DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Port City International University

Course Code : CSE 412

Course Title : Compiler Sessional

Report 01

Submitted To:

Md. Muhtadir Rahman

Department of Computer Science and Engineering

Port City International University

Submitted By:

Rafi Uddin Chotan

Id: CSE 01906805

Trimester: Summer 2022

Department of Computer Science and EngineeringPort City International University

Date of Submission: 3 July,2022

Experiment No-01 :Environment Setup, Introduction to Lexical Analysis.

Introduction

Before 1975 writing a compiler was a very time-consuming process. Then Lesk [1975] and Johnson [1975] published papers on lex and yacc. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in Aho [2006]. Flex and bison, clones for lex and yacc, can be obtained for free from GNU and Cygwin. Cygwin is a 32-bit Windows ports of the GNU software. In fact Cygwin is a port of the Unix operating system to Windows and comes with compilers gcc and g++. To install simply download and run the setup executable. Under devel install bison, flex, gcc-g++, gdb, and make. Under editors install vim. Lately I've been using flex and bison under the Cygwin environment.

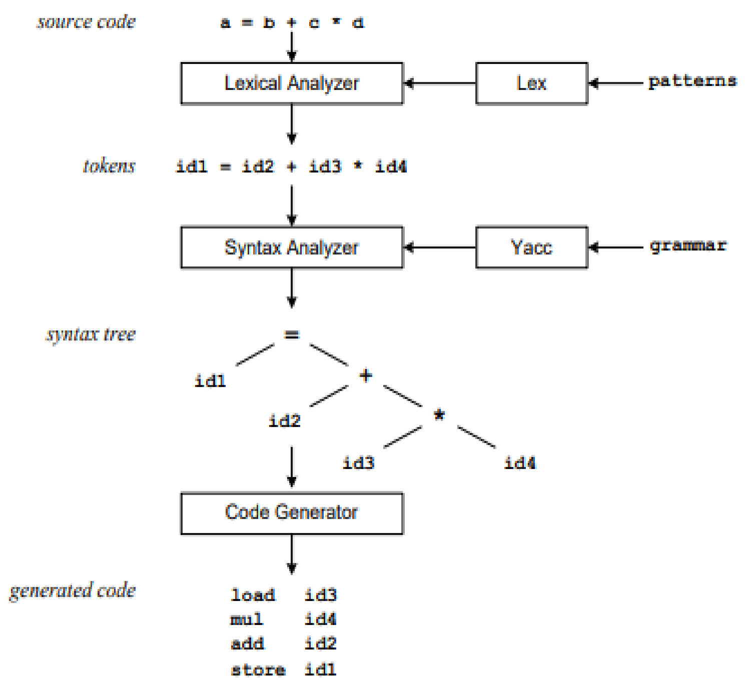


Figure 1: Compilation Sequence

The patterns in the above diagram is a file you create with a text editor. Lex will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index. The grammar in the above diagram is a text file you create with a text edtior. Yacc will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depthfirst walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language.

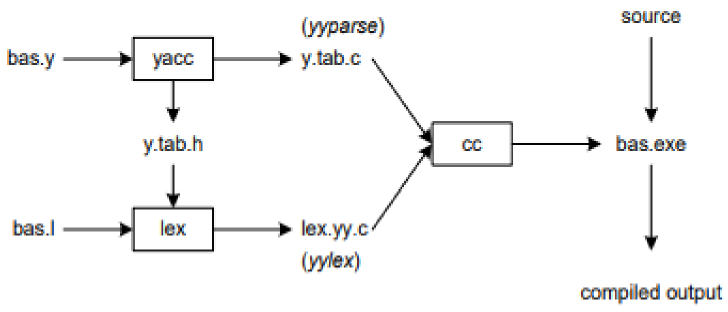


Figure 2: Building a Compiler with Lex/Yacc

Figure 2 illustrates the file naming conventions used by lex and yacc. We'll assume our goal is to write a BASIC compiler. First, we need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y). Commands to create our compiler, bas.exe, are listed below:

```
yacc –d bas.y                    # create y.tab.h, y.tab.c
lex bas.l                        # create lex.yy.c
cc lex.yy.c y.tab.c –obas.exe    # compile/link
```

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser), that includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The –d option causes yacc to generate definitions for tokens and place them in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c. Finally, the lexer and parser are compiled and linked together to create executable bas.exe. From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token

Lex

Theory

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value. The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter(letter|digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*" operator
- alternation, expressed by the "|" operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

