PRACTICAL FILE

Department:	**Computer Science and Engineering**
Session:	**January - June**
Subject:	**System Programming**
Subject Code:	**BTCS-409**
Semester:	**4th**

Syllabus

1. Create a menu driven interface for
a) Displaying contents of a file page wise
b) Counting vowels, characters, and lines in a file.
c) Copying a file

2. Write a program to check balance parenthesis of a given program. Also generate the error report.

3. Write a program to create symbol table for a given assembly language program.

4.Write a program to create symbol table for a given high-level language program.

5.Implementation of single pass assembler on a limited set of instructions.

6.Exploring various features of debug command.

7.Use of LAX and YACC tools

List of Practical

Sr. No.	Topic
1	Introducion to SP.and its components.
2	Create a menu driven interface for a) Displaying contents of a file page wise b) Counting vowels, characters, and lines in a file. c) Copying a file
3	Write a program to check balance parenthesis of a given program. Also generate the error report.
4	Write a program to create symbol table for a given assembly language program.
5	Write a program to create symbol table for a given high-level language program.
6	Implementation of single pass assembler on a limited set of instructions.
7	Exploring various features of debug command.
8	Use of LAX and YACC tools.
9	*Write a program to implement an absolute loader.

*Learning Beyond Syllabus Write a program to implement an absolute loader.

Experiment 1

AIM: INTRODUCTION OF SYSTEM PROGRAMMING. AND ITS COMPONENTS

INTRODUCTION

System programming (or systems programming) is the activity of computer programming system software. The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user (e.g. word processor), whereas systems programming aims to produce software which provides services to the computer hardware (e.g. disk defragmenter). It requires a greater degree of hardware awareness.

COMPONENTS

Linker

For modularity of the program, it is better to break programs into several modules (subroutines). It is even better to put common routine, like reading a hexadecimal number, writing a hexadecimal number etc. which could be used by a lot of other programs also into a separate file. These files are assembled (translated) separately. After each has been successfully assembled, they can be linked together to form a large file, which constitutes the computer program. The program that links several programs is called the linker.

Loader

A loader is a program that places programs into main memory and prepares them for execution. The loader's target language is machine language, its source language is nearly machine language. Loading is ultimately bound with the storage management function of operating systems and is usually performed later than assembly or compilation. The period of executions of user's program is called execution time. The period of translating a user's source program is called assembly or compile time. Load time refers to the period of loading and preparing an object program for execution

Compiler

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If

the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler

Assembler

An assembler is a program that takes basic computer instruction and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term *assembly language*.

Compiler

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language

Macro

A macro in computer science is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to a replacement output sequence (also often a sequence of characters) according to a defined procedure. The mapping process that instantiates (transforms) a macro use into a specific sequence is known as *macro expansion*. Macros are used to make a sequence of computing instructions available to the programmer as a single program statement, making the programming task less tedious and less error-prone.

Experiment 2

AIM: Create a menu driven interface for

a) Displaying contents of a file page wise

b) Counting vowels, characters, and lines in a file.

c) Copying a file

```
#include<iostream.h>
#include<conio.h>
void main()
{
//clear the screen.
clrscr();
//declare variable type float and char
float a,b,area;
char ch;
//Input the choice.
cout<<"Enter c for circle"<<endl;
cout<<"Enter s for square"<<endl;


cout<<"Enter r for rectangle"<<endl;
cout<<"Enter t for triangle"<<endl;
cin>>ch;
//conditional switch statement.
switch (ch)
{
case 'c':
cout<<"Enter radius"<<endl;
cin>>a;
area=3.14*a*a;
break;
case 's':
cout<<"Enter the side"<<endl;
cin>>a;
area=a*a;
break;
case 'r':
cout<<"Enter the length"<<endl;
cin>>a;
cout<<"Enter the breadth"<<endl;
cin>>b;
area=a*b;
break;
case 't':
cout<<"Enter the height"<<endl;
cin>>a;
cout<<"Enter the base"<<endl;
cin>>b;
area=0.5*a*b;
break;
default:
cout<<"Syntax Error";
}
//print the area.
cout<<"Area is "<<area;
//get character
getch();
}
```

