Aishwarya Shukla

Roll no. 324002

Gr no. 22010492

Batch: D1

**Assignment no. 04**

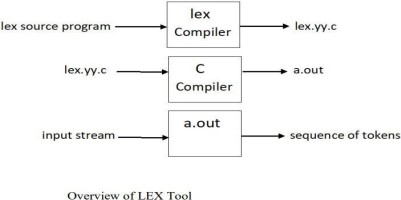
**Aim:** To write a program using LEX specifications to implement lexical analysis phase of compiler to generate count of no. of words, lines and characters of given input file.

# Objective:

1. For parts of speech for subset of ENGLISH language with SYMBOL TABLE
2. Write Lexical Analyzer with SYMBOL TABLE for subset of ‘C’ programming language

# Theory:

Lex is a tool for building lexical analysers. A lexer takes an arbitrary input stream and tokenizes it, i.e., divides it up into lexical tokens. This tokenized output can then be processed further, usually by yacc, or it can be the "end product."

Lex and C are tightly coupled. Lex itself doesn't produce an executable program; instead it translates the lex specification into a file containing a C routine called yylex( ).The program calls yylex( ) to run the lexer.

# Regular Expressions in LEX:

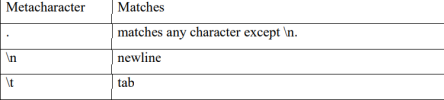
A regular expression is a pattern description using a "meta" language, a

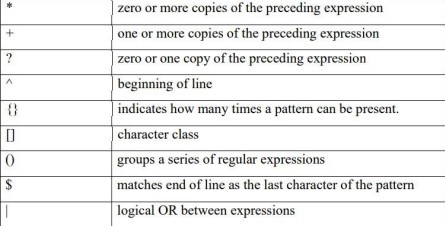
language that you use to describe particular patterns of interest. The characters used in this metalanguage are part of the standard ASCII character set used in UNIX and MS-DOS.

Examples:

1. digit: [0-9]

1. integer: [0-9]+
2. alphabet: [A-Za-z]
3. word: [A-Za-z]+





# Functions:

yytext- where text matched most recently is stored

yyleng- number of characters in text most recently matched ylval- associated value of current token

yywrap- wrapup, return 1 if done, 0 if not done

# Programming in LEX:

The simplest Lex program:

%%

.|\n ECHO;

%%

It acts very much like the UNIX cat command run with no arguments. Lex automatically generates the actual C program code needed to handle reading the input file and sometimes, as in this case, writing the output as well.

Recognizing words with LEX:

Given example shows a simple lex specification to recognize these verbs.

%{

/\* This is my first lex program \*/

%}

%%

[\t ]+

/\* ignore whitespace \*/ ; is I

am l

are { printf ("%s: is a verb\nn, yytext); }

[A-Za-z] { printf("%s: is not a verb\nm, yytext);

.|\n

%%

main()

{

yylex();

}

Here, the first section is definition section:

%{

/\* This is my first lex program \*/

%}

%%

It introduces any initial C program code we want copied into the final program. This is especially important if, for example, we have header files that must be

included for code later in the file to work. We surround the C code with the special delimiters "%{" and "%I." Lex copies the material between "%{" and "%I" directly to the generated C file.

The %% marks the end of this section.

The next section is the rules section. Each rule is made up of two parts: a pattern and an action, separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. The first rule in our example is the following:

[\t ]+ /\* ignore whitespace \*/ ;

The square brackets, "[]", indicate that any one of the characters within the brackets matches the pattern. For our example, we accept either "\t" (a tab character) or " " (a space). The "+" means that the pattern matches one or more consecutive copies of the sub-pattern that precedes the plus. Thus, this pattern describes whitespace. The second part of the rule, the action, is simply a semicolon, a do-nothing C statement. Its effect is to ignore the input.

The next set of rules uses the " I " (vertical bar) action. This is a special action that means to use the same action as the next pattern, so all of the verbs use the action specified for the last one.

Our first set of patterns is: is I

am l

are { printf ("%s: is a verb\nn, yytext); }

Our patterns match any of the verbs in the list. Once we recognize a verb, we execute the action, a C printf statement. The array yytext contains the text that matched the pattern. This action will print the recognized verb followed by the string “: is a verb\n”.

The last two rules are:

[a-zA-Z]+ { printf("%s: is not a verb\n, yytext);)

.|\n

The pattern "[a-zA-Z]+" is a common one: it indicates any alphabetic string with at least one character. The "-" character has a special meaning when used inside square brackets: it denotes a range of characters beginning with the character to the left of the "-" and ending with the character to its right. Our action when we see one of these patterns is to print the matched token and the string “: is not a verb\n”. The final section is the user subroutines section, which can consist of any legal C code. Lex copies it to the C file after the end of the lex generated code.

We have included a main() program.

%%

main()

{

yylex();

}

The lexer produced by lex is a C routine called yylex(), so we call it.\* Unless the actions contain explicit return statements, yylex() won't return until it has processed the entire input.

A lex specification consists of three sections: a definition section, a rules section, and a user subroutines section. The first section, the definition section, handles options lex will be using in the lexer, and generally sets up the execution environment in which the lexer operates.

