BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]

Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300101
Name	Adwait Purao
Class and Batch	TE Computer Engineering - Batch B
Date	19/01/2024
Lab #	1
Aim	Design a Lexical analyser for multiple programming languages and implement using lex tool.
Objective	The experiment focuses on crafting a Lexical Analyzer for multiple programming languages by leveraging the Lex tool. This entails outlining and executing lexical rules and patterns to identify and sort tokens, utilizing appropriate lexemes. The ultimate goal is to generate a program that precisely tokenizes input source code. Essential actions tied to this objective include designing, implementing, specifying, recognizing, categorizing, and generating – all pivotal tasks for the successful development of a Lexical Analyzer through the Lex tool.
Theory	**Introduction:** Imagine reading a book, but instead of absorbing complete sentences, you first focus on individual words and punctuation marks. That's essentially what lexical analysis does in the world of programming languages! It's the first step in understanding a program's code, like a meticulous chef chopping ingredients before cooking. **What is Lexical Analysis?** Lexical analysis, also known as tokenization, breaks down the source code (the recipe) into smaller, meaningful units called tokens (the chopped ingredients) [1]. These tokens are the building blocks of a language's syntax (the grammar), like keywords ("if", "for"), identifiers ("variable names"), operators ("+", "-"), punctuation ("()", ";"), and literals (numbers, strings) [1]. **Lexical Analyzer:** This crucial task is handled by a tool called a lexical analyzer (lexer or scanner). Think of it as a skilled chef, armed with a set of rules and tools, scanning the code character by character. The lexer uses powerful patterns called regular expressions to identify valid tokens, like a chef recognizing specific shapes and sizes of ingredients [1]. It then organizes these tokens into a neat stream, ready for the next step in the compilation process. **Lex**

Just like a chef has their favorite recipe book, programmers often use tools like Lex to build lexical analyzers. Lex takes a specification file (the recipe instructions) filled with regular expressions and actions for each token [2]. It then whips up C code for the lexer, like a magical recipe transformer! This code can then be compiled and integrated into a complete compiler or interpreter, turning the raw ingredients into a delicious program.

Designing Lexers for Different Languages:
Each programming language has its own unique vocabulary and syntax, like different cuisines with distinct flavors and cooking techniques. To design a lexer for a specific language, you need to:

Understand the language's grammar: Analyze the language's "recipe book" to identify the valid tokens and how they can be combined [1].
Craft the perfect recipes (regular expressions): Create precise patterns that match the shapes and sizes of valid tokens in the language.
Write the Lex specification file: Use a special format to tell Lex what ingredients to look for and how to prepare them (token actions) [2].
Generate and use the lexer: Run Lex to create the C code, then compile and use it as part of your compiler or interpreter.

The Benefits of Lexical Analysis:

Just like chopping ingredients makes cooking easier, lexical analysis offers several advantages:

Efficiency: It breaks down the input into smaller, more manageable chunks, allowing the parser (the chef's assistant) to focus on the bigger picture (the program's structure).
Flexibility: It allows for the use of keywords and reserved words, adding flavor and variety to programming languages.
Error Detection: It can catch typos, missing punctuation, and undefined variables, saving time during debugging [3].
Code Optimization: It can identify common patterns and replace them with more efficient code, making the program run faster and smoother [4].

While powerful, lexical analysis also has its limitations:

Complexity: Designing and implementing lexers can be challenging, requiring significant computational resources.
Limited Error Detection: It can't catch all errors, like logic or type issues, leaving some problems hidden in the dish [3].
Increased Code Size: Keywords and reserved words can add bulk to the code, making it less readable and understandable.
Reduced Flexibility: These keywords and reserved words can also restrict the programmer's freedom, limiting how they can use certain words and phrases [3].