Experiment 3

AIM: Write a program to check balance parenthesis of a given program.

```
#include <iostream.h>

#include <conio.h>
#include "IntStack.h"
#include <stdio.h>

void main(void)
{
    char bracket[20];
    gets (bracket);
    char arr[6];
    int i=0;
    while(i<20)
    {
        switch(bracket[i])
            {
                case '[':
                {
                    arr[0]=1;
                    break;
                }
                case '{':
                {
                    arr[1]=2;
                    break;
                }
                case '(':
                {
                    arr[2]=3;

 break;
                }
                case ')':
                {
                    arr[3]=3;
                    break;
                }
                case '}':
                {
                    arr[4]=2;
                    break;
                }
            case ']':
            {
               arr[5]=1;
               break;
            }
            default:
               cout<<"";
      }
      i++;
   }
   if(arr[3]==arr[2])

cout<<"";
   else
      cout<<" ) or ( is missing "<<endl;

   if(arr[1]==arr[4])
      cout<<"";
   else
      cout<<" } or { is missing "<<endl;

   if(arr[5]==arr[0])
      cout<<"";
   else
      cout<<" ] or [ is missing"<<endl;
}
```

OUTPUT:-

Input

```
()
```

Output

```
} or { is missing
 ] or [ is missing
```

Experiment 4

AIM: To write a program to generate the symbol table for the given assembly language.

```
#include<stdio.h>
#include<conio.h>
struct sym
{
char lab[10];
int val;
};
void main ()
{
FILE *f1;
char la[10],op[10],opr[10],a[1000],c,key[10];
int i,j,lc=0,m=0,flag,ch=0;
struct sym s[10];
clrscr();
f1=fopen("a1.txt","r");
c=fgetc(f1);
i=0;
printf ("\n SOURCE PROGRAM \n");
while(c!=EOF)
{
a[i]=c;
c=fgetc(f1);
i++;
}

while(ch<4)
{
printf("1-symbol table creation\n");
printf("2-serch\n");
printf("3-display\n");
printf(">3-Exit\n");
printf("enter ur choice\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
i=0;
while(strcmp(op,"end")!=0)
{
if(a[i]=='\t')
{
strcpy(la," ");
i++;
}
else
{
j=0;
while(a[i] !='\t')
{
la[j]=a[i];
i++;
j++;

}
la[j]='\0';
i++;
}
if(a[i]=='\t')
{
strcpy(op," ");
i++;
}
else
{
j=0;
while(a[i]!='\t')
{
op[j]=a[i];
i++;
j++;
}
op[j]='\0';
i++;
}
if(a[i]=='\t')
{
strcpy(opr," ");
i++;
}
else

{
j=0;
while(a[i]!='\n')
{
opr[j]=a[i];
i++;
j++;
}
opr[j]='\0';
i++;
}
```

```
j=0;
if(strcmp(la," ")!=0)
{
strcpy(s[m].lab,la);
if(strcmp(op,"start")==0)
{
lc=atoi(opr);
s[m].val=lc;
m++;
printf("%s\t%s\t%s\n",la,op,opr);
continue;
}
else if(strcmp(op,"equ")==0)
{
s[m].val=atoi(opr);
m++;

}
else if(strcmp(op,"resw")==0)
{
s[m].val=lc;
lc=lc+atoi(opr) *3;
m++;
}
else if(strcmp(op,"resb")==0)
 {
s[m].val=lc;
lc=lc+atoi(opr);
m++;
}
else
{
s[m].val=lc;
lc=lc+3;
m++;
}
}
else
lc=lc+3;
printf("%s\t%s\t%s\n",la,op,opr);
}
break;
case 2:
printf("enter the lable to be searched\n");

scanf("%s",&key);
flag=0;
for(i=0;i<m;i++)
{
if(strcmp(key,s[i].lab)==0)
```

```
{
printf("%s\t%d\n",s[i].lab,s[i].val);
flag=1;
break;
}
else
continue;
}
if(flag==0)
printf("lable not found\n");
break;
case 3:
printf("\n symbol table \n");
for(i=0;i<m;i++)
printf("\n%s\t%d\n",s[i].lab,s[i].val);
break;
}
}
}
```

Output :

```
add     start   1000
        lda     one

add     val
        sta     two
val     equ     10
one     word    100
two     resw    1
        end     add
1-symbol table creation
2-serch
3-display
>3-Exit
enter ur choice
3
symbol table
add     1000
val     10
one     1009
two     1012
1-symbol table creation
2-serch
3-display
>3-Exit
enter ur choice
2
enter the lable to be searched
val
val     10
1-symbol table creation
```

```
2-serch
3-display
>3-Exit
enter ur choice
2
enter the lable to be searched
qw
lable not found
1-symbol table creation
2-serch
3-display
>3-Exit
enter ur choice
4
```

Result :- Thus we write a program to generate the symbol table for the given assembly language

Experiment 5

AIM: Write a program to create symbol table for a given high-level language program.

```
#include<stdlib.h>
#include<stdio.h>
#include <iostream.h>
#include <conio.h>
void main()
{
char a;
int i;

for(i=0;i<255;i++)
{
a = i;
cout<<"\nSymbol "<<i<<" = "<<a;
}
getch();
}
```

OUTPUT:-

```
Symbol 0 =
Symbol 1 = ☺
Symbol 2 = ☻
Symbol 3 = ♥
Symbol 4 = ♦
Symbol 5 = ♣
Symbol 6 = ♠
Symbol 7 =
Symbol 8 =
Symbol 9 =
Symbol 10 =

Symbol 11 = ♂
Symbol 12 = ♀
Symbol 13 =
Symbol 14 = ♫
Symbol 15 = ☼
Symbol 16 = ►
Symbol 17 = ◄
Symbol 18 = ↕
Symbol 19 = ‼
Symbol 20 = ¶
Symbol 21 = §
Symbol 22 = ▬
Symbol 23 = ↨
```

Symbol 24 = ↑
Symbol 25 = ↓
Symbol 26 =

Result:- :- Thus we write a program to generate the symbol table for the given high-level language program.

Experiment 6

AIM: Implementation of single pass assembler on a limited set of instructions.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
void main()
{
FILE *pf,*fp;
 struct instruction
 {
   char label[10];
   char opcode[10];
   char operand[10];
 }il;

struct symtab
 {
   char name[10];
   int address;
 }s[20];

struct optab
 {
   char opn[20];
   char mc[20];

 }o[5]={{"ADD","14"},{"LDA","50"},{"STA","02"},
           {"JSUB","12"},{"JEQU","04"}};

struct object
 {
  int address;
  char objcode[20];
 }obj;

int locctr=0,i,j,no,f,length,ns=0,n,v,l;
char *p;
clrscr();
fp=fopen("assemble.txt","r");
pf=fopen("object.txt","w+");
fscanf(fp,"%s%s%s\n",il.label,il.opcode,il.operand);
if(strcmp(il.opcode,"START")==0)
 {
   locctr=atoi(il.operand);
   v=locctr;
   fscanf(fp,"%s%s%s\n",il.label,il.opcode,il.operand);
 }
else
   locctr=0;
while(strcmp(il.opcode,"END")!=0)
 {
   f=1;
   if(strcmp(il.label,"-")!=0)

 {
     for(i=0;i<ns;i++)
     {
      if(strcmp(il.label,s[i].name)==0)
      {
            f=0;
            printf("There is an error due to the duplication");
            exit(0);
            }
     }
     if(f==1)
     {
       strcpy(s[ns].name,il.label);
       s[ns].address=locctr;
       ++ns;
     }
   }
   if(strcmp(il.opcode,"RESW")==0)
   {
     strcpy(obj.objcode,"X");
     length=3*atoi(il.operand);
     locctr+=length;
   }
   else if(strcmp(il.opcode,"WORD")==0)
   {
     strcpy(obj.objcode,"00000");
     strcat(obj.objcode,il.operand);

 length=3;
     locctr+=length;
   }
   else if(strcmp(il.opcode,"RESB")==0)
   {
     strcpy(obj.objcode,"N");
     length=atoi(il.operand);
     locctr+=length;
   }
   else if(strcmp(il.opcode,"BYTE")==0)
   {
      strcpy(obj.objcode,"454f46");
      n=strlen(il.operand);
      length=n-3;
      locctr+=length;
    }
    else
    {
      length=3;
      locctr+=length;
      for(i=0;i<5;i++)
      {
        if(strcmp(il.opcode,o[i].opn)==0)
        {
              no=0;
              for(j=0;j<ns;j++)
              {

 if(strcmp(il.operand,s[j].name)==0)
               {
                no++;
                strcpy(obj.objcode,o[i].mc);
                itoa(s[j].address,p,10);
                strcat(obj.objcode,p);
                break;
               }
              }
              if(no==0)
              {
              printf("There is an undefined symbol");
              exit(0);
              }
        }
      }
    }
    obj.address=locctr-length;
    fprintf(pf,"%d %s %s\n",obj.address,obj.objcode);
    fscanf(fp,"%s %s %s\n",il.label,il.opcode,il.operand);
    }
l=locctr-v;
fclose(pf);
fclose(fp);
printf("The one pass assembler is run successfully\n");
printf("The length of the program is %d\n",l);
printf("SYMBOL TABLE\n");

printf("NAME\t\tADDRESS\n");
printf("~~~~\t\t~~~~~~~\n");
for(i=0;i<ns;i++)
 {
 printf("%s\t\t%d\n",s[i].name,s[i].address);
```

```
 }
getch();
 }
```

<u>INPUT FILE(assemble.txt)</u>

```
ADDN START 1000
FIRST WORD 5
TWO WORD 8
RESULT RESW 2
TEMP BYTE C'EOF'
PROG LDA FIRST
- ADD TWO
- STA RESULT
- END PROG
```

<u>OUTPUT(OBJECT.TXT)</u>

```
ADDR CODE
1000 000005
1003 000008
1006 X
1012 454f46
1015 501000

1018 141003
1021 021006
```

Result :- Thus we write a C program to Implement Single pass Assembler.

Experiment 7

AIM: Exploring various features of debug command.

All commands are available when the full debugger is in use.
Restrictions
If you have a Command Level or Break/End Restart feature in effect, or the break key is disabled, the available options are restricted to:

a	**Abort program**
q	**Quit program**
c	**Continue (may be allowed, depends on reason debug was entered) end Terminate debugger**
o	**Log off**

Note that there are several ways in which the break key can be disabled. You can use commands such as INHIBIT-BREAK-KEY or BREAK- KEY-OFF. In these cases, the debugger is never entered. Another way is to execute a BREAK OFF statement within the program. In this case, the debugger will still be invoked if a run-time error occurs - such as trying to read a record from a non-file variable.
Command List
?
Display a help screen showing all available debug commands and the program status.
>filename
Open and truncate the file filename and send it the current breakpoints and trace table entries. This can be used in future to replicate the current environment by the use of the command. note that you may write debugger scripts yourself with an editor rather than use the > command.
<filename
Open the file filename, then read and execute each line as if it has been entered at the keyboard. Any current trace or breakpoint table entries are deleted then replaced by those recorded in filename.

!command
Spawn another process and execute the command. The previous command thus used can also be recalled and executed by the !! command.
<Ctrl>+D
Display the next 11 lines of source in the current file.
Nn
Set the current display line to line nn in the current file and then display the line. Note that the program execution counter remains unchanged, it is only the display pointer that is changed. A command such as s (see later) will correctly execute the next line in the programmed sequence, not the newly displayed line.
#text
Ignored, and so can be used as a comment line in debugger scripts later executed with the command.

a{-nn} {mm}
Kills the program and any parent process or program that called it. The program aborts with an exit code of 203, and the kill signal is sent to any parent process. The nn value is used to change the exit code, whilst the mm value changes the signal number sent with the kill command. Operation of this command can be altered by setting the Command Level Restart option - see the Systems Housekeeping chapter of the System Administrators Reference Manual for more details.
b
Display all currently active breakpoints.
b {-t} nn{,file}
Set a breakpoint at line nn in the current file or that specified by the file modifier. If the -t option is specified then the breakpoint will cause a display of all the trace variables rather than halting the program.
b {-t} varname
This form of the b command will cause the debugger to be entered whenever the contents of the specified variable are changed.
b {-t} ex1 op ex2 {AND|OR .....}
Set a breakpoint at the line whose value is obtained by performing the operation op on expressions ex1 and ex2. The operator can be one of eq, !=, <>equal is also available. See later for a full description of expressions. The -t option will cause the debugger to display all the trace points rather than halting program execution.
c
Continue execution of the program.
d {-tbed} {*nn}
Delete breakpoint and/or trace table entries, and will normally prompt for confirmation. The t and b switches refer to trace and breakpoints respectively. The * switch deletes all of the specified entries without prompting. The nn switch deletes the entry nn in the given trace or breakpoint table, also without prompting. The d and e switches respectively disable or enable the given entry without removing it from the table.

e name
Edit the file specified by name. This file is then the file used by other debug commands such as <Ctrl>+D.
end
Synonym for "quit".
f {on|off}
A debug breakpoint is set for a filename change. This break can be set to on or off. If the program is continued (C command) the debugger will be entered the next time the source file changes.
h {-rs{n}} {nn|on|off }
Displays a history of the source lines executed, and current status of the debugger commands used. The on and off switches toggle the recording of lines executed, and when on, the nn value gives the number of executed lines to display (1024 maximum). The -r switch displays in reverse order, and -s{n} shows n source lines.
j {-g}
The j command displays a complete history of both GOSUB and external subroutine calls. When issued without options the command will only display information about the current program or subroutine. The -g (global) option will show a breakdown of the entire application.

l {-acf{nn}} text
Locate the string text in the current file. The switches used are: *a* to look for every occurrence; *c* to make the search case insensitive; *nn* to limit the search to the next *nn* lines; *f* to start the search from the start of the file. The command *l/* will execute the previous locate.
m
Displays the current memory status. Shows space allocated by the function malloc().
n {nn}
Displays the next nn lines of source from the current file, which is automatically loaded by the debugger if the p command has been used or it resides in the current working directory.
off
Enter o or off to log off. If you enter off (or OFF), the effect is immediate. If you enter o (or O), you will be prompted for confirmation. The same restrictions apply as for the OFF command; if there are non-jBASE programs active, OFF will only terminate jBASE programs until it encounters the first non-jBASE program - probably the login shell.
p {pathlist}
Defines the list of directories and pathnames (delimited by :) that the debugger will then search to find source codes. p without a pathlist displays the current Path.
q {nn}
Quit the program. nn is the termination status returned to any calling program.
r device
The debugger will take all input from, and send all output to, the specified device. Note that if the device is

another terminal (or Xterm shell), that you will need to prevent the target shell from interfering with the input stream by issuing the sleep command to it. A large value should be used or the sleep should be issued repeatedly in a loop.
s {-t{m}d} {nn}
Continue execution of jBC code in single line steps before returning to debug. The value nn changes the number of lines executed before returning to debug. The *-t* switch is used to display the trace table after every line executed, rather than wait for entry to debug. The *d* switch sets a delay before executing each line of code. m is used to set the delay in seconds (default is 5 deci-seconds).
S {-t{m}d} {nn}
Same as s except this will 'step over' subroutine calls, the code within the subroutine will not be displayed.
t
Display the current trace table.
t {-fg} expr
Add the value specified by expr to the trace table. When debug is entered, all the values in the table are displayed. The f switch is used to fully evaluate expr, whilst the g switch extends the display of expr to all levels.
v {-gmsrv} {expr}
Evaluate expr and display the result. The effects of the switches are: *g* to extend the display of expr to all data areas. *m* to allow variable modification within expr. When a variable is modified with the m option binary characters may be entered using the octal sequence \nnn. The sequence \010 would therefore be replaced by CHAR(8) in the modified variable. The sequence \\ evaluates to the single character \ and a sequence such as \x evaluates to the single character x (i.e. the \ will be lost).

To set a variable to null assign it the value: \0
Example:

```
jBASE debugger->v -m COLOR
  COLOR                    : GREEN = \0
jBASE debugger->V COLOR
  COLOR                    :
jBASE debugger->
```

To set a variable to character zero, i.e. CHAR (0), assign it the value: \00
w nn
Display a window of source code. The default is 9 lines with 4 before and after the current one. The value nn is used to change this parameter.

Experiment 8

AIM: Use of LAX and YACC tools.

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 depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language.

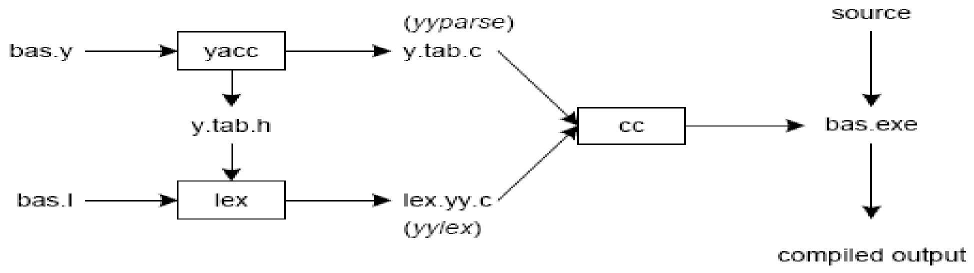

Figure 2: Building a Compiler with Lex/Yacc

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.

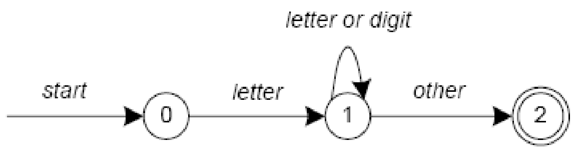

Figure 3: Finite State Automaton

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
```