Declaration for the word counting program:

%{

#include #include int char\_count=0,word\_count=0,line\_count=0; %}

The section bracketed by "%{" and "%}" is C code which is copied verbatim into the lexer. It is placed early on in the output code so that the data definitions contained here can be referenced by code within the rules section. In our example, the code block here declares three variables used within the program to track the number of characters, words, and lines encountered.

Here is our sample word count's rules section:

%%

[\n] {line\_count=line\_count+1;}

[^\n \t]+ {word\_count++;char\_count=char\_count+yyleng;}

The beginning of the rules section is marked by a "%%". In a pattern, lex replaces the name inside the braces {} with substitution, the actual regular expression in the definition section. Our example increments the number of lines, words and characters after the lexer has recognized a complete word. Our sample also uses the lex internal variable yyleng which contains the length of the string our lexer recognized. When our lexer recognizes a newline, it will increment both the character count and the line count. Similarly, if it recognizes any other character it increments the character count. For this lexer, the only "other characters" it could recognize would be a full stop.

The third and final section of the lex specification is the user subroutines section:

%%

int main()

{

yyin=fopen("file.txt","r"); yylex();

printf("\nFile Contents-"); printf("\nLine count: %d",line\_count);

printf("\nWord count: %d",word\_count); printf("\nCharacter count: %d \n\n",char\_count);

return 0;

}

int yywrap()

{

return 1;

}

Once again, it is separated from the previous section by "%%". The user subroutines section can contain any valid C code. It first calls the lexer's entry point yyIex() and then calls printf() to print the results of this run.

The first argument the program is called with is the input file “file.txt” to open for processing. A lex lexer reads its input from the standard I/O file yyin, so you need only change yyin as needed. The default value of yyin is stdin, since the default input source is standard input.

When yylex() reaches the end of its input file, it calls yywrap(), which returns a value of 0 or 1. If the value is 1, the program is done and there is no more input. If the value is 0, on the other hand, the lexer assumes that wrap() has opened another file for it to read, and continues to read from yyin. The default yywrap() always returns 1. By providing our own version of yywrap(), we can have our program read all of the files named on the command line, one at a time

# Symbol Tables

Our second example isn't really very different. We list more words than we did before, and in principle we could extend this example to as many words as we want. It would be more convenient, though, if we could build a table of words as the lexer is running, so we can add new words without modifying and recompiling the lex program. In our next example, we do just that-allow for the dynamic declaration of parts of speech as the lexer is running, reading the words to declare from the input file. Declaration lines start with the name of a part of speech followed by the words to declare.

These lines, for example, declare four nouns and three verbs: noun dog cat horse cow

verb chew eat lick

The table of words is a simple symbol table, a common structure in lex and yacc applications. A C compiler, for example, stores the variable and structure names, labels, enumeration tags, and all other names used in the program in its symbol table. Each name is stored along with information describing the name.

In a C compiler the information is the type of symbol, declaration scope, variable type, etc. In our current example, the information is the part of speech.

Adding a symbol table changes the lexer quite substantially. Rather than putting separate patterns in the lexer for each word to match, we have a single pattern that matches any word and we consult the symbol table to decide which part of speech we've found. The names of parts of speech

# Code (a) ;

%{

/\*

\* We expnd upon the first example by adding recognition of scane other \* parts of speech.

\*/ #include

%}

%%

[\t ]+ /\* ignore whitespace \*/ ; is |

am |

are | were |

was |

be | being | been | do | does | did | will | would | should | can | could | has | have | had |

go {printf("%s: is a verb

\n",yytext);} very | simply |

gently |

queitly | calmly |

angrily {printf("%s: is an adverb\n",yytext);} to | from |

behind | above |

below |

between {printf("%s: is a preposition\n",yytext);} if |

then |

and | but |

or {printf("%s: is a conjunction\n",yytext);} their |

my | your | his | her |

its {printf("%s: is an adjevtive\n",yytext);}

I | you | he | she | we |

they {printf("%s: is a pronoun\n",yytext);}

[a-zA-Z]+ {printf("%s: don't recognize, might be a noun\n",yytext);} .|\n

{ECHO; }

%%

int yywrap()

{

return 0;

}

void yyerror()

{

printf("Invalid varible declaration\n");

}

int main()

{

yylex();

yywrap();

}

# Output:



**Code (b):**

%{

#include

%}

%% [\t]+ ; ‘

int |

char | float |

bool {printf("%s: is a datatype\n",yytext);} if |

else | while | switch | for |

struct {printf("%s: is a keyword\n",yytext);} ";" {printf("%s: is a semicolon\n",yytext);} [a-zA-Z]+[a-zA-Z0-9]\* {printf("%s: is an

identifier\n",yytext);} "%s" | "%d" |

"%c" |

"%f" {printf("%s: is a format specifier\n",yytext);} "+" |

"-" |

"\*" |

"/" |

"%" {printf("%s: is a arithmetic operator\n",yytext);} "&&" | "||" {printf("%s: is a logical operator\n",yytext);} "<" | "<=" |

">" |

">=" |

"==" {printf("%s: is a relational operator\n",yytext);} "[" |

"{" |

"]" |

"}" |

"(" |

")" {printf("%s: is a punctuation symbol\n",yytext);} \n ;

%%

int yywrap()

{

return 0;

}

void yyerror()

{

printf("Invalid varible declaration\n");

}

int main() { yylex();

yywrap();

}

KUP; /\* not found \*/

# Output

