Sankalchand Patel College of Engineering-Visnagar

Affiliated
to
Gujarat Technological University, Ahmedabad

Computer Engineering Department

Laboratory Manual

B. E. Third Year (Semester– VI)

Sub: System Programming (160706)

Sankalchand Patel College of Engineering-Visnagar

G'Nagar-Ambaji link road,
Visnagar- 384315, Dist. Mehsana (N.G.)

Phone No. (02765) 232008 , 227342,
Fax No. (02765) 224982

Nootan Sarva Vidhyalay Kelvani Mandal-VISNAGAR

Sankalchand Patel College of Engineering, Visnagar

G'Nagar-Ambaji link road,
Visnagar- 384315,

Dist. Mehsana (N.G.)

Name : _____

Address : _____

_____

Subject : _____

Branch : _____

Enrollment No : _____

Sankalchand Patel College of Engineering-Visnagar

Computer Engineering Department

Certificate

This is to certify that

Mr./Ms._____Enrollment No._____of ***6th Semester*** *Degree course in* ***Computer Engineering*** *has satisfactorily completed his/her term work in* ***System Programming (160706)*** *from _____ to _____ within four walls of SPCE, Visnagar.*

Date of Submission _____

Staff in charge _____

Head (CE/IT) _____

Sankalchand Patel College of Engineering-Visnagar

Computer Engineering Department

Index

Name:_____ Enrollment No._____

Sr No	Practical	Page	Date of start	Date of Completion	Initials of Staff	Remark
1.	System Calls (open(), read(), write(), close()) in C Language.	1				
2.	System Calls (fork(), wait()) in C Language.	4				
3.	Study of Lex - A Lexical Analyzer Generator.	7				
4.	Write a Lex program to count the number of vowels and consonants in a given string.	11				
5.	Write a Lex program to count the number of characters, words, spaces, end of lines in a given input file.	13				
6.	Write a Lex program to count no of: a) +ve and –ve integers b) +ve and –ve fractions	15				
7.	Write a Lex program to recognize whether a given sentence is simple or compound.	17				
8.	Write a Lex program to count the no of ‘scanf’ and ‘printf’ statements in a C program. Replace them with ‘readf’ and ‘writef’ statements respectively.	19				
9.	Write a Lex program to recognize and count the number of identifiers in a given input file.	21				
10.	Study of Device Drivers.	23				
11.	Study of Common Object File Format.	25				
12.	Implement a program for a Predictive Parser.	27				

Practical – 1

Aim: System Calls (open(), read(), write(), close()) in C Language.

A system call is just what its name implies—a request for the operating system to do something on behalf of the user's program

- open()
- read()
- write()
- close()

open()

```
#include <fcntl.h>
int open( char *filename, int access, int permission );
```

The available access modes are

O_RDONLY O_WRONLY O_RDWR

O_APPEND O_BINARY O_TEXT

The permissions are

S_IWRITE S_IREAD S_IWRITE | S_IREAD

The open() function returns an integer value, which is used to refer to the file. If unsuccessful, it returns -1, and sets the global variable errno to indicate the error type.

read()

```
#include <fcntl.h>
int read( int handle, void *buffer, int nbyte );
```

The read() function attempts to read nbytes from the file associated with handle, and places the characters read into buffer. If the file is opened using O_TEXT, it removes carriage returns and detects the end of the file.

The function returns the number of bytes read. On end-of-file, 0 is returned, on error it returns -1, setting errno to indicate the type of error that occurred.

write()

```
#include <fcntl.h>
int write( int handle, void *buffer, int nbyte );
```

The write() function attempts to write nbytes from buffer to the file associated with handle. On text files, it expands each LF to a CR/LF.

The function returns the number of bytes written to the file. A return value of -1 indicates an error, with errno set appropriately.

close()

```
#include <fcntl.h>
int close( int handle );
```

The close() function closes the file associated with handle. The function returns 0 if successful, -1 to indicate an error, with errno set appropriately.