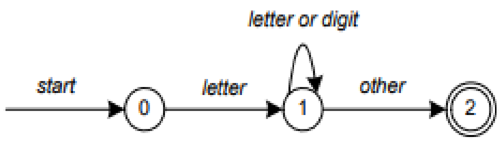
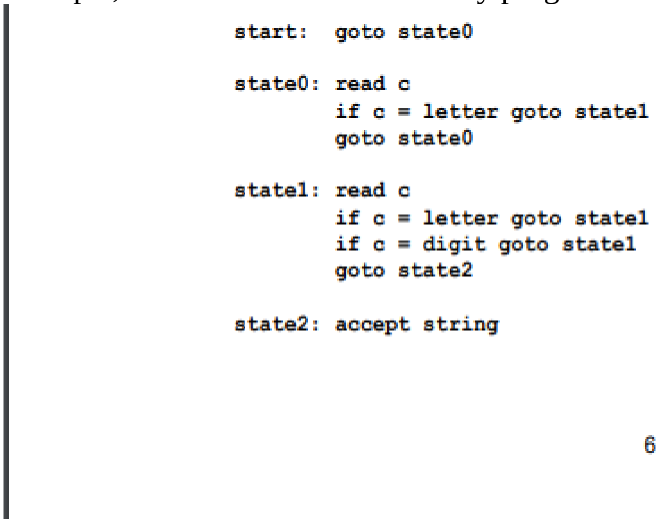


Figure 3: Finite State Automaton

In Figure 3 state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. Any FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

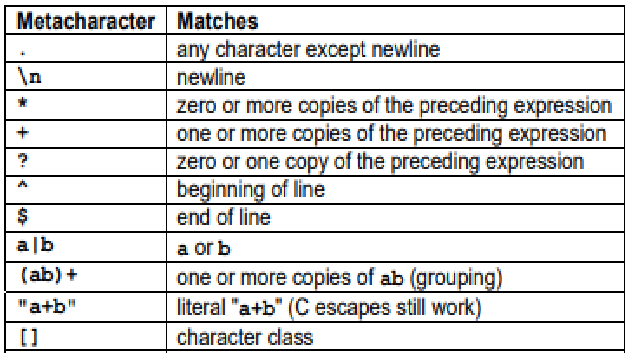


```
start:  goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

6

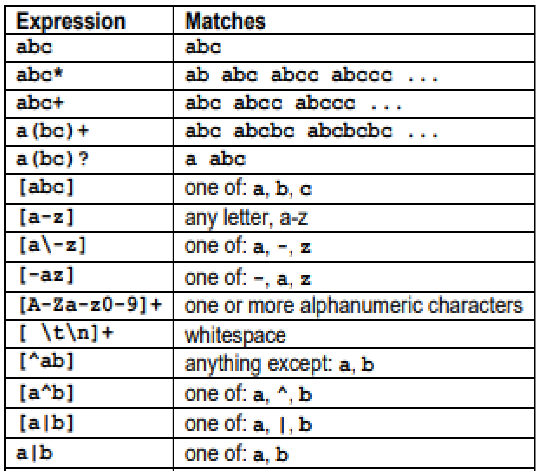
This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next input character and current state the next state is easily determined by indexing into a computer-generated state table. Now we can easily understand some of lex's limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "(" we push it on the stack. When a ")" is encountered we match it with the top of the stack and pop the stack. However lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks

Practice



Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line
`$`	end of line
`a|b`	a or b
`(ab)+`	one or more copies of ab (grouping)
`"a+b"`	literal "a+b" (C escapes still work)
`[]`	character class

Table 1: Pattern Matching Primitives



Expression	Matches
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`abc+`	`abc abcc abccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`
`[abc]`	one of: a, b, c
`[a-z]`	any letter, a-z
`[a\-z]`	one of: a, -, z
`[-az]`	one of: -, a, z
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: a, b
`[a^b]`	one of: a, ^, b
`[a|b]`	one of: a, |, b
`a|b`	one of: a, b

Table 2: Pattern Matching Examples

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class normal operators lose their meaning. 7 Two operators allowed in a character class are the hyphen (“-”) and circumflex (“^”). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to Lex is divided into three sections with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

%%

Input is copied to output one character at a time. The first %% is always required, as there must always be a rules section. However if we don’t specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are stdin and stdout, respectively. Here is the same example with defaults explicitly coded:

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

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

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, “.” and “\n”, with an ECHO action associated for each pattern. Several macros and variables are predefined by lex. ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable yytext is a pointer to the matched string (NULL-terminated) and yyleng is the length of the matched string. Variable yyout is the output file and defaults to stdout. Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a main function. In this case we simply call yylex that is the main entry-point for lex. Some implementations of lex include copies of main and yywrap in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Table 3: Lex Predefined Variables

Here is a program that does nothing at all. All input is matched but no action is associated with any pattern so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin** and defaults to **stdin**.

