LAB SESSION 3: TEXT ANALYSIS USING LEX

DATE:

AIM: To implement a program for word frequency analysis in a text file using the Lex Tool.

PROBLEM DEFINITION: Develop a Lex program to analyze a text file and count the frequency of each word, ignoring case and punctuation. The program should:

1. Recognize words consisting of letters (a-z, A-Z).
2. Convert all words to lowercase.
3. Ignore numbers, punctuation marks, and special symbols.
4. Print the frequency of each word found in the input file.

Input: A text file with multiple lines of text.
Output: List of words and their frequencies, sorted alphabetically.

THEORY: Lex in compiler design is a program used to generate scanners or lexical analyzers, also called tokenizers. These tokenizers identify the lexical pattern in the input program and convert the input text into the sequence of tokens. It is used with the YACC parser generator. Eric Schmidt and Mike Lesk initially developed the code for Lex, which was intended for Unix-based systems.

The function of Lex is as follows:

Lexical Analyzer Creation: The process begins by creating a program called lex.l using Lex's language. This program defines the rules and patterns for recognizing tokens in the source code

Lex Compiler Execution: The lex.1 program is then executed using the Lex compiler. This step generates a C program named lex.yy.c

C Compiler Execution: The C compiler is then used to compile the generated lex.yy.c program. The result is an object program referred to as a.out

Lexical Analysis: The a.out object program is essentially a lexical analyzer. When this program is run, it takes an input stream of source code and transforms it into a sequence of tokens based on the rules defined in the original lex.1 program

Working of Lex

The working of lex in compiler design as a lexical analysis takes place in multiple steps. Firstly we create a file that describes the generation of the lex analyzer. This file is written in Lex language and has a .l extension. The lex compiler converts this program into a C file called lex.yy.c. The C compiler then runs this C file, and it is compiled into a.out file. This a.out file is our working Lexical Analyzer which will produce the stream of tokens based on the input text.

Conflicts in Lex

Conflicts arise in Lex if a given string matches two or more different rules in Lex. Or when an input string has a common prefix with two or more rules. This causes uncertainty for the lexical analyzer, so it needs to be resolved.

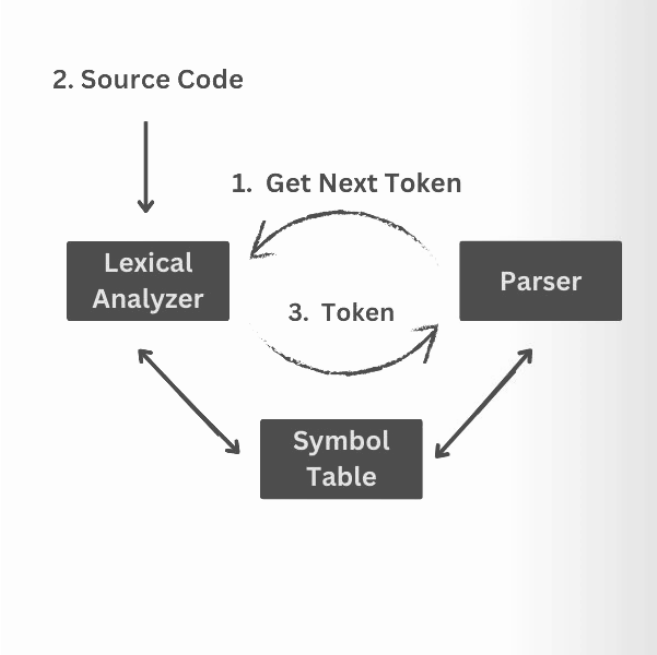
Conflict resolution

Conflicts in Lex can be resolved by following two rules-

1. The longer prefix should be preferred over a shorter prefix.
2. Pick the pattern listed first in the Lex program if the longest possible prefix corresponds to two or more patterns.

The Architecture of Lexical Analyzer

The task of the lexical analyzer is to read the input character in the source code and produce tokens one by one. The scanner produces tokens when it is requested by the parser. The lexical analyzer also avoids any whitespace and comments while creating tokens. If any error occurs, the analyzer correlates these errors with the source file and the line number.



Lookahead Operator

An additional character is read by Lex to differentiate between other patterns of a token. This is done by Lex by reading an extra character ahead of the valid lexeme. However, sometimes, we want a particular pattern to be matched to the input only when certain other characters follow it. Then, we may use a slash in a pattern to indicate the end of the part of the pattern that matches the lexeme.

For example:

The Lookahead operator is the addition operator read by Lex to distinguish different patterns for a token. A lexical analyzer reads one character ahead of a valid lexeme and then retracts to produce a token.

In some languages, keywords are not reserved. So the statements

IF (A, B) = 10 and IF(condition) THEN

Results in conflict about whether to produce IF as an array name or a keyword. To resolve this, the lex rule for the Keyword IF can be written as,

IF/\ (.* \) { letter }

Frequently Asked Questions

How to make a compiler with lex?

Creating a compiler with Lex involves a few key steps. First, we need to define regular expressions that describe the patterns of tokens in the source code. Then, we need to write a Lex specification file where you associate these regular expressions with corresponding actions.

What is the difference between C and lex?

C and Lex serve distinct roles. C is a versatile programming language used for various applications, offering a wide range of features. On the other hand, Lex is a specialized tool that generates code (often in C) for lexical analysis – the task of tokenizing source code.

Why do we need a lexical analyzer?

Lexical Analyzer is used to perform lexical analysis. The lexical analyzer also performs tasks like removing the white spaces and comments from the source code and is related to the source code's error messages.

What is a lexical error with an example?

A lexical error is a string of characters that doesn't follow any rules specified for the creation of tokens, and it cannot be categorized as a token. Examples of lexical errors include spelling mistakes, identifier or numerical constant length exceeding, etc.

PROGRAM:

Lex file:

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

#define MAXWORDS 1000

struct {
   char word[50];
```

```
   int count;
} table[MAXWORDS];

int nwords = 0;

void insert_word(char *w) {
   for (int i = 0; i < nwords; i++) {
      if (strcmp(table[i].word, w) == 0) {
         table[i].count++;
         return;
      }
   }
   strcpy(table[nwords].word, w);
   table[nwords].count = 1;
   nwords++;
}

%}

%%
[a-zA-Z]+   {
            char temp[50];
            int i;
            for(i=0; yytext[i]; i++)
               temp[i] = tolower(yytext[i]);
            temp[i] = '\0';
            insert_word(temp);
         }

[0-9]+            ;   /* ignore numbers */
[ \t\n\r.,;:!?\"\'()-]+   ;   /* ignore punctuation/whitespace */
.                 ;   /* ignore other characters */
%%

int cmp(const void *a, const void *b) {
   return strcmp(((struct {char word[50]; int count;}*)a)->word,
               ((struct {char word[50]; int count;}*)b)->word);
}

int main(int argc, char *argv[]) {
   if (argc < 2) {
      printf("Usage: %s <inputfile>\n", argv[0]);
```

```
        return 1;
    }

    FILE *fp = fopen(argv[1], "r");
    if (!fp) {
        printf("Could not open file %s\n", argv[1]);
        return 1;
    }

    yyin = fp;
    yylex();
    fclose(fp);

    qsort(table, nwords, sizeof(table[0]), cmp);

    printf("\nWord Frequency Analysis:\n");
    for (int i = 0; i < nwords; i++) {
        printf("%s : %d\n", table[i].word, table[i].count);
    }

    return 0;
}

int yywrap() {
    return 1;
}
```

input.txt:

```
Lex is powerful. Lex makes lexical analyzers.
Lex helps in compiler design, and lex is useful.
```

OUTPUT:

```
Word Frequency Analysis:
analyzers : 1
and : 1
compiler : 1
design : 1
helps : 1
```

in : 1
is : 2
lex : 4
lexical : 1
makes : 1
powerful : 1
useful : 1

CONCLUSION:

In this experiment, we implemented a Lex program to analyse word frequencies in a text file. The program successfully converted words to lowercase, ignored punctuation and numbers, and displayed results in alphabetical order. This exercise gave us hands-on experience with Lex, improved our understanding of lexical analysis and conflict resolution, and showed how regular expressions and tokenization can be applied not only in compiler design but also in practical text-processing tasks.