Implementation of open(), read(), write() and close() functions

Pract1.c

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

int main()
{
      int fd;
      char buffer[80];
      static char message[]="Hello, SPCE - Visnagar";
      fd=open("myfile.txt",O_RDWR);
      if (fd != -1)
            {
            printf("myfile.txt opened with read/write access\n");
            write(fd,message,sizeof(message));
            lseek(fd,0,0);
            read(fd,buffer,sizeof(message));
            printf("%s -- was written to myfile.txt \n",buffer);
            close(fd);
            }
}
```

Note: First create empty file with **myfile.txt** before running the commands.

Commands: # gcc –o pr pract1.c
./pr

Output & Conclusion by student:

Practical – 2

Aim: System Calls (fork(), wait()) in C Language.

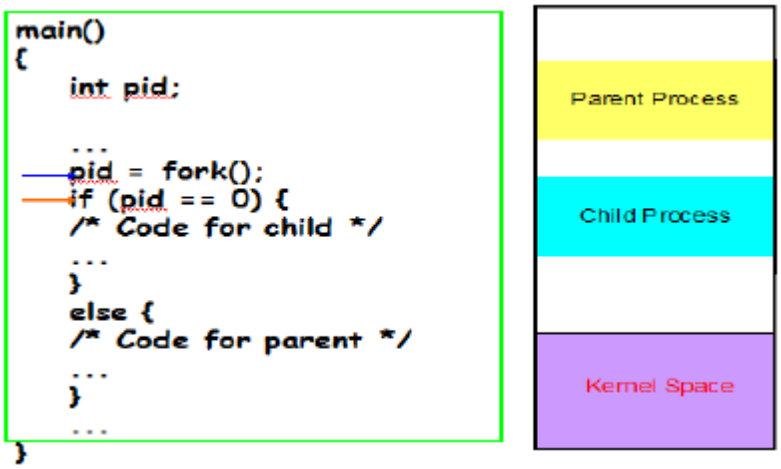
fork()

When the fork system call is executed, a new process is created which consists of a copy of the address space of the parent. The return code for fork is zero for the child process and the process identifier of child is returned to the parent process.

On success, both processes continue execution at the instruction after the fork call.

On failure, -1 is returned to the parent process.

Fork() – Sample Code



Implementation of fork() system call using C program.

Pract2a.c

```
#include <sys/types.h>
main()
{
      pid_t pid;
      pid = fork();
      if (pid == 0)
            printf("\n I'm the child process");
      else if (pid > 0)
            printf("\n I'm the parent process. My child pid is %d", pid);
      else
            perror("error in fork");
}
```

Commands: # gcc –o pr pract2a.c
./pr

wait()

 The wait system call suspends the calling process until one of its immediate children terminates.

 If the call is successful, the process ID of the terminating child is returned.

 Zombie process—a process that has terminated but whose exit status has not yet been received by its parent process or by init.

 pid_t wait(int *status);

where status is an integer value where the UNIX system stores the value returned by child process.

Implementation of wait() system call using C program.

Pract2b.c

```
#include <stdio.h>
void main()
{
      int pid, status;
      pid = fork();
      if(pid == -1) {
            printf("fork failed\n");
            exit(1);
      }
      if(pid == 0) { /* Child */
            printf("Child here!\n");
      }
      else { /* Parent */
            wait(&status);
            printf("Well done Child!\n");
      }
}
```

Commands: # gcc –o pr pract2b.c
./pr

Output & Conclusion by student:

Practical – 3

Aim: Study of Lex - A Lexical Analyzer Generator.

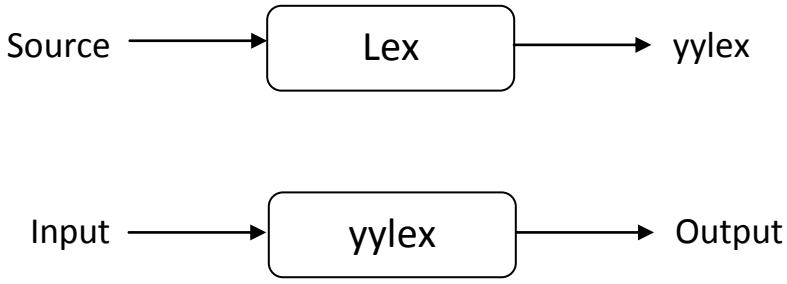
Introduction

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See below Figure.



For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$   ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates ``one or more ..."; and the $ indicates ``end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$   ;
[ \t]+    printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex Source Definitions

The general format of Lex source is:

{definitions}
%%
{rules}
%%
{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule. As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is name translation and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                              [0-9]
E                              [DEde][-+]?{D}+
%%
{D}+                           printf("integer");
 {D}+"."{D}*({E})?             |
{D}*"."{D}+({E})?              |
{D}+{E}
```

Usage:

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named lex.yy.c. The I/O library is defined in terms of the C standard library.

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same.

UNIX. The library is accessed by the loader flag -ll. So an appropriate set of commands is lex source cc lex.yy.c -ll The resulting program is placed on the usual file a.out for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of input, output and unput are given, the library can be avoided.

Practical – 4

Aim: Write a Lex program to count the number of vowels and consonants in a given string.

```
%{
     #include<stdio.h>
     int vowels=0;
     int cons=0;
%}
%%
[aeiouAEIOU] {vowels++;}
[a-zA-Z] {cons++;}
%%
int yywrap()
{
     return 1;
}
main()
{
     printf("Enter the string.. at end press ^d\n");
     yylex();
     printf("No of vowels=%d\nNo of consonants=%d\n",
     vowels,cons);
}
```

How to Compile:

1. Save the file wite *.l* extention. E.g. *vowels.l*
2. *lex vowels.l*
3. *gcc –o objfile lex.yy.c*
4. *./objfile*

Output & Conclusion by student:

Practical – 5

Aim: Write a Lex program to count the number of characters, words, spaces, end of lines in a given input file.

```
%{
     #include<stdio.h>
     int c=0, w=0, s=0, l=0;
%}
WORD  [^   \t\n,\.:]+
EOL  [\n]
BLANK  [  ]
%%
{WORD} {w++; c=c+yyleng;}
{BLANK} {s++;}
{EOL} {l++;}
.  {c++;}
%%
int yywrap()
{
     return 1;
}
main(int argc, char *argv[])
{
     if(argc!=2)
     {
          printf("Usage: <./a.out> <sourcefile>\n");
          exit(0);
     }
     yyin=fopen(argv[1],"r");
     yylex();
     printf("No of characters=%d\nNo of words=%d\nNo of
     spaces=%d\n No of lines=%d\n",c,w,s,l);
}
```

Output & Conclusion by student:

Practical – 6

Aim: Write a Lex program to count no of:

a) +ve and –ve integers b) +ve and –ve fractions

```
%{
     #include<stdio.h>
     int posint=0, negint=0,posfraction=0, negfraction=0;
%}
%%
[-][0-9]+ {negint++;}
[+]?[0-9]+ {posint++;}
[+]?[0-9]*\.[0-9]+ {posfraction++;}
[-][0-9]*\.[0-9]+ {negfraction++;}
%%
int yywrap()
{
     return 1;
}

main(int argc, char *argv[])
{
     if(argc!=2)
     {
          printf("Usage: <./a.out> <sourcefile>\n");
          exit(0);
     }
yyin=fopen(argv[1],"r");
yylex();
printf("No of +ve integers=%d\n No of -ve integers=%d\n No of
+ve fractions=%d\n No of -ve fractions=%d\n", posint, negint,
posfraction, negfraction);
}
```

Output & Conclusion by student:

Practical – 7

Aim: Write a Lex program to recognize whether a given sentence is simple or compound.

```
%{
      #include<stdio.h>
      int is_simple=1;
%}
%%
[   \t\n]+[aA][nN][dD][   \t\n]+ {is_simple=0;}
[   \t\n]+[oO][rR][   \t\n]+ {is_simple=0;}
[   \t\n]+[bB][uU][tT][   \t\n]+ {is_simple=0;}
. {;}
%%
int yywrap()
{
      return 1;
}

main()
{
      int k;
      printf("Enter the sentence.. at end press ^d");
      yylex();
      if(is_simple==1)
      {
            printf("\nThe given sentence is simple");
      }
      else
      {
            printf("\nThe given sentence is compound");
      }
}
```

Output & Conclusion by student:

Practical – 8

Aim: Write a Lex program to count the no of 'scanf' and 'printf' statements in a C program. Replace them with 'readf' and 'writef' statements respectively.

```
%{
#include<stdio.h>
int pc=0, sc=0;
%}
%%
"printf" { fprintf(yyout,"writef"); pc++;}
"scanf" { fprintf(yyout,"readf"); sc++;}
%%
int yywrap()
{
return 1;
}

main(int argc, char *argv[])
{
     yyin=fopen(argv[1],"r");
     yyout=fopen(argv[2],"w");
     yylex();
     printf("No of printf statements = %d\nNo of scanf
     statements=%d\n", pc, sc);
}
```

Output & Conclusion by student:

Practical – 9

Aim: Write a Lex program to recognize and count the number of identifiers in a given input file.

```
%{
      #include<stdio.h>
      int id=0;
%}
%%
[a-zA-Z][a-zA-Z0-9_]* { id++ ; ECHO; printf("\n");}
.+ { ;}
\n { ;}
%%
int yywrap()
{
return 1;
}

main (int argc, char *argv[])
{
      if(argc!=2)
      {
            printf("Usage: <./a.out> <sourcefile>\n");
            exit(0);
      }
      yyin=fopen(argv[1],"r");
      printf("Valid identifires are\n");
      yylex();
      printf("No of identifiers = %d\n",id);
}
```

Output & Conclusion by student:

Practical – 10

Aim: Study of Device drivers.

Theory:

Introduction:

The purpose of a device driver is to handle requests made by the kernel with regard to a particular type of device. There is a well defined and consistent interface for the kernel to make these requests. By isolating device specific code in device drivers and by having a consistent interface to the kernel, adding a new device is easier.

A device driver is a software module that resides within the Digital Unix kernel and is the software interface to a hardware device or devices. A hardware device is a peripheral, such as a disk controller , tape controller , or network controller device. In general , there is a one device driver for each type of hardware device. Device drivers can be classified as:

1) Block Device Drivers
2) Character Device Drivers(including terminal drivers)
3) Network Device Drivers
4) Pseudodevice drivers.

1. Block Device Driver

A block device driver is a driver that performs I/O by using file system block sized buffers from a buffer cache supplied by the kernel. The kernel also provides for the device driver support interfaces that copy data between the buffer cache and the address space of a process.
Block device drivers are particularly well- suited for disk drives, the most common block devices. For block devices , all I/O occurs through the buffer cache.

2. Character Device Driver:

A character device driver does not handle I/O through the buffer cache, so it is not tied to a single approach for handling I/O. You can use a character device driver for a device such as a line printer that handles one character at a time. However , character drivers are not limited to performing I/O one character at a time. (despite the name " character" driver) . For example , tape drivers frequently perform I/O in 10 K chunks. You can also use a character device driver when it is

necessary to copy data directly to or from a user process. Because of their flexibility in handling I/O , many drivers are character drivers . Line printers, interactive terminals, and graphics displays are examples of devices that require character device drivers.

A terminal device driver is actually a character device driver that handles I/O character processing for a variety of terminal devices. Like any character device, a terminal device can accept or supply a stream of data based on a request from a user process. It cannot be mounted as a file system and therefore does not use data caching.

3. Network Device Driver

A network Device driver attaches a network subsystem to a network interface , prepares the network interface for operation, and governs the transmission and reception of network frames over the network interface.

4. Psedodevice Driver

Not all device drivers control physical hardware . Such device drivers are called “peudodevice” drivers. Like block and character device drivers , pseudodevice drivers make use of the device driver interfaces. Unlike block and character device drivers, pseudodevice drivers do not operate on a bus. One example of a pseudodevice driver is the pseudoterminal or pty terminal driver , which simulates a terminal device . The pty terminal driver is a character device driver typically used for remote logins.

Viva Voice:

1) What is a device driver?
2) What is the difference between character driver and block driver?
3) What are the header files included for line printer driver?
4) What are the major design issues for designing a RAM derive?
5) How do device drivers handle interrupts?

Practical – 11

Aim: Study of Common Object File Format.

Theory:

Common Object File Format(COFF). COFF is the format of the output file produced by the assembler and the link editor.