	Types of Tokens in Python: **Keywords:** These are reserved words with predefined meanings that play essential roles in shaping the language's structure and control flow [5]. Examples include "if," "else," "for," "while," "def," "class," and "import." **Identifiers:** These represent the names assigned to various elements within the code, such as variables, functions, classes, and modules [6]. They must adhere to Python's naming conventions and start with a letter or underscore. **Literals**: These denote fixed values directly embedded within the code [5]. Common types include: **Numeric literals:** Representing numbers, they encompass integers and floating-point numbers [5]. **String literals:** Enclosing sequences of characters within quotes [5]. **Operators:** These tokens enact operations on values [5]. Python supports a diverse range of operators, including arithmetic, comparison, and logical operators [5]. **Punctuators:** These serve as structural elements within Python code, similar to punctuation in natural language [5]. They include parentheses, commas, colons, brackets, and braces [5].
Implementation / Code	

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

enum {
    START, IN_ID, IN_NUM, IN_COMMENT, IN_STRING, IN_INDENTATION
};

int state = START;
%}

%option noyywrap

%%
"if"|"else"|"while"|"for"|"return"|"elif" { printf("KEYWORD: (%s)\n",
yytext); state = START; }
"{"|"}"|"("|")"|"="|";"|"["|"]"|"+"|"-"|"*"|"/" { printf("SEPARATOR:
(%s)\n", yytext); state = START; }
```

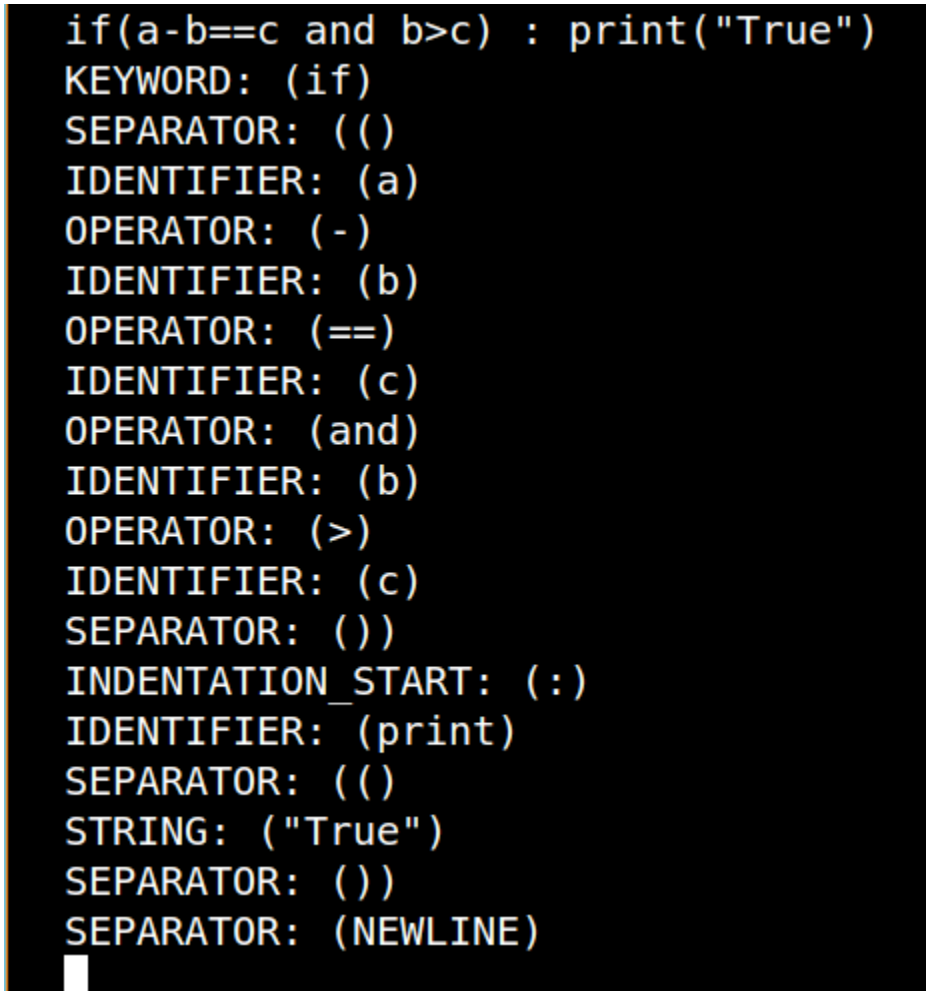
```
"=="|">="|"<="|">"|"<"|"<>"|"!="|"and"|"or" { printf("OPERATOR:
(%s)\n", yytext); state = START; }
[0-9]+(\.[0-9]+)?([eE][-+]?[0-9]+)? { printf("NUMBER: (%s)\n",
yytext); state = START; }
[a-zA-Z_][a-zA-Z0-9_]* {
    if (strcmp(yytext, "if") != 0 &&
        strcmp(yytext, "else") != 0 &&
        strcmp(yytext, "while") != 0 &&
        strcmp(yytext, "for") != 0 &&
        strcmp(yytext, "return") != 0 &&
        strcmp(yytext, "elif") != 0) {
        printf("IDENTIFIER: (%s)\n", yytext);
    } else {
        printf("KEYWORD: (%s)\n", yytext);
    }
    state = START;
}
\"([^"\\]|\\.)*\" { printf("STRING: (%s)\n", yytext); state = START; }
#.* { printf("COMMENT: (%s)\n", yytext); state = START; }
[ \t] ; // Skip whitespace
\n { printf("SEPARATOR: (NEWLINE)\n"); state = START; } // Treat
newline as a separator
: {
    printf("INDENTATION_START: (%s)\n", yytext);
    state = START;
}
. {
    // Handle transitions
    switch (state) {
        case START:
            if (*yytext == '#') {
                state = IN_COMMENT;
            } else if (*yytext == '"') {
                state = IN_STRING;
            } else if (isdigit(*yytext)) {
                state = IN_NUM;
```

```
            } else if (isalpha(*yytext) || *yytext == '_') {
                state = IN_ID;
            } else if (*yytext == '.') {
                printf("SEPARATOR: (%s)\n", yytext);
                state = START;
            } else if (*yytext == '+' || *yytext == '-' || *yytext ==
'*' || *yytext == '/') {
                printf("OPERATOR: (%s)\n", yytext);
                state = START;
            } else {
                printf("INVALID: (%s)\n", yytext);
            }
            break;
        case IN_ID:
            if (!(isalnum(*yytext) || *yytext == '_')) {
                unput(*yytext);
                state = START;
            }
            break;
        case IN_NUM:
            if (!isdigit(*yytext) && *yytext != '.') {
                unput(*yytext);
                state = START;
            }
            break;
        case IN_COMMENT:
            if (*yytext == '\n') {
                state = START;
            }
            break;
        case IN_STRING:
            if (*yytext == '"') {
                state = START;
            }
            break;
    }
```

```
}
%%
int main() {
    yylex();
    return 0;
}
```

Output

```
adwait@spit:~/Documents/SPCC$ flex Exp1.l
adwait@spit:~/Documents/SPCC$ gcc lex.yy.c -o lexer -lfl
adwait@spit:~/Documents/SPCC$ ./lexer
if(a==b or a+b>=c): return "Adwait studies in SPIT"
KEYWORD: (if)
SEPARATOR: (()
IDENTIFIER: (a)
OPERATOR: (==)
IDENTIFIER: (b)
OPERATOR: (or)
IDENTIFIER: (a)
OPERATOR: (+)
IDENTIFIER: (b)
OPERATOR: (>=)
IDENTIFIER: (c)
SEPARATOR: ())
INDENTATION_START: (:)
KEYWORD: (return)
STRING: ("Adwait studies in SPIT")
SEPARATOR: (NEWLINE)
if(a-b==c and b>c) : print("True")
KEYWORD: (if)
SEPARATOR: (()
IDENTIFIER: (a)
OPERATOR: (-)
IDENTIFIER: (b)
OPERATOR: (==)
IDENTIFIER: (c)
```



Conclusion

Through this experiment, I gained an understanding of Lex, a tool for creating lexical analyzers, and how it identifies tokens in a given language. Specifically, I explored the concept of tokens in the Python language. The process involved using the Flex package in Ubuntu to compile a Lex file and then utilizing the generated file to identify and categorize tokens in a Python-like input.

References

[1] Aho, A. V., Sethi, R., & Ullman, J. D. (2007). Compilers: principles, techniques, and tools (2nd ed.). Addison-Wesley.

[2] Lex Manual. Retrieved January 24, 2024, from https://man7.org/linux/man-pages/man1/lex.1p.html

[3] GeeksforGeeks. (2023, October 27). Introduction of Lexical Analysis. Retrieved January 24, 2024, from https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

[4] Introduction to Lex and Yacc. Retrieved January 24, 2024,

	[5] Van Rossum, G., & Drake, F. L. (2011). Python 3 reference manual. Scotts Valley, CA: CreateSpace. [6] Guido van Rossum, B. Warsaw, & N. Coghlan. (2023, January 23). Python documentation https://docs.python.org/.