```
%{
    int yylineno;
%}
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with “%{“ and “%}” markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit     [0-9]
letter    [A-Za-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*         count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

Yacc

Theory

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

```
E -> E + E
E -> E * E
E -> id
```

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

```
E -> E * E         (r2)
  -> E * z         (r3)
  -> E + E * z     (r1)
  -> E + y * z     (r3)
  -> x + y * z     (r3)
```

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single nonterminal. This is known as bottom-up or shift-reduce parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:

```
 1    . x + y * z    shift
 2    x . + y * z    reduce(r3)
 3    E . + y * z    shift
 4    E + . y * z    shift
 5    E + y . * z    reduce(r3)
 6    E + E . * z    shift
 7    E + E * . z    shift
 8    E + E * z .    reduce(r3)
 9    E + E * E .    reduce(r2)         emit multiply
10    E + E .        reduce(r1)         emit add
11    E .            accept
```

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a handle and we are reducing the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack to change x to E. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition. Consider the shift at step 6. Instead of shifting we could have reduced and apply rule r1. This would result in addition having a higher precedence than multiplication. This is known as a shiftreduce conflict. Our grammar is ambiguous because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule

E -> E + E

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply yacc with directives that indicate which operator has precedence. The latter method is simpler and will be demonstrated in the practice section. The following grammar has a reduce-reduce conflict. With an id on the stack we may reduce to T, or E.

E -> T
E -> id
T -> id

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

Experiment No: 02

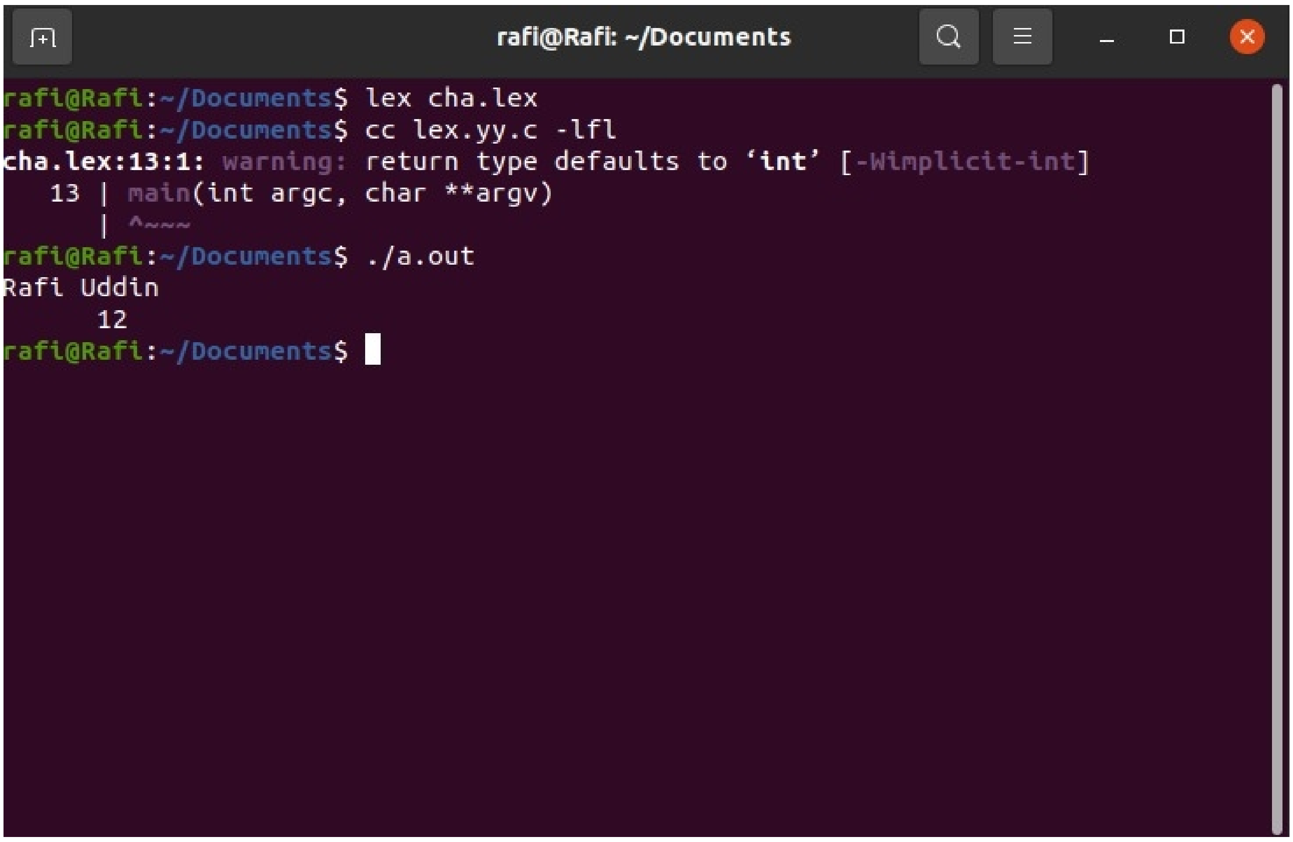
Experiment Name: Write a Lex Program to find the character counts in a string.

Problem Statement: The words can consist of lowercase characters, uppercase characters and digits. Below is the implementation to count the number of characters.

Source Code:

```
%{
int chars = 0;
%}

%%

[a-zA-Z]+  { chars += strlen(yytext); }
\n         { chars++; }
.          { chars++; }

%%

main(int argc, char **argv)
{
  yylex();
  printf("%8d\n", chars);
}
```

Output:



```
rafi@Rafi: ~/Documents
rafi@Rafi:~/Documents$ lex cha.lex
rafi@Rafi:~/Documents$ cc lex.yy.c -lfl
cha.lex:13:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   13 | main(int argc, char **argv)
      | ^~~~
rafi@Rafi:~/Documents$ ./a.out
Rafi Uddin
       12
rafi@Rafi:~/Documents$ 
```

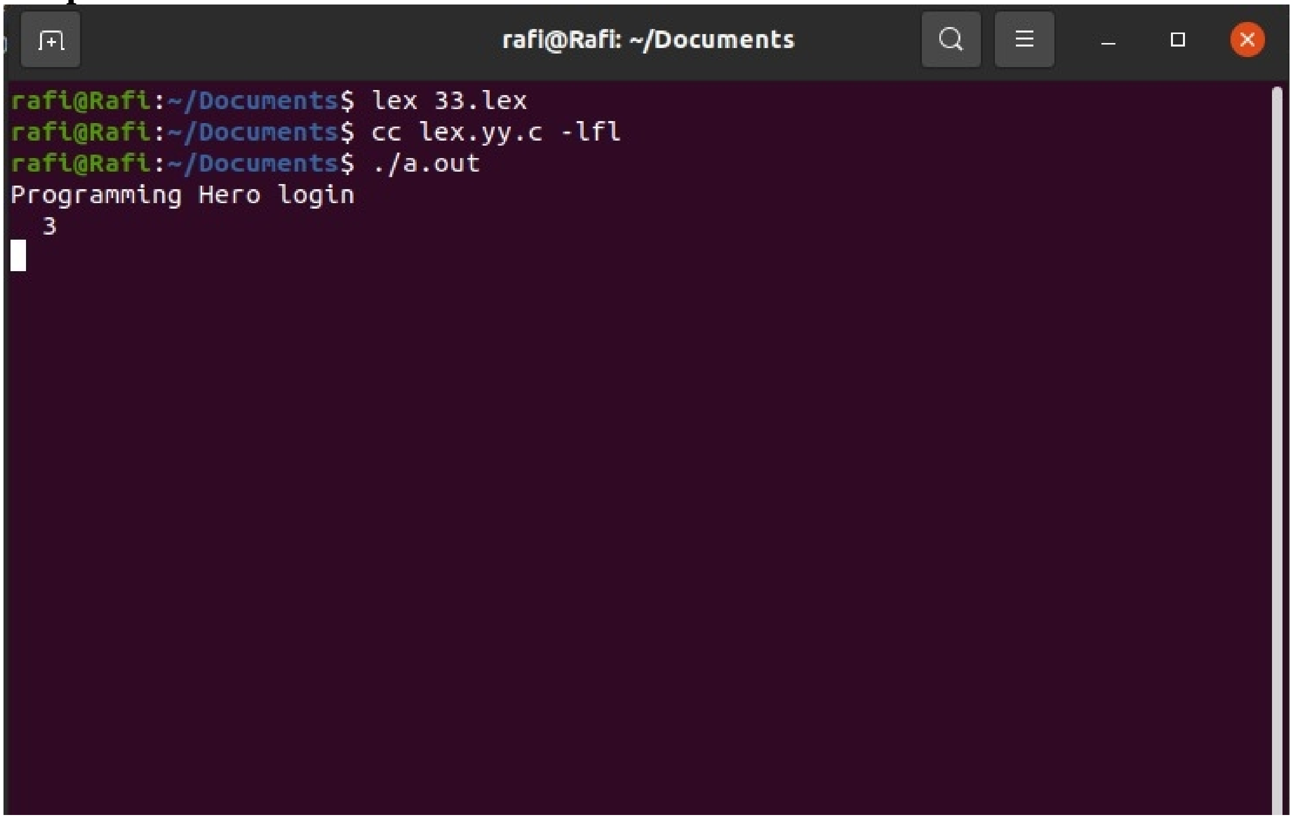
Conclusion: We have looked at several sources to articulate an introduction into the topic of LEX and its role in compiler design, Operating Systems etc. We have provided a programmatic basis for how lex maybe used, how it maybe specifies, how its variables maybe used and finally how it all comes together in code. Further scope for research includes a more thorough computer science or computing based approach to lex and not just a programmatic one.

Experiment No: 03
Experiment Name: Write a Lex Program to find the Word counts in the string.
Problem Statement: The words can consist of lowercase characters, uppercase characters and digits. Below is the implementation to count the number of words.
Source Code:

```
%{
int words = 0;
%}

%%
[a-zA-Z]+ { words++;}

%%

main(int argc, char **argv)
{
  yylex();
  printf("%8d\n", words);
```

}

Output:



```
rafi@Rafi:~/Documents$ lex 33.lex
rafi@Rafi:~/Documents$ cc lex.yy.c -lfl
rafi@Rafi:~/Documents$ ./a.out
Programming Hero login
  3
```

Experiment No: 04

Experiment Name: Write a Lex Program to find a string that starts with “a” & also ends with the same

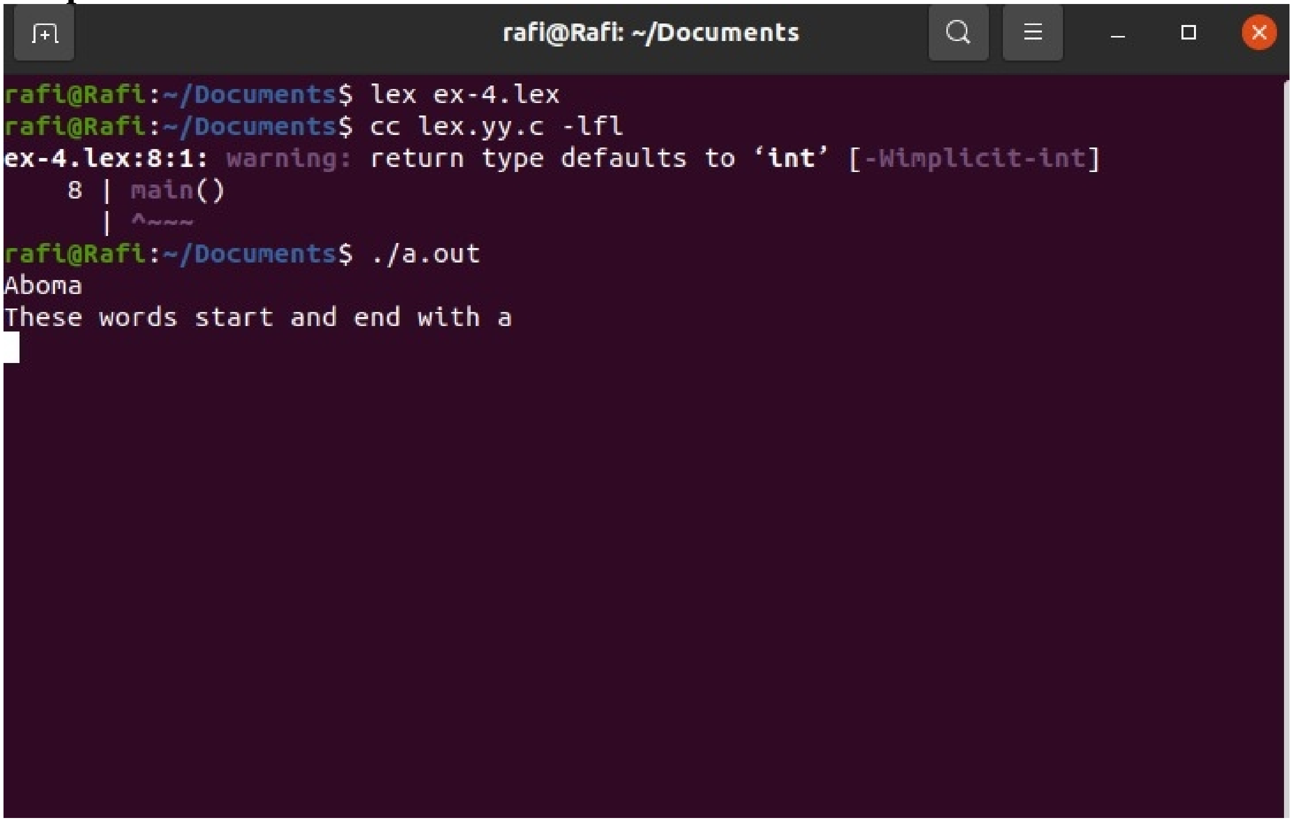
Problem Statement: The words can consist of lowercase characters, uppercase characters and digits. Below is the implementation to find whether a string starts with "a" & also ends with the same.

