LAB SESSION 1: INTRODUCTION TO LEX

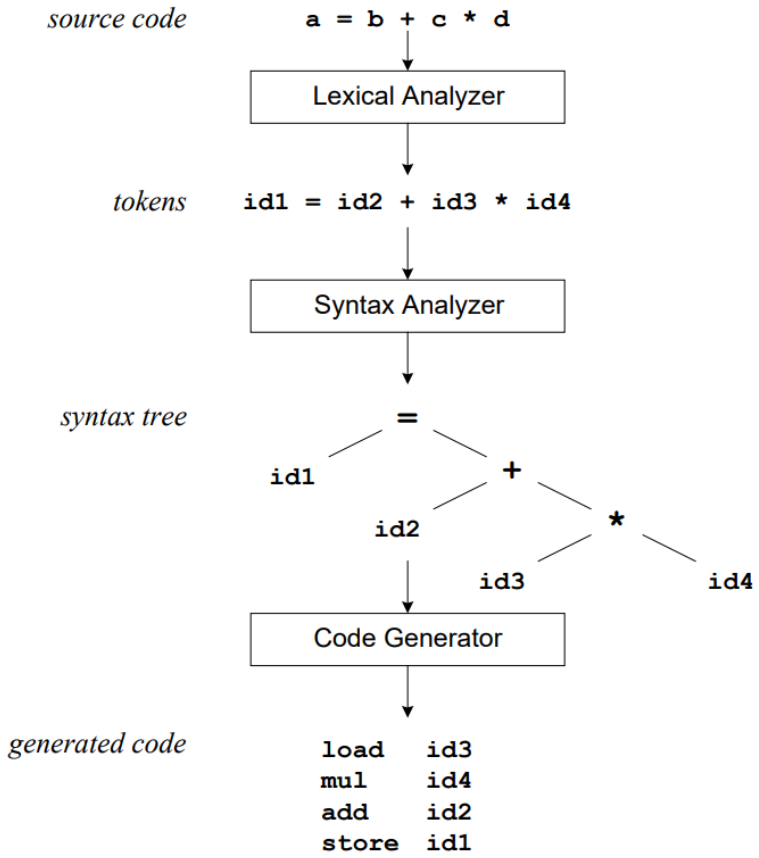
DATE:

AIM: To implement a basic program using the Lex Tool.

PROBLEM DEFINITION: Implement the following programs:

1. A LEX program to find if the input is integer, real number or word
2. A LEX program to convert decimal numbers to hexadecimal numbers.
3. A Lex program to include line numbers in a given source program
4. A LEX program to compute the average of a given set of numbers.

THEORY: The unix utility lex parses a file of characters. It uses regular expression matching; typically it is used to 'tokenize' the contents of the file. In that context, it is often used together with the yacc utility. However, there are many other applications possible.



Lex generates C code for a lexical analyzer, or scanner. It uses patterns that match strings in the input and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. This is illustrated in Figure.

As lex 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 the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex that allow it to scan and match strings in the input. Each pattern in lex has an associated action. Typically an action returns a token, representing the matched string, for subsequent use by the parser.

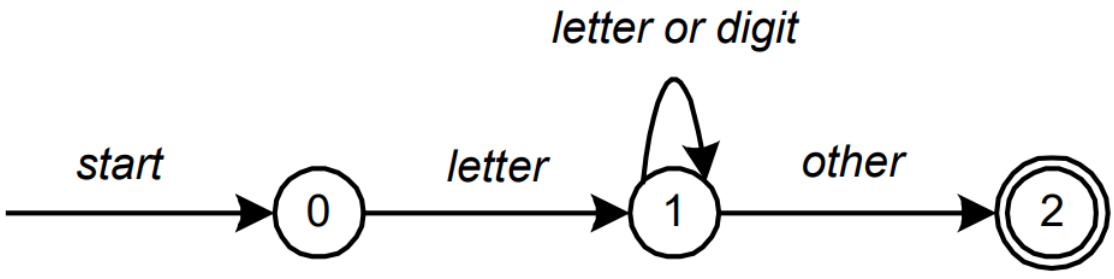
To begin with, however, we will simply print the matched string rather than return a token value. We may scan for identifiers using the regular expression.

```
letter(letter|digit)*
```

This pattern matches a string of characters that begins with a single letter, and is 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 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.



In Figure, 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 state 2, the accepting state. Any FSA may be expressed as a computer program.

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. Lex, however, 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.

Metacharacter	*Matches*
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of preceding expression
`+`	one or more copies of preceding expression
`?`	zero or one copy of preceeding 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

Metacharacters in Lex

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

Sample Regular Expressions in Lex

Regular expressions in lex are composed of metacharacters. Pattern matching examples are shown in Table. Within a character class, normal operators lose their meaning. 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.

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

Pre defined variables in Lex

How to execute

1. Type lex lexfile.l
2. Type gcc lex.yy.c.
3. Type ./a.exe

PROGRAM 1:

```
%{
#include <stdio.h>
int yylex(void);
int yywrap(void);
%}

%%
[0-9]+              { printf("%s is an INTEGER\n", yytext); }
[0-9]+"."[0-9]+     { printf("%s is a REAL NUMBER\n", yytext); }
[a-zA-Z]+           { printf("%s is a WORD\n", yytext); }
.|\n                { /* ignore other characters */ }
%%

int main() {
```

```
   yylex();
   return 0;
}

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

OUTPUT 1:

```
PS C:\Users\Joseph\Desktop\compiler design\expt1> flex expt1a.l
PS C:\Users\Joseph\Desktop\compiler design\expt1> gcc lex.yy.c -o lex.exe
PS C:\Users\Joseph\Desktop\compiler design\expt1> ./lex
1
1 is an INTEGER
3.14
3.14 is a REAL NUMBER
hello
hello is a WORD
bye 12 1.11
bye is a WORD
12 is an INTEGER
1.11 is a REAL NUMBER
```

PROGRAM 2:

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

int yylex(void);
int yywrap(void);
%}

%%
[0-9]+  {
   int num = atoi(yytext);
   int n = num;
   char hex[32];
   int i = 0;
   printf("Decimal: %d\n", num);
   if (num == 0) {
```

```
      printf("Step 1: %d / 16 = 0, remainder = 0\n", num);
      printf("Hexadecimal: 0\n");
   } else {
      while (n > 0) {
         int rem = n % 16;
         printf("Step %d: %d / 16 = %d, remainder = %d", i+1, n, n/16, rem);
         if (rem >= 10)
            printf(" ('%c')", 'A' + (rem - 10));
         printf("\n");
         hex[i++] = (rem < 10) ? ('0' + rem) : ('A' + (rem - 10));
         n /= 16;
      }
      printf("Hexadecimal: ");
      for (int j = i - 1; j >= 0; j--) putchar(hex[j]);
      printf("\n");
   }
}
.|\n    ;
%%

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

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

OUTPUT 2:

```
PS C:\Users\Joseph\Desktop\compiler design\expt1> flex expt1b.l
PS C:\Users\Joseph\Desktop\compiler design\expt1> gcc lex.yy.c -o lex.exe
PS C:\Users\Joseph\Desktop\compiler design\expt1> ./lex
10
Decimal: 10
Step 1: 10 / 16 = 0, remainder = 10 ('A')
Hexadecimal: A
11
Decimal: 11
Step 1: 11 / 16 = 0, remainder = 11 ('B')
Hexadecimal: B
15
```

```
Decimal: 15
Step 1: 15 / 16 = 0, remainder = 15 ('F')
Hexadecimal: F
16
Decimal: 16
Step 1: 16 / 16 = 1, remainder = 0
Step 2: 1 / 16 = 0, remainder = 1
Hexadecimal: 10
```

PROGRAM 3:

```
%{
#include <stdio.h>
int yylex(void);
int yywrap(void);
int lineno = 1;
%}

%%
.*\n    { printf("%d: %s", lineno++, yytext); }
.       { printf("%d: %s", lineno++, yytext); }
%%

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

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

sample_input.c:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int a, b, sum;

   printf("Enter two numbers: ");
   scanf("%d %d", &a, &b);

   sum = a + b;
```

```
    if (sum > 100) {
        printf("Large sum: %d\n", sum);
    } else {
        printf("Sum: %d\n", sum);
    }

    for (int i = 0; i < 5; i++) {
        printf("Loop iteration: %d\n", i);
    }

    return 0;
}
```

OUTPUT 3: (pasted directly in terminal)

```
PS C:\Users\Joseph\Desktop\compiler design\expt1> flex expt1c.l
PS C:\Users\Joseph\Desktop\compiler design\expt1> gcc lex.yy.c -o lex.exe
PS C:\Users\Joseph\Desktop\compiler design\expt1> ./lex
#include <stdio.h>
1: #include <stdio.h>
#include <stdlib.h>
2: #include <stdlib.h>

3:
int main() {
4: int main() {
    int a, b, sum;
5:     int a, b, sum;

6:
    printf("Enter two numbers: ");
7:     printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
8:     scanf("%d %d", &a, &b);

9:
    sum = a + b;
10:     sum = a + b;
```

```
11:
   if (sum > 100) {
12:    if (sum > 100) {
       printf("Large sum: %d\n", sum);
13:        printf("Large sum: %d\n", sum);
   } else {
14:    } else {
       printf("Sum: %d\n", sum);
15:        printf("Sum: %d\n", sum);
   }
16:    }

17:
   for (int i = 0; i < 5; i++) {
18:    for (int i = 0; i < 5; i++) {
       printf("Loop iteration: %d\n", i);
19:        printf("Loop iteration: %d\n", i);
   }
20:    }

21:
   return 0;
22:    return 0;
}
```

OUTPUT 3: (using file input)

```
C:\Users\Joseph\Desktop\compiler design\expt1> flex expt1c.l

PS C:\Users\Joseph\Desktop\compiler design\expt1> gcc lex.yy.c -o lex.exe

PS C:\Users\Joseph\Desktop\compiler design\expt1> Get-Content sample_input.c |
./lex.exe

1: #include <stdio.h>

2: #include <stdlib.h>

3:

4: int main() {
```

```
5:    int a, b, sum;
6:
7:    printf("Enter two numbers: ");
8:    scanf("%d %d", &a, &b);
9:
10:    sum = a + b;
11:
12:    if (sum > 100) {
13:        printf("Large sum: %d\n", sum);
14:    } else {
15:        printf("Sum: %d\n", sum);
16:    }
17:
18:    for (int i = 0; i < 5; i++) {
19:        printf("Loop iteration: %d\n", i);
20:    }
21:
22:    return 0;
23: }
```

PROGRAM 4:

```
%{
#include <stdio.h>
int yylex(void);
int yywrap(void);
```

```
int sum = 0, count = 0;
%}

%%
[0-9]+    { sum += atoi(yytext); count++; }
.|\n     ;
%%

int main() {
   yylex();
   if (count > 0)
      printf("Average = %.2f\n", (float)sum/count);
   else
      printf("No numbers entered.\n");
   return 0;
}

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

OUTPUT 4:

```
PS C:\Users\Joseph\Desktop\compiler design\expt1> flex expt1d.l
PS C:\Users\Joseph\Desktop\compiler design\expt1> gcc lex.yy.c -o lex.exe
PS C:\Users\Joseph\Desktop\compiler design\expt1> ./lex
12 13 14 15
Average = 13.50
PS C:\Users\Joseph\Desktop\compiler design\expt1> gcc lex.yy.c -o lex.exe
PS C:\Users\Joseph\Desktop\compiler design\expt1> ./lex

No numbers entered.
```

CONCLUSION:

This experiment demonstrated the use of Lex for pattern recognition, text processing, and simple computations. We successfully identified integers, real numbers, and words, converted decimal numbers to hexadecimal, added line numbers to source code, and computed averages. Key learnings include using regular expressions, integrating Lex with C, handling input efficiently, and performing computations on matched patterns, which are fundamental skills for building lexical analyzers and compiler components.

Roll No: 22B-CO-023 Batch: D Name: Joseph Jonathan Fernandes