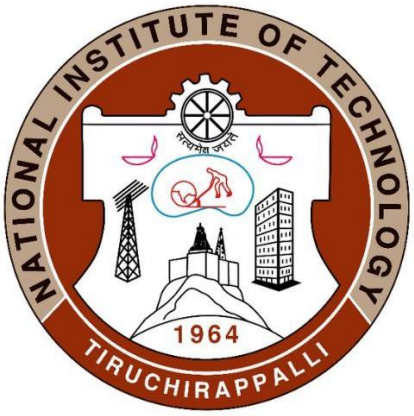
NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI



CSPC62

COMPILER DESIGN

TOPIC: Python Compiler

DONE BY:

S.No	Name	RollNo
1	Gokul Adethya T	106121045
2	Raghavan Balanathan	106121099
3	Srinivasa Raghavan P	106121129

LAB REPORT-1
Subtopic: Lexical Analyzer

A. Develop the components of your programming language.

B. Write regex for each of them and draw the corresponding DFA.

C. Write Lex code implementing the patterns and corresponding actions.

D. Write codes for handling errors during lexical analysis.

E. Compile the Lex code and create your own lexical analyser (L).

F. Write small programs in the language you have developed.

G. Compile your programs and show that your lexical analyser is creating correct tokens and handling errors correctly.

A. Develop the components of your programming language.

We define the following components for our Python-like language: -

Data Types: Data types specify the type that can be used in a programming language. Common data types include integers, floating-point numbers, characters, strings, arrays, and more complex types like structures and objects.

Variables: Variables are used to store data temporarily during the execution of a program. They have a name and a data type and can hold different values at different times during program execution.

Operators: Operators are symbols or keywords that perform operations on operands. They include arithmetic operators (+, -, *, /), relational operators (==, !=, <, >), logical operators (and, or, not), and others.

Control Structures: Control structures dictate the flow of execution within a program. They include conditional statements (if-else), loops (for, while), and branching statements (break, continue, return).

Functions/Methods: Functions or methods are blocks of code that perform a specific task. They allow for code reuse and modular programming. Functions typically accept input parameters, perform operations, and may return a result.

C. **Write Lex code implementing the patterns and corresponding actions.**

```
[0-9]+ {
    yylval.text = strdup(yytext); // Assign the matched text to yylval.text
    debug(yylval.text); // Print debug information for the number token
    return T_Number; // Return the token type T_Number
}

[_a-zA-Z][_a-zA-Z0-9]* {
    yylval.text = strdup(yytext); // Assign the matched text to yylval.text
    debug(yylval.text); // Print debug information for the identifier token
    return T_ID; // Return the token type T_ID
}

\"([^\"\n])*\" {
    yylval.text = strdup(yytext); // Assign the matched text to yylval.text
    debug(yylval.text); // Print debug information for the string token
    return T_String; // Return the token type T_String
}

\'([^\'\n])*\' {
    yylval.text = strdup(yytext); // Assign the matched text to yylval.text
    debug(yylval.text); // Print debug information for the string token
    return T_String; // Return the token type T_String
}

"#"([a-z]|[0-9]|[A-Z]|" ")* {
    // This rule matches comments starting with '#'. No action is taken.
    // Comments are usually ignored and not associated with any token.
}
{whitespace} {
    // This rule matches whitespace characters. No action is taken.
    // Whitespace characters are usually ignored in the tokenization process.
}
"\n" {
    yycolumn = 1;
    debug("NL");
    return T_NL;
}
<<EOF>> {
    debug("EOF");
    return T_EndOfFile;
}
%%
```

This section defines rules for matching keywords, operators, literals, etc., and assigns corresponding tokens. These rules match the newline characters, reset the column count, print debug information, and return the newline token and marks the end of the Flex rules section.

D. Write codes for handling errors during lexical analysis.

Lexical errors refer to mistakes or issues in the lexical analysis phase of a compiler or interpreter, where the input text is broken down into tokens. These errors occur when the lexer (often generated by tools like Lex) encounters input that does not conform to the defined grammar or rules of the language.

Some examples of lexical errors include:

- Invalid characters: Characters that are not recognized by the lexer or are not allowed in the language.

```
whitespace [ ]
Multiline_comment \'\'\'.+\'\'\'
```

- Illegal token sequences: Sequences of characters that do not form valid tokens in the language.

```
[_a-zA-Z][_a-zA-Z0-9]* {yylval.text = strdup(yytext); debug(yylval.text); return T_ID;}
```

- We impose restrictions on identifier length (256) and exceeding those limits could lead to a lexical error.

```
static void debug(const char *X){
    #ifdef DEBUG
    if(startFlag){ startFlag=0; }
    if(strcmp(X, "NL")==0){ printf("T_%s\n%d ", X, yylineno); }
    else{
        if(strlen(X) < 256){ printf("T_%s ", X); }
        else{ printf("Max length crossed: %s, Length: %d", X, strlen(X));exit(1); }
    }
    #endif
}
```

- Numeric format errors: Incorrectly formatted numbers, such as "123.456.789".

```
[0-9]+("."[0-9]+)?(E[+-]?[0-9]+)? {yylval.text = strdup(yytext); debug(yylval.text); return T_Number;}
```

When lexical errors are encountered, the lexer typically raises an error to indicate the issue. In Lex (or similar lexer generators), the yyerror function is often used to report errors.

E. Compile the Lex code and create your own lexical analyser (L).

```
"pass" {debug("Pass"); return T_Pass;}
"if" {debug("If"); return T_If;}
"else" {debug("Else"); return T_Else;}
"elif" {debug("Elif"); return T_Elif;}
"in" {debug("In"); return T_In;}
"while" {debug("While"); return T_While;}
"for" {debug("For"); return T_For;}
"range" {debug("Range"); return T_Range;}
"try" {debug("Try"); return T_Try;}
"except" {debug("Except"); return T_Except;}
"finally" {debug("Finally"); return T_Finally;}
```

Above lex code uses reserved keywords to give priority over identifiers. The debug function is used for debugging purposes, specifically to print information about the tokens encountered during the lexing process and returns the corresponding token identifier for the parser.

```
"and" {debug("And"); return T_And;}
"or" {debug("Or"); return T_Or;}
"not" {debug("Not"); return T_Not;}

"True" {debug("True"); return T_True;}
"False" {debug("False"); return T_False;}
"None" {debug("None"); return T_None;}

":" {debug("Cln"); return T_Cln;}
"," {debug("Comma"); return T_Comma;}
">" {debug("GT"); return T_GT;}
"<" {debug("LT"); return T_LT;}
">=" {debug("EGT"); return T_EGT;}
"<=" {debug("ELT"); return T_ELT;}
"==" {debug("EQ"); return T_EQ;}
"!=" {debug("NEQ"); return T_NEQ;}

"(" {debug("OP"); return T_OP;}
")" {debug("CP"); return T_CP;}
"[" {debug("OB"); return T_OB;}
"]" {debug("CB"); return T_CB;}
"{" {debug("OCB"); return T_OCB;}
"}" {debug("CCB"); return T_CCB;}

"+" {debug("PL"); return T_PL;}
"-" {debug("MN"); return T_MN;}
"*" {debug("ML"); return T_ML;}
"**" {debug("EXP"); return T_EXP;}
"/" {debug("DV"); return T_DV;}
"%" {debug("Mod"); return T_Mod;}
"//" {debug("IDiv"); return T_IDiv;}
```

```
"=" {debug("Assign"); return T_EQL;}
"+=" {debug("PLAssign"); return T_PLAssign;}
"-=" {debug("MNAssign"); return T_MNAssign;}
"*=" {debug("MLAssign"); return T_MLAssign;}
"/=" {debug("DVAssign"); return T_DVAssign;}
"%=" {debug("ModAssign"); return T_ModAssign;}
"//=" {debug("IDivAssign"); return T_IDivAssign;}
"**=" {debug("ExpAssign"); return T_ExpAssign;}
```

Basic punctuations, arithmetic, logical and relational operators have been implemented in our lexer.

```
[0-9]+("."[0-9]+)?(E[+-]?[0-9]+)? {yylval.text = strdup(yytext); debug(yylval.text); return T_Number;}

[_a-zA-Z][_a-zA-Z0-9]* {yylval.text = strdup(yytext); debug(yylval.text); return T_ID;}
\"([^\"\n])*\" {yylval.text = strdup(yytext); debug(yylval.text); return T_String;}
\'([^\'\n])*\' {yylval.text = strdup(yytext); debug(yylval.text); return T_String;}
([a-z]|[0-9]|[A-Z]|" "|"\t")*"#"([a-z]|[0-9]|[A-Z]|" "|"\t")* {}
{whitespace} {}
"\n" {yycolumn=1; debug("NL"); return T_NL;}
<<EOF>> {debug("EOF"); return T_EndOfFile;}

%%
```

Above lex snippet explains regex rules for integer and floating-point numbers, identifier, strings, comments, and whitespaces.

F. Write small programs in the language you have developed.

INPUT:

```
import hWorld
x=10
y=5

#Comment1
print(x+y)

def F1():
        while(x==y):
        c=0
        z=10
        b=z

        if(z!=b):
                c=10+b
        else:
                c=10+z
```

OUTPUT:

```
------All Symbol Tables-------
Lexeme  Token   Declaration
hWorld  PackageName     1
10      Constant        2
x       Identifier      2
5       Constant        3
y       Identifier      3
F1      Func_Name       8
Def     Def             8
0       Constant        10
c       Identifier      10
10      Constant        11
z       Identifier      11
b       Identifier      12
Else    Else            15
If-Elif If-Elif         13
While   While           9
--------------------------------
```

G. Compile your programs and show that your lexical analyser is creating correct tokens and handling errors correctly.

INPUT1 (Variable Declaration – should start with alphabet):

```
import math
#comment1
x=10
2dfd=0
```

OUTPUT1

```
1 T_IMPT T_math T_NL
2 T_NL
3 T_X T_Assign T_10 T_NL
4 T_2 T_dfd
Syntax Error at Line 4, Column : 4
```

INPUT2 (Variable Length Limit - max 256):

```
import math
#comment1
sdggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggg=10
dfd=0
```

OUTPUT2

```
--------------------------------Token Sequence-------------------------
1 T_IMPT T_math T_NL
2 T_NL
3 Max length crossed:
sdggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggg, Length: 28%
```

INPUT3 (Number format):

```
import math
#comment1
df=10.1.2
2dfd=0
```

OUTPUT3

```
--------------------------------Token Sequence-------------------------
1 T_IMPT T_math T_NL
2 T_NL
3 T_df T_Assign T_10 .T_1
Syntax Error at Line 3, Column : 7
```