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;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we have a match in the rules section the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

```
%{
    int nchar, nword, nline;
%}
%%
\n         { nline++; nchar++; }
[^ \t\n]+  { nword++, nchar += yyleng; }
.          { nchar++; }
%%
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    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 *shift-reduce* 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.

Practice, Part I

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

Input to yacc is divided into three sections. The definitions section consists of token declarations and C code bracketed by "**%{**" and "**%}**". The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

```
%token INTEGER
```

This definition declares an **INTEGER** token. Yacc generates a parser in file **y.tab.c** and an include file, **y.tab.h**:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex includes this file and utilizes the definitions for token values. To obtain tokens yacc calls **yylex**. Function **yylex** has a return type of int that returns a token. Values associated with the token are returned by lex in variable **yylval**. For example,

```
[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }
```

would store the value of the integer in **yylval**, and return token **INTEGER** to yacc. The type of **yylval** is determined by **YYSTYPE**. Since the default type is integer this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+]         return *yytext;        /* return operator */
```

the character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator. Generated token values typically start around 258 because lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator:

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+\n]      return *yytext;

[ \t]       ; /* skip whitespace */

.           yyerror("invalid character");

%%

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

Internally yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals that represent the current parsing state. The value stack is an array of **YYSTYPE** elements and associates a value with each element in the parse stack. For example when lex returns an **INTEGER** token yacc shifts this token to the parse stack. At the same time the corresponding **yylval** is shifted to the value stack. The parse and value stacks are always synchronized so finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator:

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

%token INTEGER

%%

program:
        program expr '\n'         { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                   { $$ = $1; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

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

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified and followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.

With left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected we print the value of the expression. When we apply the rule

```
expr: expr '+' expr           { $$ = $1 + $3; }
```

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case we pop "**expr '+' expr**" and push "**expr**". We have reduced the stack by popping three terms off the stack and pushing back one term. We may reference positions in the value stack in our C code by specifying "**$1**" for the first term on the right-hand side of the production, "**$2**" for the second, and so on. "**$$**" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. As a consequence the parse and value stacks remain synchronized.

Experiment 9

AIM: To write a program to Implement absolute Loader

```
#include <stdio.h>
#include <conio.h>
void main()
{
        FILE *txt;
        char c,programe[10],startadd[5],length[5];
        char *addr;
        int i,a,b;
clrscr();
txt=fopen("absinp.txt","r");
c=getc(txt);
        if(c=='H')
        {
                fscanf(txt,"%s %d %s",programe, &addr ,length);
                printf("\n\n STARTING ADDRESS =%u",addr);
        }
        while(c!='T')
        c=getc(txt);
if(c=='T')
{
        for(i=0;i<9;i++)
        c=getc(txt);

        c=getc(txt);
        while(c!='\n')

{
                a=((int)c-48);
                a*=10;
                c=getc(txt);
                b=((int)c-48);
                *addr = a+b;
                printf("\n\nADDRESS = %u VALUE STORED =%d",addr,*addr);
                addr++;
                c=getc(txt);
        }
}
getch();
}
```

INPUT:

```
H COPY 5000 1E
T 5000 1E 141033482039001036281030301015482061
E 5000
```

OUTPUT:

```
STARTING ADDRESS = 5000
```

```
ADDRESS = 5000 VALUE STORED = 14

ADDRESS = 5001 VALUE STORED = 10

ADDRESS = 5002 VALUE STORED = 33

ADDRESS = 5003 VALUE STORED = 48


ADDRESS = 5004 VALUE STORED = 20

ADDRESS = 5005 VALUE STORED = 39

ADDRESS = 5006 VALUE STORED =  0

ADDRESS = 5007 VALUE STORED =  10

ADDRESS = 5008 VALUE STORED =  36

ADDRESS = 5009 VALUE STORED = 28

ADDRESS = 5010 VALUE STORED = 10

ADDRESS = 5011 VALUE STORED = 30

ADDRESS = 5012 VALUE STORED = 30

ADDRESS = 5013 VALUE STORED = 10

ADDRESS = 5014 VALUE STORED = 15

ADDRESS = 5015 VALUE STORED = 48

ADDRESS = 5016 VALUE STORED = 20

ADDRESS = 5017 VALUE STORED = 61
```

Result: Thus we write a program to Implement a Absolute Loader