The following are some key features of COFF:

- Applications can add system dependent information to the object file without causing access utilities to become obsolete.
- Space is provided for symbolic information used by debuggers and other applications.
- Programmers can modify the way the object file is constructed by providing directives at compile time.

The object file supports user –defined sections and contains extensive information for symbolic software testing. An object file contains:

- A File header
- Optional header information
- A table of section headers
- Data corresponding to the section headers.
- Relocation information
- Line numbers
- A symbol table
- A String table

Object File Format

FILE HEADER
Optional Information
Section 1 Header
…
Section n Header
Raw Data for Section 1
…..
Raw Data for Section n
Relocation Info for Sect. 1
…
Relocation Info for Sect .n
Line Numbers for Sect.1
…
Line Numbers for Sect. n
SYMBOL TABLE
STRING TABLE

Viva Voice:

1) What are the header files included in File header section?

2) Distinguish between Symbol table and String table entries.

3) Give difference between Static and Dynamic relocation

4) What are the contents of section headers?

Practical – 12

Aim: Implement a program for a predictive parser (grammar from the book).

```
#include<stdio.h>
#include<string.h>
#include<conio.h>

char a[10];
int top = -1,i;
void error()
{
      printf("Syntax Error");
}
void push(char k[]) //Pushes The Set Of Characters on to the Stack
{
      for(i=0;k[i] != '\0';i++)
      {
            if(top < 9)
                  a[++top] = k[i];
      }
}
char TOS()        //Returns TOP of the Stack
{
      return a[top];
}
void pop()       //Pops 1 element from the Stack
{
      if(top >= 0)
      a[top--] = '\0';
}
void display()  //Displays Elements Of Stack
{
      for(i=0;i <= top;i++)
            printf("%c",a[i]);
}
void display1(char p[],int m) //Displays The Present Input String
{
      int l;
      printf("\t");
      for(l=m;p[l] != '\0';l++)
            printf("%c",p[l]);
}
char* stack()
{
      return a;
}
int main()
{
      char ip[20],r[20],st,an;
      int ir,ic,j = 0,k;
      char t[5][6][10] = {"$","$","TH","$","TH","$","+TH","$",
                          "e","e","$","e","$","$","FU","$",
                          "FU","$","e","*FU","e","e","$","e",
                          "$","$","(E)","$","i","$"};
```

```
	clrscr();
	printf("\nEnter any String(Append with $) : ");
	gets(ip);
	printf("\n");
	printf("Stack\tInput\tOutput\n\n");
	push("$E");
display();
	printf("\t%s\n",ip);
	for(j=0;ip[j] != '\0';)
	{
		if(TOS() == an)
		{
			pop();
			display();
			display1(ip,j+1);
			printf("\tPOP\n");
			j++;
		}
		an = ip[j];
		st = TOS();
		if(st == 'E')
			ir = 0;
		else if(st == 'H')
				ir = 1;
		else if(st == 'T')
				ir = 2;
		else if(st == 'U')
				ir = 3;
		else if(st == 'F')
				ir = 4;
		else
		{
			error();
			break;
		}
		if(an == '+')
			ic = 0;
		else if(an == '*')
				ic = 1;
		else if(an == '(')
				ic = 2;
		else if(an == ')')
				ic = 3;
		else if((an >= 'a' && an <= 'z') || (an >= 'A' && an <= 'Z'))
		{
			ic = 4;
			an = 'i';
}
		else if(an == '$')
				ic = 5;
		strcpy(r,strrev(t[ir][ic]));
		strrev(t[ir][ic]);
		pop();
		push(r);
		if(TOS() == 'e')
		{
			pop();
```

```
                display();
                display1(ip,j);
                printf("\t%c -> %c\n",st,238);
          }
          else
          {
                display();
                display1(ip,j);
                printf("\t%c -> %s\n",st,t[ir][ic]);
          }
          if(TOS() == '$' && an == '$')
                break;
          if(TOS() == '$')
          {
                error();
                break;
          }
     }
     k = strcmp(stack(),"$");
     if(k == 0)
          printf("\n Given String is accepted");
     else
          printf("\n Given String is not accepted");

     getch();
     return 0;
}
```

Output & Conclusion by student: