOR***

Regular expression r',|;|\(|\)|{|}'

I. ***t_ignore*** – ignores the tab spaces and new line character t_ignore = ' \t\n'

J. ***t_error*** - finite state if it doesn't match the lexeme for any accepting state will be caught as error.

2.2 Pseudocode:

```
// Declare the token classes
Tokens ← ('IDENTIFIER', 'KEYWORD', 'CONSTANT', 'ARITHMETIC_OP','LOGICAL_OP',
'SEPARATOR','COMMENT','MISSING_COMMENT')

// Define regular expressions for token matching
Define regular expression for multi-line comment (t_COMMENT)
Define regular expression for incomplete multi-line comment (t_MISSING_COMMENT)
Define regular expression for keywords (t_KEYWORD)
Define regular expression for identifiers (t_IDENTIFIER)
Define regular expression for constants (t_CONSTANT)
Define regular expressions for arithmetic operators (t_ARITHMETIC_OP)
Define regular expressions for logical operators (t_LOGICAL_OP)
Define regular expressions for separators (t_SEPARATOR)
// Ignore whitespace characters
Ignore whitespace, tabs, and newlines (t_ignore)
Define an error-handling function for invalid characters (t_error)
// Create the lexer
lexer ← CreateLexer()

// Open and read the source code file
fh ← Open the source code file
data ← Read the content of the file
Initialize the lexer with the data
Tokenize the input
 Initialize an empty list to store tokens (token_list)
 while True:
      Get the next token from the lexer (tok)
      If tok is None (no more tokens)
                  exit the loop
      end if
      Append (tok.type, tok.value) to token_list
  end while
  Print token_list
```

PHASE 3: RISK ANALYSIS

No risk.

PHASE 4: VERIFICATION

In order to successfully run the program the name of the source code file has to exactly match with the input file name and the input file has to be present in the current directory. The program has been tested several times in order to check build of lexical tokens and it successfully returns the list of matching tokens and their token class name, handles the lexical error.

PHASE 5: CODING

The code has been pushed to the git hub , and the link has been provided below.

https://github.com/akgWMU/pa3-lexical-analysis-Sathya-Ramesh

PHASE 6: TESTING

Testing of lexical analyzer

1) Source code free of errors

```
C:\Users\Sathya\OneDrive\Desktop\prgming_grad\assignment_3>python compiler.py
Source Code:
/* This is a multi-line
       comment*/
    int c;
    if (c == d) { IF = a; }
Tokens List
[('COMMENT', '/* This is a multi-line\n       comment*/'), ('KEYWORD', 'int'), ('IDENTIFIER', 'c'), ('SEPARATOR', ';'), ('KEYWORD',
'if'), ('SEPARATOR', '('), ('IDENTIFIER', 'c'), ('LOGICAL_OP', '=='), ('IDENTIFIER', 'd'), ('SEPARATOR', ')'), ('SEPARATOR', '{'), ('
IDENTIFIER', 'IF'), ('ARITHMETIC_OP', '='), ('IDENTIFIER', 'a'), ('SEPARATOR', ';'), ('SEPARATOR', '}')]
```

2) Source code which has missing closing comment

```
C:\Users\Sathya\OneDrive\Desktop\prgming_grad\assignment_3>python compiler.py
Source Code:
/* This is a multi-line
       comment*/
    int c;
    if (c == d) { IF = a; }
       /* missing close comment
Tokens List
[('COMMENT', '/* This is a multi-line\n       comment*/'), ('KEYWORD', 'int'), ('IDENTIFIER', 'c'), ('SEPARATOR', ';'), ('KEYWORD',
'if'), ('SEPARATOR', '('), ('IDENTIFIER', 'c'), ('LOGICAL_OP', '=='), ('IDENTIFIER', 'd'), ('SEPARATOR', ')'), ('SEPARATOR', '{'), ('
IDENTIFIER', 'IF'), ('ARITHMETIC_OP', '='), ('IDENTIFIER', 'a'), ('SEPARATOR', ';'), ('SEPARATOR', '}'), ('MISSING_COMMENT', '/* miss
ing close comment')]
```

3) Source code which has invalid identifier

```
C:\Users\Sathya\OneDrive\Desktop\prgming_grad\assignment_3>python compiler.py
Source Code:
/* This is a multi-line
       comment*/
    int c;
    if (c == d) { IF = a; }
       int 4afc;
Invalid Identifier '4afc'
Tokens List
[('COMMENT', '/* This is a multi-line\n       comment*/'), ('KEYWORD', 'int'), ('IDENTIFIER', 'c'), ('SEPARATOR', ';'), ('KEYWORD',
'if'), ('SEPARATOR', '('), ('IDENTIFIER', 'c'), ('LOGICAL_OP', '=='), ('IDENTIFIER', 'd'), ('SEPARATOR', ')'), ('SEPARATOR', '{'), ('
IDENTIFIER', 'IF'), ('ARITHMETIC_OP', '='), ('IDENTIFIER', 'a'), ('SEPARATOR', ';'), ('SEPARATOR', '}'), ('KEYWORD', 'int')]
```

4) Source code which has invalid character

```
C:\Users\Sathya\OneDrive\Desktop\prgming_grad\assignment_3>python compiler.py
Source Code:
/* This is a multi-line
       comment*/
    int c;
    if (c == d) { IF = a; }
       $@
Illegal character '$'
Illegal character '@'
Tokens List
[('COMMENT', '/* This is a multi-line\n       comment*/'), ('KEYWORD', 'int'), ('IDENTIFIER', 'c'), ('SEPARATOR', ';'), ('KEYWORD',
'if'), ('SEPARATOR', '('), ('IDENTIFIER', 'c'), ('LOGICAL_OP', '=='), ('IDENTIFIER', 'd'), ('SEPARATOR', ')'), ('SEPARATOR', '{'), ('
IDENTIFIER', 'IF'), ('ARITHMETIC_OP', '='), ('IDENTIFIER', 'a'), ('SEPARATOR', ';'), ('SEPARATOR', '}')]
```

5) Source code containing constants

```
Source Code:
/* This is a multi-line
        comment*/
    int c;
        float pi = 3.14;
    if (c == d) { IF = a; }
Tokens List
[('COMMENT', '/* This is a multi-line\n        comment*/'), ('KEYWORD', 'int'), ('IDENTIFIER', 'c'), ('SEPARATOR', ';'), ('KEYWORD',
'float'), ('IDENTIFIER', 'pi'), ('ARITHMETIC_OP', '='), ('CONSTANT', '3.14'), ('SEPARATOR', ';'), ('KEYWORD', 'if'), ('SEPARATOR', '(
'), ('IDENTIFIER', 'c'), ('LOGICAL_OP', '=='), ('IDENTIFIER', 'd'), ('SEPARATOR', ')'), ('SEPARATOR', '{'), ('IDENTIFIER', 'IF'), ('A
RITHMETIC_OP', '='), ('IDENTIFIER', 'a'), ('SEPARATOR', ';'), ('SEPARATOR', '}')]
```

PHASE 7: REFINING THE PROGRAM

The program currently reads the content of the source file, the source code file name is being statically passed. The program could be enhanced to accept the file name as an command line argument, hence forth avoiding the mis-spelled file name error. The compiler program contains the first component lexical analyzer, the same program could be enhanced to make use of this token stream, passed to the parser, which uses them to construct an abstract syntax tree (AST) or perform other forms of syntactic analysis.

PHASE 8: PRODUCTION

Folder containing ‘compiler.py’, input file ‘source_code.txt’ & report has been compressed to a zip file and the same has been added to the dropbox submission.