Source Code:

```
%{
#include<stdio.h>
%}
%%
((a|A)*[a-z|A-Z]*[0-9]*(a|A)) {printf("These words start and end with a");}
.* {printf("These word doesn't not start and end with a");}
%%
main()
{
yylex();
return 0;
}
int yywrap()
{
```

}

Output:



rafi@Rafi: ~/Documents

```
rafi@Rafi:~/Documents$ lex ex-4.lex
rafi@Rafi:~/Documents$ cc lex.yy.c -lfl
ex-4.lex:8:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
    8 | main()
      | ^~~~
rafi@Rafi:~/Documents$ ./a.out
Aboma
These words start and end with a
```

Experiment No: 06

Experiment Name: Write a Lex Program to check the given word a string or a Arithmetic Operator

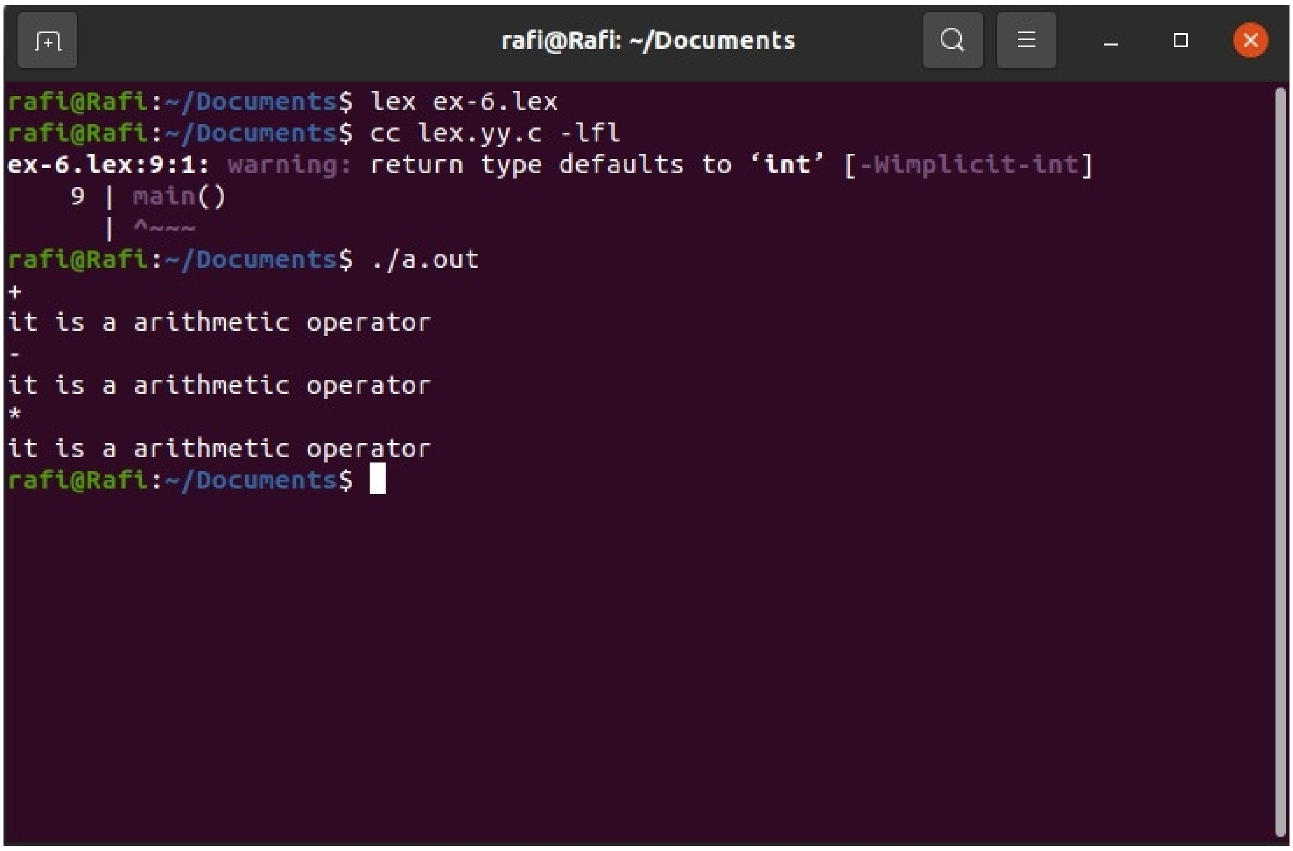
Problem Statement: The words can consist of lowercase characters, uppercase characters and digits.These operators are + (addition), - (subtraction), * (multiplication), / (division), and % (modulo). Below is the implementation to check the given word a string or an Arithmetic Operator.

