NED University Of Engineering & Technology

Department of Computer Science & IT

Complex Computing Problem
Theory of Programming Languages
TPL (CT-367)

Student Name	Roll Number
Muhammad Salman Sarwar	CT-21071
Abdul Rafay Chohan	CT-21087
Syed Muhammad Zaryab	CT-21090

Submitted to:
Sir Muhammad Kamran

Date: June 30, 2024

Contents

1.0 Assessment Rubrics

Course Code: CT-367			Course Title: Theory of Programming Language
Criteria and Scales			
Excellent (3)	**Good (2)**	**Average (1)**	**Poor (0)**
Criterion 1: Understanding the Problem: How well the problem statement is understood by the student			
Understands the problem clearly and clearly identifies the underlying issues and functionalities.	Adequately understands the problem and identifies the underlying issues and functionalities.	Inadequately defines the problem and identifies the underlying issues and functionalities.	Fails to define the problem adequately and does not identify the underlying issues and functionalities.
Criterion 2: Research: The amount of research that is used in solving the problem			
Contains all the information needed for solving the problem	Good research, leading to a successful solution	Mediocre research which may or may not lead to an adequate solution	No apparent research
Criterion 3: Code: How complete the code is along with the assumptions and selected functionalities			
Complete Code according to the according to the selected functionalities of the given case with clear assumptions	Incomplete Code according to the selected functionalities of the given case with clear assumptions	Incomplete Code according to the selected functionalities of the given case with unclear assumptions	Wrong code and naming conventions
Criterion 4: Report: How thorough and well organized is the solution			
All the necessary information clearly organized for easy use in solving the problem	Good information organized well that could lead to a good solution	Mediocre information which may or may not lead to a solution	No report provided

2.0 Introduction

In this assignment, we aim to build a lexical analyzer for the Python programming language. A lexical analyzer, also known as a lexer, is a program that converts a sequence of characters into a sequence of tokens. This process is a crucial first step in the compilation of source code, as it simplifies the syntax analysis phase by identifying and classifying the various elements of the code

3.0 Problem Statement

DEVISE appropriate functionalities (Tokenisation, built-in functions, keywords, identifiers) to **BUILD** a lexical analyser for any language. You may add your own assumptions to complete this case.

4.0 Problem Description

The lexical analyzer, commonly called the lexeme generator or scanner, is a crucial step in the compilation of source code. Its primary responsibility is to scan the input source code and parse it into lexemes or tokens, which are streams of meaningful components including keywords, identifiers, operators, and literals. Based on a set of predetermined rules, the lexeme generator uses pattern matching on the input to identify these tokens. **Building** a lexeme generator takes a lot of complex computer issues/challenges. You are required to build a lexical analyser for any of the programming languages with a predefined set of rules.

5.0 Description of Selected Language

We have selected the Python programming language for building our lexical analyzer. Python is a widely used general-purpose programming language that is known for its efficiency and control over system resources. Its simplicity and widespread use make it an ideal candidate for this project, as it allows us to easily define and recognize various tokens such as keywords, identifiers, operators, and literals.

6.0 Selected Functionality

- Keywords
- Identifiers
- Operators
- Literals

7.0 Rules / Regular Expression

```
DIGIT       [0-9]
LETTER      [a-zA-Z]
IDENTIFIER  {LETTER}({LETTER}|{DIGIT})*
NUMBER      {DIGIT}+
WHITESPACE  [ \t\n]+
STRING      \"(\\.|[^\\"])*\"
```

8.0 Source Code

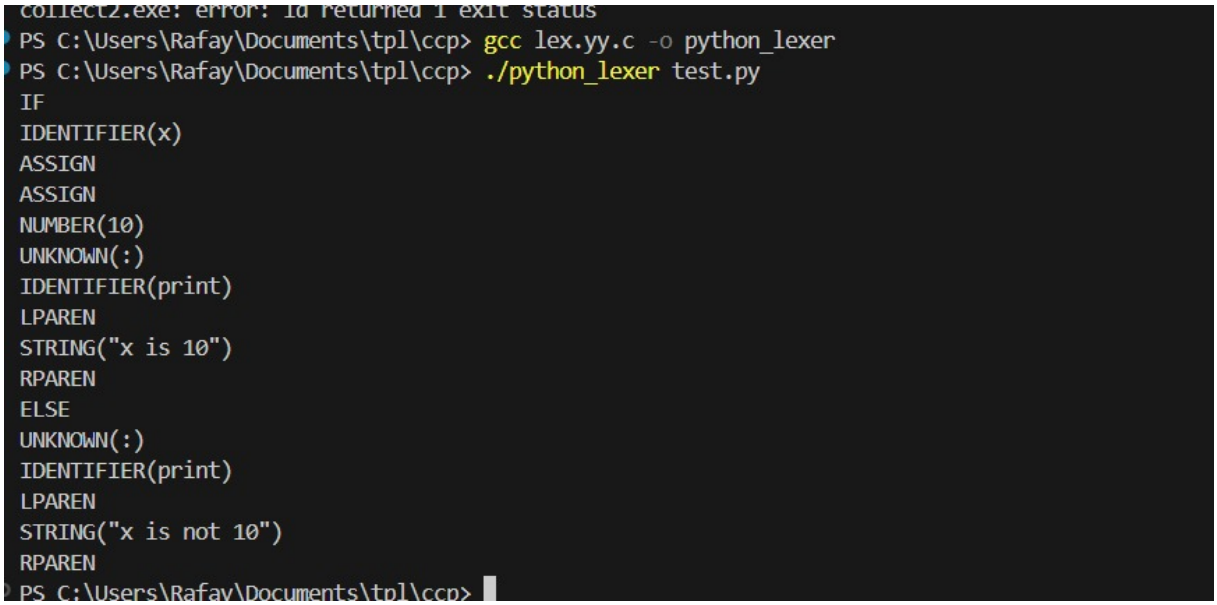
```
%{
#include <stdio.h>
%}

DIGIT       [0-9]
LETTER      [a-zA-Z]
IDENTIFIER  {LETTER}({LETTER}|{DIGIT})*
NUMBER      {DIGIT}+
WHITESPACE  [ \t\n]+
STRING      \"(\\.|[^\\"])*\"

%%

"if"        { printf("IF\n"); }
"else"      { printf("ELSE\n"); }
"while"     { printf("WHILE\n"); }
"return"    { printf("RETURN\n"); }
"int"       { printf("INT\n"); }

{IDENTIFIER} { printf("IDENTIFIER(%s)\n", yytext); }
{NUMBER}     { printf("NUMBER(%s)\n", yytext); }
{STRING}     { printf("STRING(%s)\n", yytext); }

"+"          { printf("PLUS\n"); }
"-"          { printf("MINUS\n"); }
"*"          { printf("TIMES\n"); }
"/"          { printf("DIVIDE\n"); }
"="          { printf("ASSIGN\n"); }

"("          { printf("LPAREN\n"); }
")"          { printf("RPAREN\n"); }
"{"          { printf("LBRACE\n"); }
"}"          { printf("RBRACE\n"); }
```

```
";"          { printf("SEMICOLON\n"); }

{WHITESPACE} { /* skip whitespace */ }

.            { printf("UNKNOWN(%s)\n", yytext); }

%%

int yywrap(void) {
    return 1;
}

int main(int argc, char **argv) {
    ++argv, --argc;  /* skip over program name */
    if (argc > 0)
        yyin = fopen(argv[0], "r");
    else
        yyin = stdin;

    yylex();
    return 0;
}
```

9.0 How to run the code

```
flex python_lexer.l
gcc lex.yy.c -o python_lexe
./python_lexer test.py
```

10.0 Output



11.0 Limitations

- The lexical analyzer does not handle comments.
- The analyzer is case-sensitive and does not recognize keywords in different cases.
- The analyzer does not handle multi-line strings or complex literals.

12.0 Conclusion

In conclusion, we successfully built a lexical analyzer for the Python programming language using Flex. This analyzer can tokenize keywords, identifiers, numbers, operators, and delimiters. Although it has some limitations, such as not handling comments or multi-line strings, it provides a foundational understanding of how lexical analysis works in a compiler. This project enhanced our understanding of lexical analysis and the initial phases of compilation, providing valuable insights into compiler design.