Source Code:

```
%{
#include<stdio.h>
%}
%%
"+"|"-"|"*"|"/" {printf("it is a arithmetic operator");}
.* {printf("it is a string");}

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

```
{
}
```

Output:



```
rafi@Rafi:~/Documents$ lex ex-6.lex
rafi@Rafi:~/Documents$ cc lex.yy.c -lfl
ex-6.lex:9:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
    9 | main()
      | ^~~~
rafi@Rafi:~/Documents$ ./a.out
+
it is a arithmetic operator
-
it is a arithmetic operator
*
it is a arithmetic operator
rafi@Rafi:~/Documents$
```

Experiment No: 07

Experiment Name: Write a Lex program to count vowels & consonants

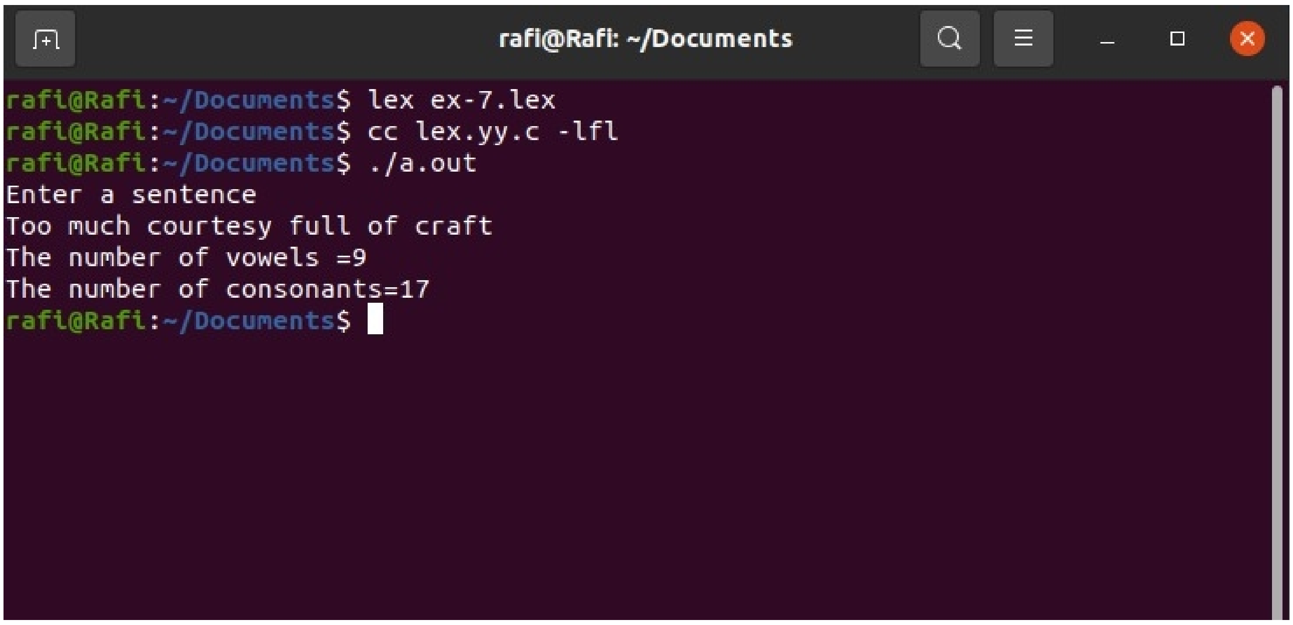
Problem Statement: The words can consist of lowercase characters, uppercase characters and digits. The letters of the alphabet that usually represent the consonant sounds are: b, c, d, f, g, h, j, k, l, m, n, p, q, r, s, t, v, w, x, y, z. Remaining characters are vowels. Below is the implementation to Count Vowels & Consonants.

Source Code:

```
%{
#include<stdio.h>
int v_count=0, c_count=0;
%}

%%
[ \t\n] {;}
[aAeEiIoOuU] { v_count++; }
[a-zA-Z] {c_count++;  }
%%

int main(){
```

```
        printf("Enter a sentence \n");
        yylex();
        printf("The number of vowels =%d \n",v_count);
        printf("The number of consonants=%d \n", c_count);
}
```

Output:



```
rafi@Rafi:~/Documents$ lex ex-7.lex
rafi@Rafi:~/Documents$ cc lex.yy.c -lfl
rafi@Rafi:~/Documents$ ./a.out
Enter a sentence
Too much courtesy full of craft
The number of vowels =9
The number of consonants=17
rafi@Rafi:~/Documents$
```

Experiment No: 08

Experiment Name: Write a Lex program to Identify Keywords.

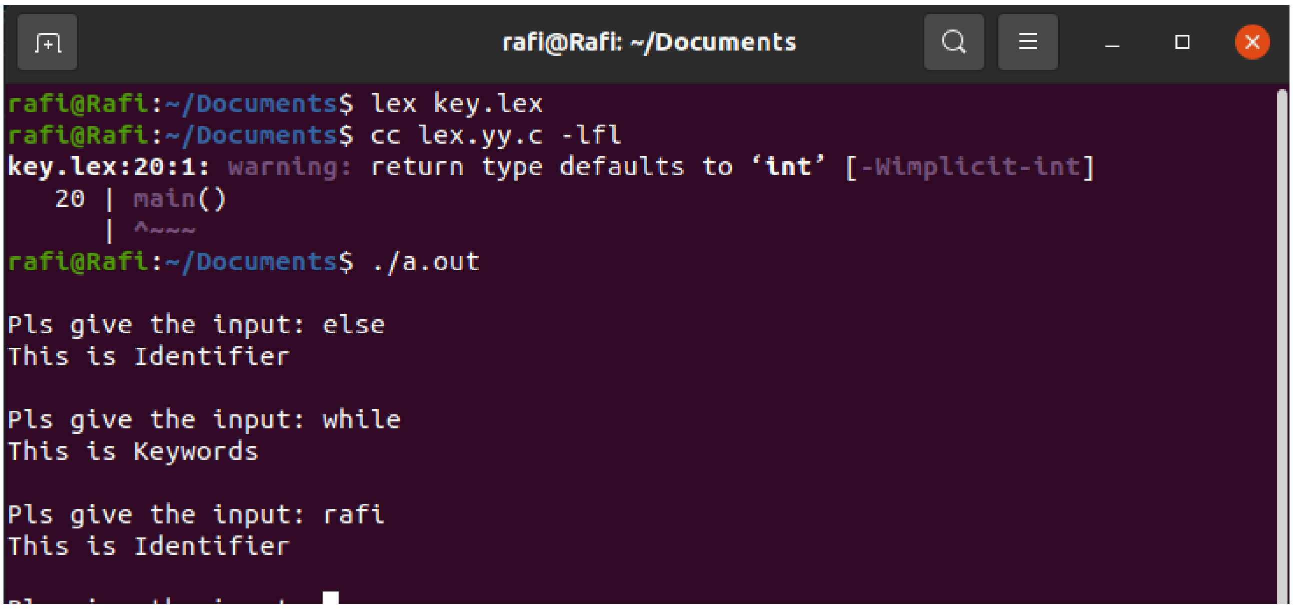
Problem Statement: The words can consist of lowercase characters, uppercase characters and digits. Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example: int money; Here, int is a keyword that indicates money is a variable of type int (integer).Below is the implementation to Identify Keywords.

Source Code:

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

%%
[\n] {printf("\n\nPls give the input: ");}

auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern|return|union|
continue|for|signed|void|do|if|static|while|default|goto|sizeof|volatile|const|float|short {printf("This is
Keywords");}

[a-z A-Z _][a-z A_Z 0-9 _]* {printf("This is Identifier");}
```

```
[0-9]* {printf("This is Number");}

[+-/*%=] {printf("This is Operator");}

.* {printf("Invalid");}

%%

main()
{
 printf("\nPls give the input: ");
 yylex();
 return 0;
}
```

Output:



rafi@Rafi: ~/Documents

```
rafi@Rafi:~/Documents$ lex key.lex
rafi@Rafi:~/Documents$ cc lex.yy.c -lfl
key.lex:20:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   20 | main()
      | ^~~~
rafi@Rafi:~/Documents$ ./a.out

Pls give the input: else
This is Identifier

Pls give the input: while
This is Keywords

Pls give the input: rafi
This is Identifier
```

Experiment No: 09
Experiment Name: Write a Lex Program to to count total number of tokens.
Problem Statement: A token is a group of characters forming a basic atomic chunk of syntax i.e. token is a class of lexemes that matches a pattern. Eg – Keywords, identifier, operator, separator.. Below is the implementation to to count total number of tokens

Source Code:

```
%{
  enum yytokentype {
    NUMBER = 258,
    ADD = 259,
    SUB = 260,
```

```
     MUL = 261,
     DIV = 262,
     ABS = 263,
     EOL = 264
   };

   int yylval;
%}

%%
"+"    { return ADD; }
"-"    { return SUB; }
"*"    { return MUL; }
"/"    { return DIV; }
"|"    { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n     { return EOL; }
[ \t]  { /* ignore whitespace */ }
.      { printf("Mystery character %c\n", *yytext); }
%%
main(int argc, char **argv)
{
  int tok;

  while(tok = yylex()) {
    printf("%d", tok);
    if(tok == NUMBER) printf(" = %d\n", yylval);
    else printf("\n");
  }
}
```

Output: