Computer Engineering Department
Compiler Design (3170701)

Practical List

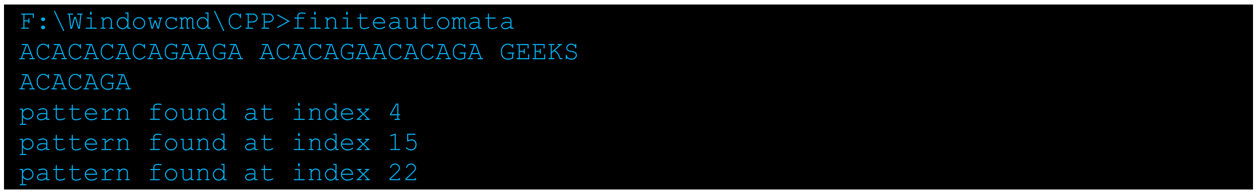
Sr. No.	Topic	CO covered	Date	Page
1.	Implementation of Finite Automata and String Validation.	CO1	28/6	1
2.	Introduction to Lex Tool.	CO1	5/7	5
3.	Implement following Programs Using Lex. (a) Generate Histogram of words (b) Ceasor Cypher (c) Extract single and multiline comments from C Program	CO1	12/7	9
4.	Implement following Programs Using Lex. (a) Convert Roman to Decimal (b) Check weather given statement is compound or simple (c) Extract html tags from .html file	CO1	26/7	12
5.	Introduction to YACC and generate Calculator Program.	CO2	2/8	16
6.	Implement a C program for constructing LL (1) parsing.	CO2	9/8	21
7.	Implement a C program to implement operator precedence parsing.	CO2	23/8	30
8.	Generate 3-tuple intermediate code for given infix expression.	CO3	6/9	34
9.	Extract Predecessor and Successor from given Control Flow Graph.	CO3	20/9	38
10.	Study of Learning Basic Block Scheduling Heuristics from optimal data	CO4	27/9	43

Practical – 1

AIM: Implementation of Finite Automata and String Validation

```
#include<iostream>
#include<set>
#include<cstring>
#include<iomanip>
using namespace std;
#define NO_OF_CHARS 256

/* This function builds the TF table
which represents Finite Automata for a
given pattern */
void computeTransFun(string pat, int M, int TF[][NO_OF_CHARS])
{
	int i, lps = 0, x;

	// Fill entries in first row
	for (x = 0; x < NO_OF_CHARS; x++)
		TF[0][x] = 0;
	TF[0][(int)pat[0]] = 1;

	// Fill entries in other rows
	for (i = 1; i <= M; i++) {
		// Copy values from row at index lps
		for (x = 0; x < NO_OF_CHARS; x++)
			TF[i][x] = TF[lps][x];

		// Update the entry corresponding to this character
		TF[i][(int)pat[i]] = i + 1;

		// Update lps for next row to be filled
		if (i < M)
			lps = TF[lps][(int)pat[i]];
	}
}

void printTF(string pat,int M, int TF[][NO_OF_CHARS]){
    set<char> ch;
    for(int i=0;i<M;++i){
        ch.insert(pat[i]);
    }
    cout<<setw(8)<<"";
    for(auto itr = ch.begin(); itr != ch.end(); itr++)
        cout<<*itr<<" ";
    cout<<"\n";
    for(int i=0;i<=M;++i){
        cout<<"q"<<i<<"| ";
        for(auto itr = ch.begin(); itr != ch.end(); itr++)
            cout<<TF[i][(int)*itr]<<" ";
        cout<<"\n";
    }
}
```

```
/* Prints all occurrences of pat in txt */
void search(string pat,string txt)
{
	int M = pat.length();
	int N = txt.length();

	int TF[M + 1][NO_OF_CHARS];

	computeTransFun(pat, M, TF);
    printTF(pat,M,TF);
	// process text over FA.
	int i, j = 0;
	for (i = 0; i < N; i++) {
		j = TF[j][(int)txt[i]];
		if (j == M) {
			cout << "pattern found at index " << i - M + 1 << endl;
		}
	}
}

/* Driver code */
int main()
{
	string txt,pat;
	// for ex.
	/*
	    txt = "ACACACACAGAAGA ACACAGAACACAGA GEEKS";
	    pat = "ACACAGA";
	    Automata
	                A C G
	          q0| 1 0 0  // 0 represents q0, 1 represents q1 and so on..
	          q1| 1 2 0
	          q2| 3 0 0
	          q3| 1 4 0
	          q4| 5 0 0
	          q5| 1 4 6
	          q6| 7 0 0
	          q7| 1 2 0
	  txt = "ACACACACAGAAGA ACACAGAACACAGA GEEKS";
	                  ^           ^      ^
	                  4           15     22
	*/
	getline(cin, txt);
	getline(cin, pat);
	search(pat, txt);
	return 0;
}
```



```
F:\Windowcmd\CPP>finiteautomata
ACACACACAGAAGA ACACAGAACACAGA GEEKS
ACACAGA
pattern found at index 4
pattern found at index 15
pattern found at index 22
```

STEP: 0 Pat = ACACAGA, Lps = 0;
Fill entry in first row with 0, TF[0][A]=1.
Lps = 0;

TF	…	A	B	C	D	E	F	G	...
Q0	0	1	0	0	0	0	0	0	0
Q1									
Q2									
Q3									
Q4									
Q5									
Q6									
Q7									

STEP: 1 Pat = ACACAGA, Lps = 0; Copy values from row at index Lps and update the entry to this character TF[1][C]=i+1, Lps<=TF[Lps][C]=0;

TF	…	A	B	C	D	E	F	G	...
Q0	0	1	0	0	0	0	0	0	0
Q1	0	1	0	2	0	0	0	0	0
Q2									
Q3									
Q4									
Q5									
Q6									
Q7									

STEP: 2 Pat = ACACAGA, Lps = 0; **STEP: 3** Pat = ACACAGA, Lps = 1;
Copy values from row at index Lps and update the entry corresponding to this character
TF[2][A]=i+1, Lps<=TF[Lps][A]=1; TF[3][C]=i+1, Lps<=TF[Lps][C]=2;

TF	…	A	B	C	D	E	F	G	...
Q0	0	1	0	0	0	0	0	0	0
Q1	0	1	0	2	0	0	0	0	0
Q2	0	3	0	0	0	0	0	0	0
Q3									
Q4									
Q5									
Q6									
Q7									

TF	…	A	B	C	D	E	F	G	...
Q0	0	1	0	0	0	0	0	0	0
Q1	0	1	0	2	0	0	0	0	0
Q2	0	3	0	0	0	0	0	0	0
Q3	0	1	0	4	0	0	0	0	0
Q4									
Q5									
Q6									
Q7									

STEP: 4 Pat = ACACAGA, Lps = 2; **STEP: 5** Pat = ACACAGA, Lps = 3;
Copy values from row at index Lps and update the entry corresponding to this character
TF[2][A]=i+1, Lps<=TF[Lps][A]=3; TF[3][G]=i+1, Lps<=TF[Lps][G]=0;

TF	…	A	B	C	D	E	F	G	...
Q0	0	1	0	0	0	0	0	0	0
Q1	0	1	0	2	0	0	0	0	0
Q2	0	3	0	0	0	0	0	0	0
Q3	0	1	0	4	0	0	0	0	0
Q4	0	5	0	0	0	0	0	0	0
Q5									
Q6									
Q7									

TF	…	A	B	C	D	E	F	G	...
Q0	0	1	0	0	0	0	0	0	0
Q1	0	1	0	2	0	0	0	0	0
Q2	0	3	0	0	0	0	0	0	0
Q3	0	1	0	4	0	0	0	0	0
Q4	0	5	0	0	0	0	0	0	0
Q5	0	1	0	4	0	0	0	6	0
Q6									
Q7									

STEP: 6 Pat = ACACAGA, Lps = 0;
Copy values from row at index Lps and update the entry corresponding to this character
TF[2][A]=i+1, Lps<=TF[Lps][A]=1;

TF	…	A	B	C	D	E	F	G	...
Q0	0	1	0	0	0	0	0	0	0
Q1	0	1	0	2	0	0	0	0	0
Q2	0	3	0	0	0	0	0	0	0
Q3	0	1	0	4	0	0	0	0	0
Q4	0	5	0	0	0	0	0	0	0
Q5	0	1	0	4	0	0	0	6	0
Q6	0	7	0	0	0	0	0	0	0
Q7									

STEP: 7 Pat = ACACAGA_, Lps = 1;
TF[3][_]=i+1, Lps<=TF[Lps][_];

TF	…	A	B	C	D	E	F	G	...
Q0	0	1	0	0	0	0	0	0	0
Q1	0	1	0	2	0	0	0	0	0
Q2	0	3	0	0	0	0	0	0	0
Q3	0	1	0	4	0	0	0	0	0
Q4	0	5	0	0	0	0	0	0	0
Q5	0	1	0	4	0	0	0	6	0
Q6	0	7	0	0	0	0	0	0	0
Q7	0	1	0	2	0	0	0	0	0

Practical – 2

AIM: Introduction to Lex Tool.

Introduction:

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 1.

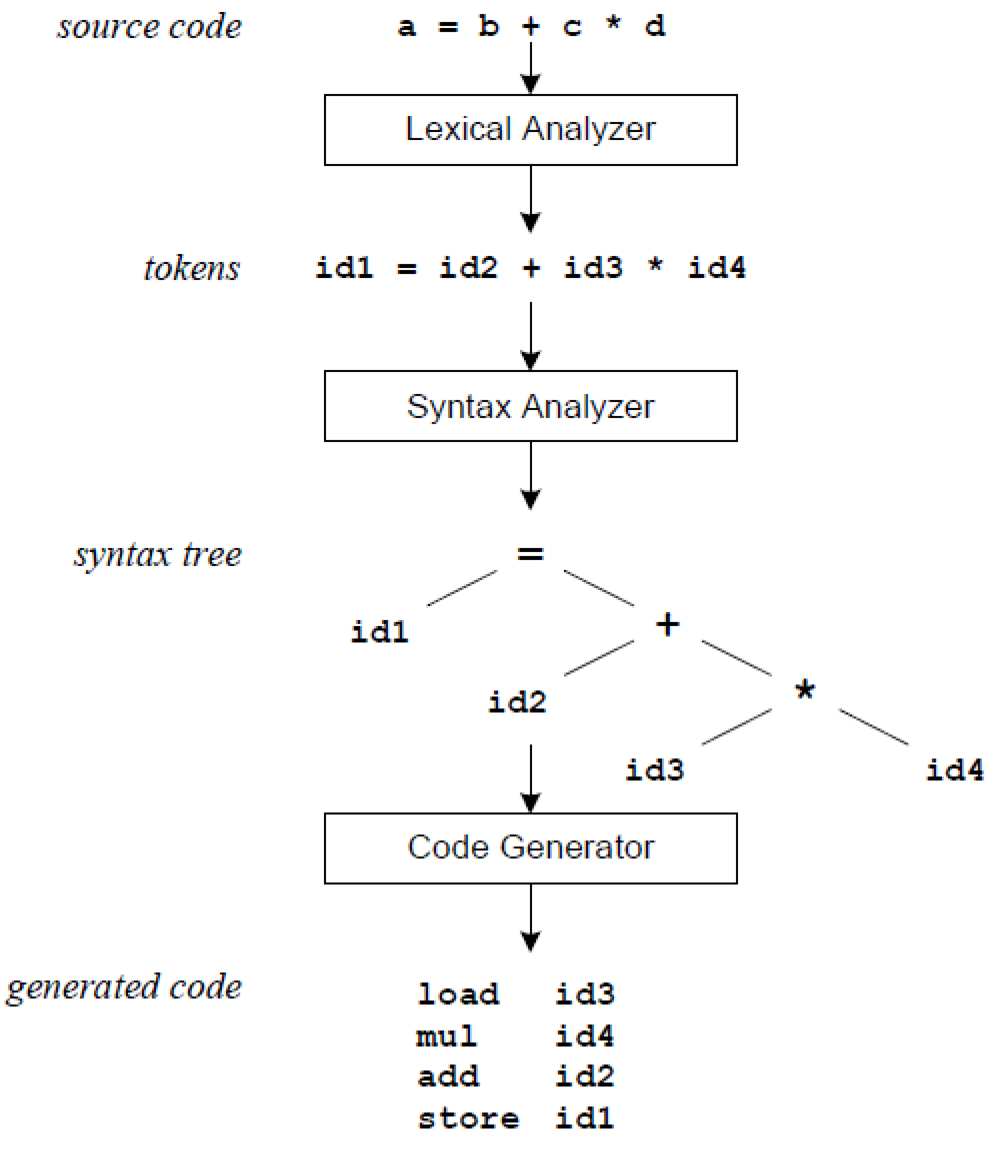


Figure 1: Compilation Sequence

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.

Pattern Matching:

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.

Any **regular expression** 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.

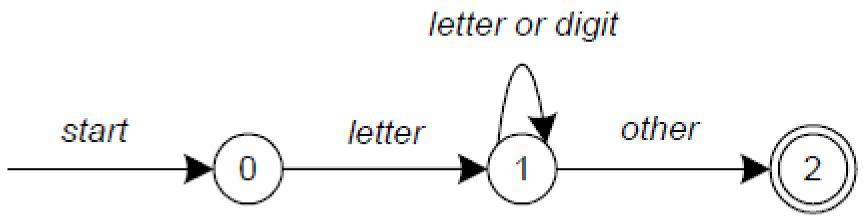


Figure 3: Finite State Automaton

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.

Limitations:

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 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

Input to Lex is divided into three sections, with %% dividing the sections. This is best illustrated by example.

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

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**, the main entry-point for lex. Some implementations of **lex include copies of main and yywrap** in a library, eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Example Program:

```
%{
        int counter = 0;
%}
letter   [a-zA-Z]

%%
{letter}+        {printf("a word\n"); counter++;}

%%
main() {
        yylex();
        printf("There are total %d words\n", counter);
}
```

- To run Lex on a source file, type
  lex scanner.l

  It produces a file named lex.yy.c which is a C program for the lexical analyzer.

- To compile lex.yy.c, type
  cc lex.yy.c –ll

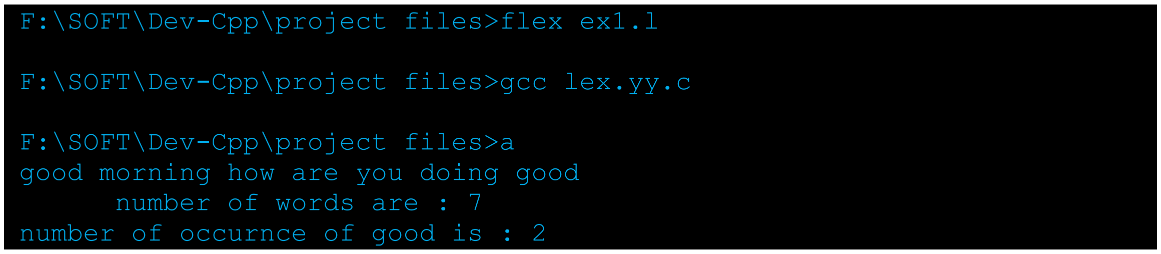
- To run the lexical analyzer program, type
  ./a.out < inputfile

Practical – 3

AIM: Implement following Programs Using Lex.

(a) Generate Histogram of words

```
/*lex program to count number of words*/
%{
  #include<stdio.h>
  #include<string.h>
  int count=0;
  char word[]="good";
  int gd=0;
%}

/* rules */
%%

([a-zA-Z]+) {count++;if(strcmp(yytext,word)==0)gd++;}
\n {return 0;}

%%

int yywrap(){
    return 1;
    }

int main(){
    yylex();
    printf("number of words are : %d\n",count);
    printf("number of occurnce of good is : %d\n",gd);
    return 0;
}
```

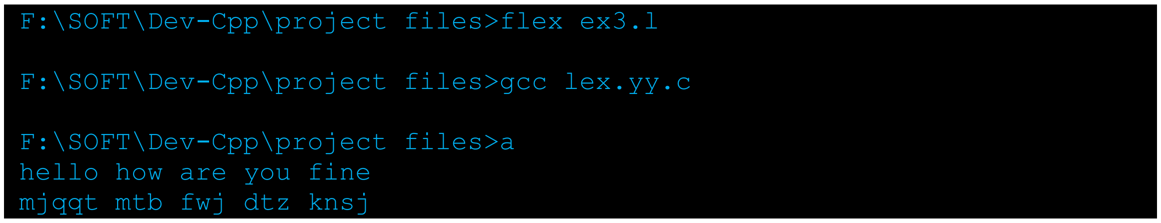


```
F:\SOFT\Dev-Cpp\project files>flex ex1.l

F:\SOFT\Dev-Cpp\project files>gcc lex.yy.c

F:\SOFT\Dev-Cpp\project files>a
good morning how are you doing good
      number of words are : 7
number of occurnce of good is : 2
```

Fig - 1

(b) Ceasor Cypher

```
/*lex program to demonstrate Ceasor Cypher encryption*/
%{
  char ch;
%}

%%
[a-z] { ch = yytext[0];
        ch = ch +5;
        if(ch>'z')ch=ch-('z'+1-'a');
        printf("%c",ch);
      }
[A-Z] { ch = yytext[0];
        ch = ch +5;
        if(ch>'Z')ch=ch-('Z'+1-'A');
        printf("%c",ch);
       }
\n {return 0;}

%%

int yywrap(){
    return 1;
}

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



```
F:\SOFT\Dev-Cpp\project files>flex ex3.l

F:\SOFT\Dev-Cpp\project files>gcc lex.yy.c

F:\SOFT\Dev-Cpp\project files>a
hello how are you fine
mjqqt mtb fwj dtz knsj
```

Fig - 2

(c) Extract single and multiline comments from C Program

```
/* Lex Program to remove comments from C program */

%{

%}

/*Rule Section*/
%%
\/\/(.*) ;

\/\*((.|\n)*)\*\/ ;
%%

int yywrap(){
    return 1;
}

int main(int k,char **argcv){
    yyin=fopen(argcv[1],"r");
    yyout=fopen("out.c","w");

    yylex();
    return 0;
}
```

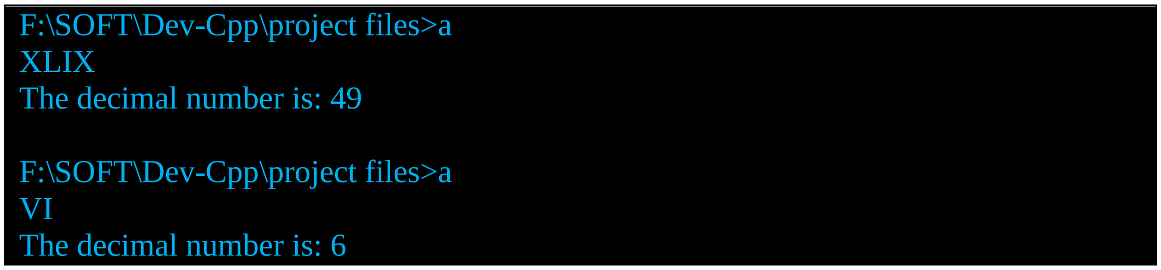
```
F:\SOFT\Dev-Cpp\project files>type test.txt
//testing
#include<stdio.h>
int main(){
/*    multiline
      comment continue..  */
return 0;
}
F:\SOFT\Dev-Cpp\project files>flex ex4.l

F:\SOFT\Dev-Cpp\project files>gcc lex.yy.c

F:\SOFT\Dev-Cpp\project files>a test.txt

F:\SOFT\Dev-Cpp\project files>type out.c

#include<stdio.h>
int main(){


return 0;
}
```

Practical – 4

AIM: Implement following Programs Using Lex

(a) Convert Roman to Decimal

```
/*lex program to Convert Roman to Decimal */
%{
  int total = 0;
%}

%%

I      total = total + 1;
IV     total = total + 4;
V      total = total + 5;
IX     total = total + 9;
X      total = total + 10;
XL     total = total + 40;
L      total = total + 50;
XC     total = total + 90;
C      total = total + 100;
CD     total = total + 400;
D      total = total + 500;
CM     total = total + 900;
M      total = total + 1000;

\n return total;

%%

int yywrap(){
    return 1;
}

int main(){
    int dec;
    dec = yylex();

    printf("The decimal number is: %d \n",dec);
    return 0;
}
```

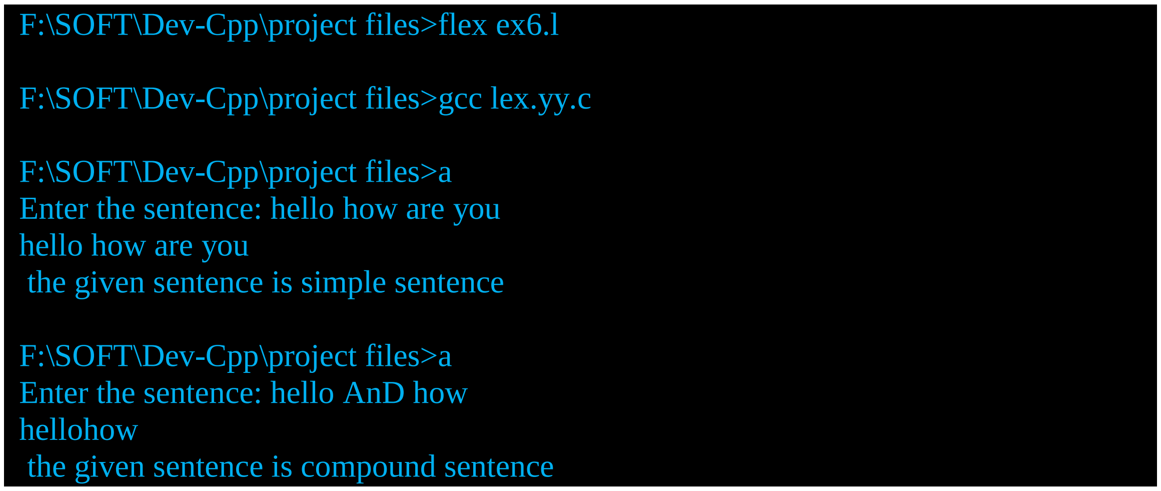


```
F:\SOFT\Dev-Cpp\project files>a
XLIX
The decimal number is: 49

F:\SOFT\Dev-Cpp\project files>a
VI
The decimal number is: 6
```

Fig - 1

(b) Check weather given statement is compound or simple

```
/* lex program to given statement is compound or simple */
%{
#include<stdio.h>
int is_simple=1;
%}

%%
[ \t]+[aA][nN][dD][ \t]+ {is_simple=0;}
[ \t]+[bB][uU][tT][ \t]+ {is_simple=0;}
[ \t]+[oO][rR][ \t]+ {is_simple=0;}
\n {return 0;}
%%

int yywrap(){
return 1;
}

int main(){
  printf("Enter the sentence: ");
  yylex();

  if(is_simple==1)
    printf("\n the given sentence is simple sentence\n");
  else
    printf("\n the given sentence is compound sentence\n");

  return 0;
}
```

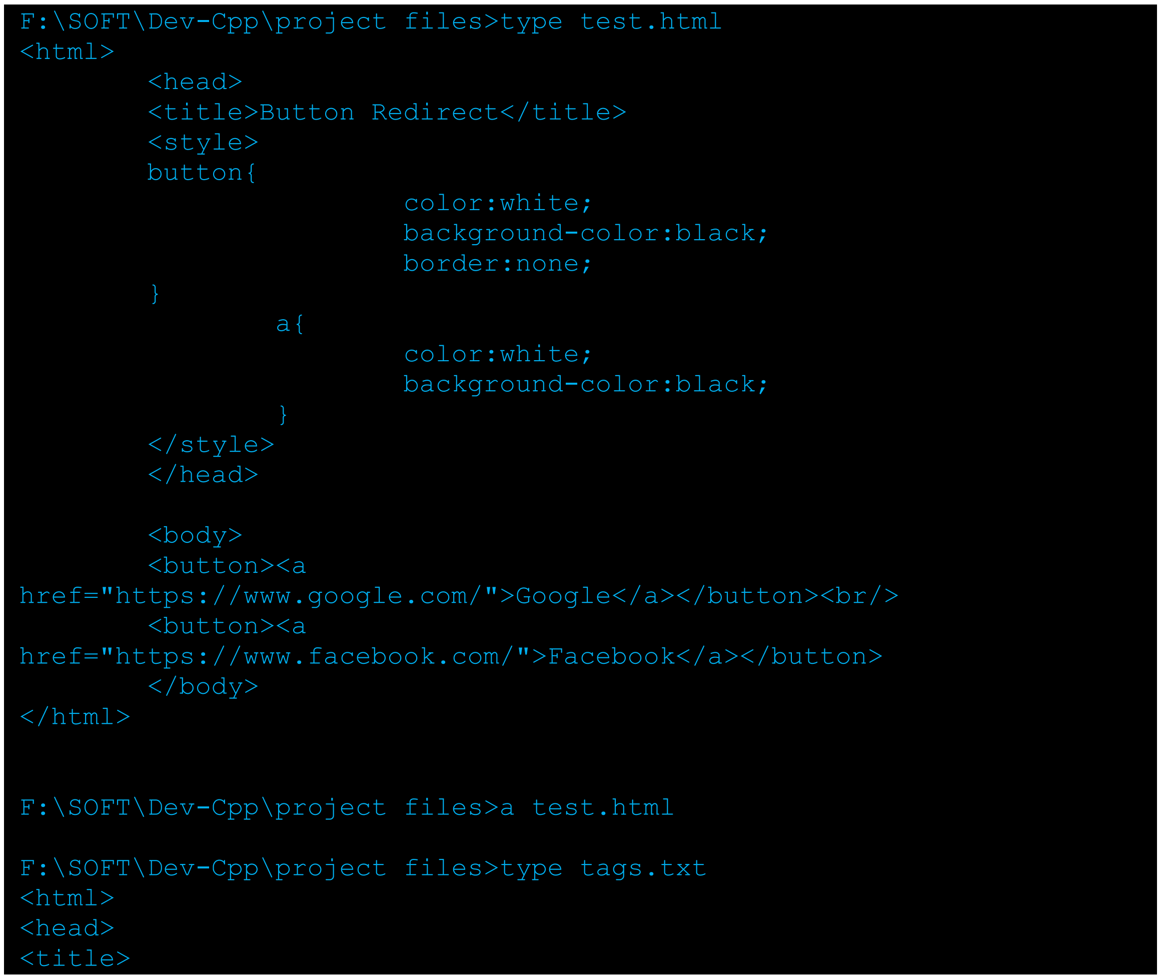


```
F:\SOFT\Dev-Cpp\project files>flex ex6.l

F:\SOFT\Dev-Cpp\project files>gcc lex.yy.c

F:\SOFT\Dev-Cpp\project files>a
Enter the sentence: hello how are you
hello how are you
 the given sentence is simple sentence

F:\SOFT\Dev-Cpp\project files>a
Enter the sentence: hello AnD how
hellohow
 the given sentence is compound sentence
```

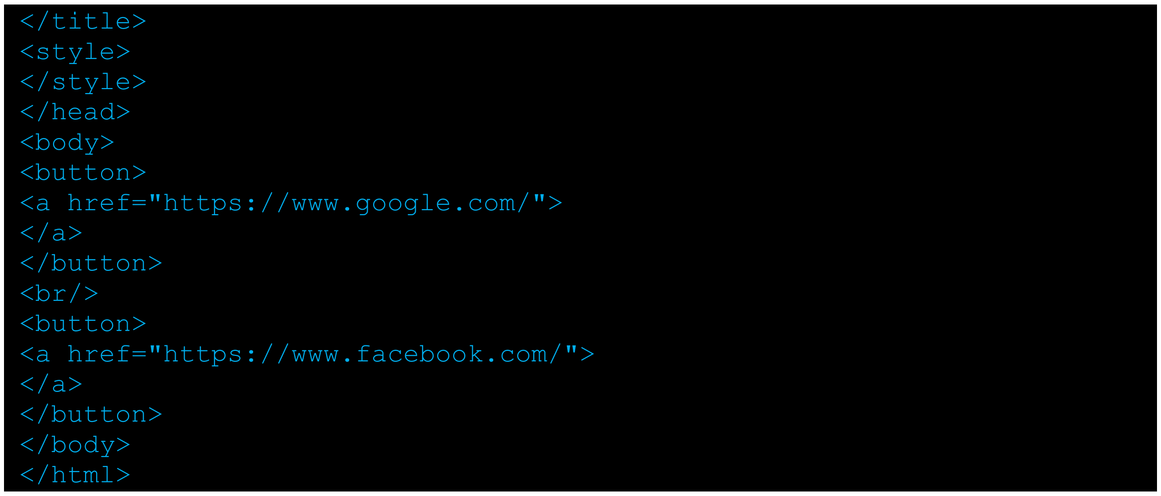
Fig - 2

(c) Extract html tags from .html file

```
/*lex program to extract html tags */
%{
  #include<stdio.h>
%}

%%
"<"[^<]*">" { /*printf("%s\n",yytext);*/ fprintf(yyout,"%s\n",yytext);}
. ;
\n ;
%%

int yywrap(){
    return 1;
}

int main(char **argcv){
    yyin=fopen(argcv[1],"r");
    yyout=fopen("tags.txt","w");
    yylex();
    return 0;
}
```



```
F:\SOFT\Dev-Cpp\project files>type test.html
<html>
        <head>
        <title>Button Redirect</title>
        <style>
        button{
                        color:white;
                        background-color:black;
                        border:none;
        }
                a{
                        color:white;
                        background-color:black;
                }
        </style>
        </head>

        <body>
        <button><a
href="https://www.google.com/">Google</a></button><br/>
        <button><a
href="https://www.facebook.com/">Facebook</a></button>
        </body>
</html>


F:\SOFT\Dev-Cpp\project files>a test.html

F:\SOFT\Dev-Cpp\project files>type tags.txt
<html>
<head>
<title>
```



```
</title>
<style>
</style>
</head>
<body>
<button>
<a href="https://www.google.com/">
</a>
</button>
<br/>
<button>
<a href="https://www.facebook.com/">
</a>
</button>
</body>
</html>
```

Fig - 3

Practical – 5

AIM: Introduction to YACC and generate Calculator Program.

Grammars for yacc are described using a variant of **Backus Naur Form (BNF)**. This technique was pioneered by John Backus and Peter Naur, and 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, replacing the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression, we actually 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

```
... 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.

When we run yacc, it generates a parser in file y.tab.c, and also creates an include file, y.tab.h:

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**, and returns the 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.

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, as 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, and represents 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.

By utilizing 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. Thus, the parse and value stacks remain synchronized.

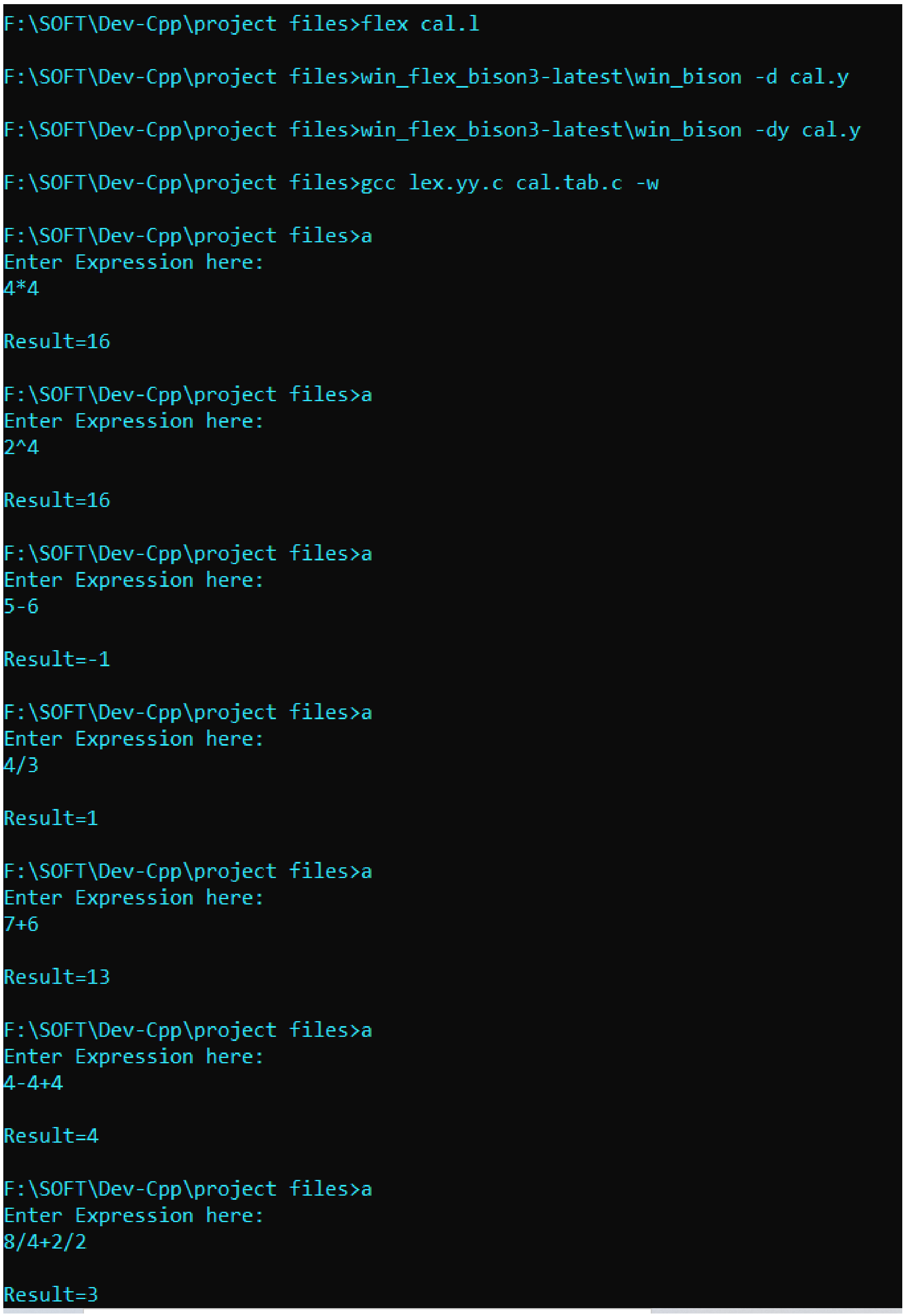
cal.y

```
%{
        /* Definition section */
        #include<stdio.h>
        #include <math.h>
%}

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

%right '^'     /* exponentiation */

/* Rule Section */
%%

calculation: E {printf("\nResult=%d\n", $$);};
E:E'+'E {$$=$1+$3;}

|E'-'E {$$=$1-$3;}

|E'*'E {$$=$1*$3;}
```

```
|E'/'E {$$=$1/$3;}

|E'%'E {$$=$1%$3;}

|E'^'E {$$=pow($1,$3);}

|'('E')' {$$=$2;}

| NUMBER {$$=$1;}

;

%%

//driver code
int main(){
 printf("Enter Expression here: \n");
 yyparse();
 return 0;
}

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

cal.l

```
%{
    #include<stdlib.h>
    #include "y.tab.h"

    int yylval;
%}

%%

[0-9]+ {
        yylval = atoi(yytext);
        return NUMBER;
        }
[\t] ;
[\n] return 0;
. return yytext[0];

%%

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



```
F:\SOFT\Dev-Cpp\project files>flex cal.l

F:\SOFT\Dev-Cpp\project files>win_flex_bison3-latest\win_bison -d cal.y

F:\SOFT\Dev-Cpp\project files>win_flex_bison3-latest\win_bison -dy cal.y

F:\SOFT\Dev-Cpp\project files>gcc lex.yy.c cal.tab.c -w

F:\SOFT\Dev-Cpp\project files>a
Enter Expression here:
4*4

Result=16

F:\SOFT\Dev-Cpp\project files>a
Enter Expression here:
2^4

Result=16

F:\SOFT\Dev-Cpp\project files>a
Enter Expression here:
5-6

Result=-1

F:\SOFT\Dev-Cpp\project files>a
Enter Expression here:
4/3

Result=1

F:\SOFT\Dev-Cpp\project files>a
Enter Expression here:
7+6

Result=13

F:\SOFT\Dev-Cpp\project files>a
Enter Expression here:
4-4+4

Result=4

F:\SOFT\Dev-Cpp\project files>a
Enter Expression here:
8/4+2/2

Result=3
```

Fig 5.1 – yacc calculator

Practical – 6

AIM: Implement a C program for constructing LL(1) parsing.

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

void followfirst(char , int , int);
void findfirst(char , int , int);
void follow(char c);

int count,n=0;
char calc_first[10][100];
char calc_follow[10][100];
int m=0;
char production[10][10], first[10];
char f[10];
int k;
char ck;
int e;

int main(int argc,char **argv)
{
	int jm=0;
	int km=0;
	int i,choice;
	char c,ch;
	printf("How many productions ? :");
	scanf("%d",&count);
	printf("\nEnter %d productions in form A=B where A and B are grammar 
symbols :\n\n",count);
	for(i=0;i<count;i++)
	{
		scanf("%s%c",production[i],&ch);
	}
	int kay;
	char done[count];
	int ptr = -1;
	for(k=0;k<count;k++){
		for(kay=0;kay<100;kay++){
			calc_first[k][kay] = '!';
		}
	}
	int point1 = 0,point2,xxx;
	for(k=0;k<count;k++)
	{
		c=production[k][0];
		point2 = 0;
		xxx = 0;
		for(kay = 0; kay <= ptr; kay++)
			if(c == done[kay])
				xxx = 1;
		if (xxx == 1)
			continue;
```

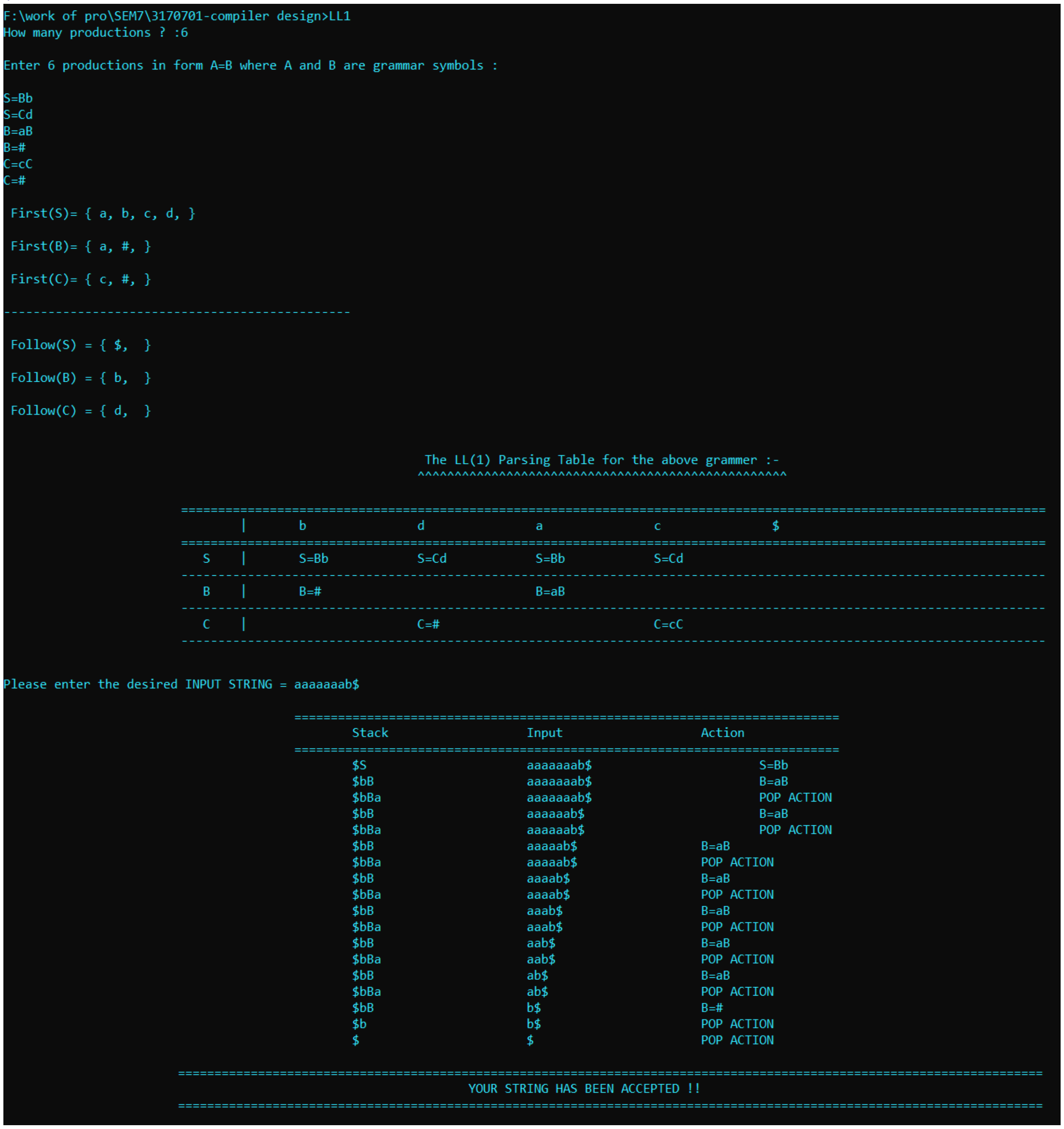
```
            findfirst(c,0,0);
            ptr+=1;
            done[ptr] = c;
            printf("\n First(%c)= { ",c);
            calc_first[point1][point2++] = c;
            for(i=0+jm;i<n;i++){
                  int lark = 0,chk = 0;
                  for(lark=0;lark<point2;lark++){
                        if (first[i] == calc_first[point1][lark]){
                              chk = 1;
                              break;
                        }
                  }
                  if(chk == 0){
                        printf("%c, ",first[i]);
                        calc_first[point1][point2++] = first[i];
                  }
            }
            printf("}\n");
            jm=n;
            point1++;
      }
      printf("\n");
      printf("-----------------------------------------------\n\n");
      char donee[count];
      ptr = -1;
      for(k=0;k<count;k++){
            for(kay=0;kay<100;kay++){
                  calc_follow[k][kay] = '!';
            }
      }
      point1 = 0;
      int land = 0;
      for(e=0;e<count;e++)
      {
            ck=production[e][0];
            point2 = 0;
            xxx = 0;
            for(kay = 0; kay <= ptr; kay++)
                  if(ck == donee[kay])
                        xxx = 1;
            if (xxx == 1)
                  continue;
            land += 1;
            follow(ck);
            ptr+=1;
            donee[ptr] = ck;
            printf(" Follow(%c) = { ",ck);
            calc_follow[point1][point2++] = ck;
            for(i=0+km;i<m;i++){
                  int lark = 0,chk = 0;
                  for(lark=0;lark<point2;lark++){
                        if (f[i] == calc_follow[point1][lark]){
                              chk = 1;
                              break;
                        }
                  }
```

```
                if(chk == 0){
                        printf("%c, ",f[i]);
                        calc_follow[point1][point2++] = f[i];
                }
        }
        printf(" }\n\n");
        km=m;
        point1++;
    }
    char ter[10];
    for(k=0;k<10;k++){
        ter[k] = '!';
    }
    int ap,vp,sid = 0;
    for(k=0;k<count;k++){
        for(kay=0;kay<count;kay++){
            if(!isupper(production[k][kay]) && production[k][kay]!= '#'
&& production[k][kay] != '=' && production[k][kay] != '\0'){
                vp = 0;
                for(ap = 0;ap < sid; ap++){
                    if(production[k][kay] == ter[ap]){
                        vp = 1;
                        break;
                    }
                }
                if(vp == 0){
                    ter[sid] = production[k][kay];
                    sid ++;
                }
            }
        }
    }
    ter[sid] = '$';
    sid++;
    printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammer
:-");
    printf("\n\t\t\t\t\t\t\t^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^\n");
    printf("\n\t\t\t================================================
==============================================================\n");
    printf("\t\t\t\t|\t");
    for(ap = 0;ap < sid; ap++){
        printf("%c\t\t",ter[ap]);
    }
    printf("\n\t\t\t================================================
==============================================================\n");
    char first_prod[count][sid];
    for(ap=0;ap<count;ap++){
        int destiny = 0;
        k = 2;
        int ct = 0;
        char tem[100];
        while(production[ap][k] != '\0'){
            if(!isupper(production[ap][k])){
                tem[ct++] = production[ap][k];
                tem[ct++] = '_';
                tem[ct++] = '\0';
```

```
                                        k++;
                                        break;
                                }
                                else{
                                        int zap=0;
                                        int tuna = 0;
                                        for(zap=0;zap<count;zap++){
                                                if(calc_first[zap][0] == production[ap][k]){
                                                        for(tuna=1;tuna<100;tuna++){
                                                                if(calc_first[zap][tuna] != '!'){
                                                                        tem[ct++] =
calc_first[zap][tuna];
                                                                }
                                                                else
                                                                        break;
                                                        }
                                                break;
                                                }
                                        }
                                        tem[ct++] = '_';
                                }
                                k++;
                        }
                        int zap = 0,tuna;
                        for(tuna = 0;tuna<ct;tuna++){
                                if(tem[tuna] == '#'){
                                        zap = 1;
                                }
                                else if(tem[tuna] == '_'){
                                        if(zap == 1){
                                                zap = 0;
                                        }
                                        else
                                                break;
                                }
                                else{
                                        first_prod[ap][destiny++] = tem[tuna];
                                }
                        }
                }
                char table[land][sid+1];
                ptr = -1;
                for(ap = 0; ap < land ; ap++){
                        for(kay = 0; kay < (sid + 1) ; kay++){
                                table[ap][kay] = '!';
                        }
                }
                for(ap = 0; ap < count ; ap++){
                        ck = production[ap][0];
                        xxx = 0;
                        for(kay = 0; kay <= ptr; kay++)
                                if(ck == table[kay][0])
                                        xxx = 1;
                        if (xxx == 1)
                                continue;
                        else{
                                ptr = ptr + 1;
                                table[ptr][0] = ck;
                        }
                }
                for(ap = 0; ap < count ; ap++){
                        int tuna = 0;
                        while(first_prod[ap][tuna] != '\0'){
                                int to,ni=0;
                                for(to=0;to<sid;to++){
                                        if(first_prod[ap][tuna] == ter[to]){
                                                ni = 1;
                                        }
                                }
                                if(ni == 1){
                                        char xz = production[ap][0];
                                        int cz=0;
                                        while(table[cz][0] != xz){
                                                cz = cz + 1;
                                        }
                                        int vz=0;
                                        while(ter[vz] != first_prod[ap][tuna]){
                                                vz = vz + 1;
                                        }
                                        table[cz][vz+1] = (char)(ap + 65);
                                }
                                tuna++;
                        }
                }
                for(k=0;k<sid;k++){
                        for(kay=0;kay<100;kay++){
                                if(calc_first[k][kay] == '!'){
                                        break;
                                }
                                else if(calc_first[k][kay] == '#'){
                                        int fz = 1;
                                        while(calc_follow[k][fz] != '!'){
                                                char xz = production[k][0];
                                                int cz=0;
                                                while(table[cz][0] != xz){
                                                        cz = cz + 1;
                                                }
                                                int vz=0;
                                                while(ter[vz] != calc_follow[k][fz]){
                                                        vz = vz + 1;
                                                }
                                                table[k][vz+1] = '#';
                                                fz++;
                                        }
                                        break;
                                }
                        }
                }
                for(ap = 0; ap < land ; ap++){
                        printf("\t\t\t   %c\t|\t",table[ap][0]);
                        for(kay = 1; kay < (sid + 1) ; kay++){
                                if(table[ap][kay] == '!')
                                        printf("\t\t");
                                else if(table[ap][kay] == '#')
```

```
                                printf("%c=#\t\t",table[ap][0]);
                        else{
                                int mum = (int)(table[ap][kay]);
                                mum -= 65;
                                printf("%s\t\t",production[mum]);
                        }
                }
                printf("\n");
                printf("\t\t\t---------------------------------------------------
-------------------------------------------------------------------");
                printf("\n");
        }
        int j;
        printf("\n\nPlease enter the desired INPUT STRING = ");
        char input[100];
        scanf("%s%c",input,&ch);
        printf("\n\t\t\t\t\t\t=============================================
========================\n");
        printf("\t\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction");
        printf("\n\t\t\t\t\t\t=============================================
========================\n");
        int i_ptr = 0,s_ptr = 1;
        char stack[100];
        stack[0] = '$';
        stack[1] = table[0][0];
        while(s_ptr != -1){
                printf("\t\t\t\t\t\t");
                int vamp = 0;
                for(vamp=0;vamp<=s_ptr;vamp++){
                        printf("%c",stack[vamp]);
                }
                printf("\t\t\t");
                vamp = i_ptr;
                while(input[vamp] != '\0'){
                        printf("%c",input[vamp]);
                        vamp++;
                }
                printf("\t\t\t");
                char her = input[i_ptr];
                char him = stack[s_ptr];
                s_ptr--;
                if(!isupper(him)){
                        if(her == him){
                                i_ptr++;
                                printf("POP ACTION\n");
                        }
                        else{
                                printf("\nString Not Accepted by LL(1) Parser !!\n");
                                exit(0);
                        }
                }
                else{
                        for(i=0;i<sid;i++){
                                if(ter[i] == her)
                                        break;
                        }
                        char produ[100];
                        for(j=0;j<land;j++){
                                if(him == table[j][0]){
                                        if (table[j][i+1] == '#'){
                                                printf("%c=#\n",table[j][0]);
                                                produ[0] = '#';
                                                produ[1] = '\0';
                                        }
                                        else if(table[j][i+1] != '!'){
                                                int mum = (int)(table[j][i+1]);
                                                mum -= 65;
                                                strcpy(produ,production[mum]);
                                                printf("%s\n",produ);
                                        }
                                        else{
                                                printf("\nString Not Accepted by LL(1) 
Parser !!\n");
                                                exit(0);
                                        }
                                }
                        }
                        int le = strlen(produ);
                        le = le - 1;
                        if(le == 0){
                                continue;
                        }
                        for(j=le;j>=2;j--){
                                s_ptr++;
                                stack[s_ptr] = produ[j];
                        }
                }
        }
        printf("\n\t\t\t=====================================================
=================================================================\n");
        if (input[i_ptr] == '\0'){
                printf("\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");
        }
        else
                printf("\n\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");
        printf("\t\t\t=========================================================
===============================================================\n");
}
void follow(char c)
{
        int i ,j;
        if(production[0][0]==c){
                f[m++]='$';
        }
        for(i=0;i<10;i++)
        {
                for(j=2;j<10;j++)
                {
                        if(production[i][j]==c)
                        {
                        if(production[i][j+1]!='\0'){
                                        followfirst(production[i][j+1],i,(j+2));
                                }
                        if(production[i][j+1]=='\0'&&c!=production[i][0]){
```

```
                        follow(production[i][0]);
                        }
                    }
                }
            }
        }
void findfirst(char c ,int q1 , int q2){
        int j;
        if(!(isupper(c))){
                first[n++]=c;
        }
        for(j=0;j<count;j++)
        {
                if(production[j][0]==c)
                {
                        if(production[j][2]=='#'){
                                if(production[q1][q2] == '\0')
                                        first[n++]='#';
                                else if(production[q1][q2] != '\0' && (q1 != 0 || q2
!= 0))
                                {
                                        findfirst(production[q1][q2], q1, (q2+1));
                                }
                                else
                                        first[n++]='#';
                        }
                        else if(!isupper(production[j][2])){
                                first[n++]=production[j][2];
                        }
                        else {
                                findfirst(production[j][2], j, 3);
                        }
                }
        }
}
void followfirst(char c, int c1 , int c2){
     int k;
     if(!(isupper(c)))
                f[m++]=c;
       else{
                int i=0,j=1;
                for(i=0;i<count;i++)
                {
                        if(calc_first[i][0] == c)
                                break;
                }
                while(calc_first[i][j] != '!')
                {
                        if(calc_first[i][j] != '#'){
                                f[m++] = calc_first[i][j];
                        }
                        else{
                                if(production[c1][c2] == '\0'){
                                        follow(production[c1][0]);
                                }
                                else{
                                        followfirst(production[c1][c2],c1,c2+1);
                                        }
                                }
                                j++;
                        }
                }
        }
```



```
F:\work of pro\SEM7\3170701-compiler design>LL1
How many productions ? :6

Enter 6 productions in form A=B where A and B are grammar symbols :

S=Bb
S=Cd
B=aB
B=#
C=cC
C=#

 First(S)= { a, b, c, d, }

 First(B)= { a, #, }

 First(C)= { c, #, }

-----------------------------------------------

 Follow(S) = { $,  }

 Follow(B) = { b,  }

 Follow(C) = { d,  }


                                                         The LL(1) Parsing Table for the above grammer :-
                                                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

                        =====================================================================================================================
                                |       b               d               a               c               $
                        =====================================================================================================================
                           S    |       S=Bb            S=Cd            S=Bb            S=Cd
                        ---------------------------------------------------------------------------------------------------------------------
                           B    |       B=#                             B=aB
                        ---------------------------------------------------------------------------------------------------------------------
                           C    |                       C=#                             C=cC
                        ---------------------------------------------------------------------------------------------------------------------


Please enter the desired INPUT STRING = aaaaaaab$

                                       ==========================================================================
                                               Stack                   Input                  Action
                                       ==========================================================================
                                               $S                      aaaaaaab$                      S=Bb
                                               $bB                     aaaaaaab$                      B=aB
                                               $bBa                    aaaaaaab$                      POP ACTION
                                               $bB                     aaaaaab$                       B=aB
                                               $bBa                    aaaaaab$                       POP ACTION
                                               $bB                     aaaaab$                B=aB
                                               $bBa                    aaaaab$                POP ACTION
                                               $bB                     aaaab$                 B=aB
                                               $bBa                    aaaab$                 POP ACTION
                                               $bB                     aaab$                  B=aB
                                               $bBa                    aaab$                  POP ACTION
                                               $bB                     aab$                   B=aB
                                               $bBa                    aab$                   POP ACTION
                                               $bB                     ab$                    B=aB
                                               $bBa                    ab$                    POP ACTION
                                               $bB                     b$                     B=#
                                               $b                      b$                     POP ACTION
                                               $                       $                      POP ACTION

                       =====================================================================================================================
                                                               YOUR STRING HAS BEEN ACCEPTED !!
                       =====================================================================================================================
```

Fig 6.1 LL1 parser

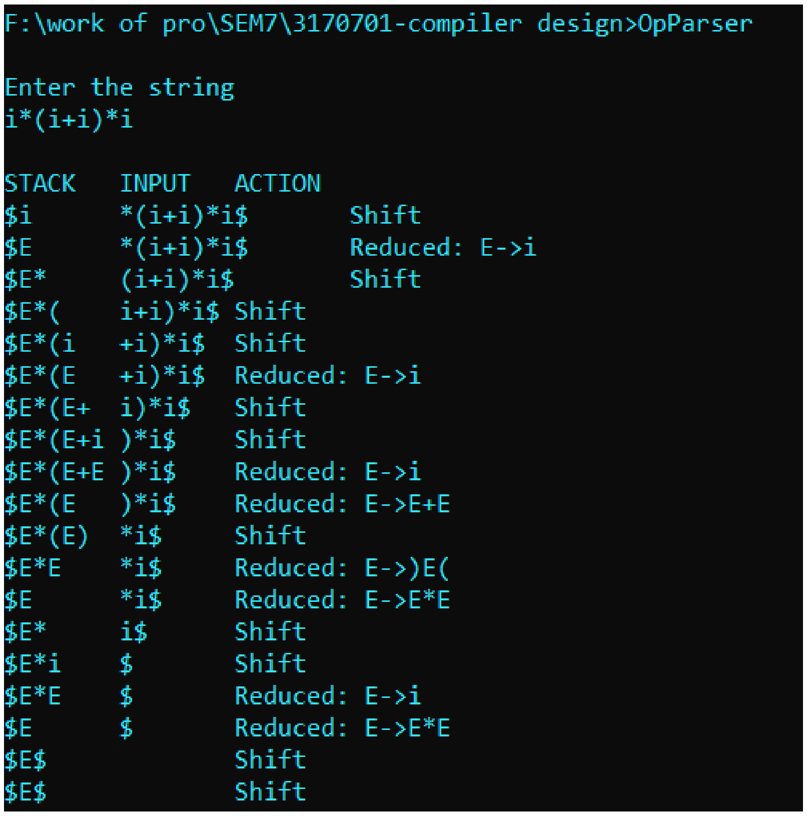
Practical – 7

AIM: Implement a C program to implement operator precedence parsing.

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

char *input;
int i=0;
char lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"};
//(E) becomes )E( when pushed to stack

int top=0,l;
char prec[9][9]={

                            /*input*/

            /*stack    +    -   *   /   ^   i   (   )   $  */

            /*  + */  '>', '>','<','<','<','<','<','>','>',

            /*  - */  '>', '>','<','<','<','<','<','>','>',

            /*  * */  '>', '>','>','>','<','<','<','>','>',

            /*  / */  '>', '>','>','>','<','<','<','>','>',

            /*  ^ */  '>', '>','>','>','<','<','<','>','>',

            /*  i */  '>', '>','>','>','>','e','e','>','>',

            /*  ( */  '<', '<','<','<','<','<','<','>','e',

            /*  ) */  '>', '>','>','>','>','e','e','>','>',

            /*  $ */  '<', '<','<','<','<','<','<','<','>',

                };

int getindex(char c)
{
switch(c)
    {
    case '+':return 0;
    case '-':return 1;
    case '*':return 2;
    case '/':return 3;
    case '^':return 4;
    case 'i':return 5;
    case '(':return 6;
    case ')':return 7;
    case '$':return 8;
    }
}
```

```c
int shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
}


int reduce()
{
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
    {
    len=strlen(handles[i]);
    if(stack[top]==handles[i][0]&&top+1>=len)
        {
        found=1;
        for(t=0;t<len;t++)
            {
            if(stack[top-t]!=handles[i][t])
                {
                found=0;
                break;
                }
            }
        if(found==1)
            {
            stack[top-t+1]='E';
            top=top-t+1;
            strcpy(lasthandle,handles[i]);
            stack[top+1]='\0';
            return 1;//successful reduction
            }
        }
   }
return 0;
}



void dispstack()
{
int j;
for(j=0;j<=top;j++)
    printf("%c",stack[j]);
}



void dispinput()
{
int j;
for(j=i;j<l;j++)
    printf("%c",*(input+j));
}
```

```
void main()
{
int j;

input=(char*)malloc(50*sizeof(char));
printf("\nEnter the string\n");
scanf("%s",input);
input=strcat(input,"$");
l=strlen(input);
strcpy(stack,"$");
printf("\nSTACK\tINPUT\tACTION");
while(i<=l)
      {
      shift();
      printf("\n");
      dispstack();
      printf("\t");
      dispinput();
      printf("\tShift");
      if(prec[getindex(stack[top])][getindex(input[i])]=='>')
            {
            while(reduce())
                  {
                  printf("\n");
                  dispstack();
                  printf("\t");
                  dispinput();
                  printf("\tReduced: E->%s",lasthandle);
                  }
            }
      }

if(strcmp(stack,"$E$")==0)
    printf("\nAccepted;");
else
    printf("\nNot Accepted;");
}
```



```
F:\work of pro\SEM7\3170701-compiler design>OpParser

Enter the string
i*(i+i)*i

STACK   INPUT   ACTION
$i      *(i+i)*i$       Shift
$E      *(i+i)*i$       Reduced: E->i
$E*     (i+i)*i$        Shift
$E*(    i+i)*i$ Shift
$E*(i   +i)*i$  Shift
$E*(E   +i)*i$  Reduced: E->i
$E*(E+  i)*i$   Shift
$E*(E+i )*i$    Shift
$E*(E+E )*i$    Reduced: E->i
$E*(E   )*i$    Reduced: E->E+E
$E*(E)  *i$     Shift
$E*E    *i$     Reduced: E->)E(
$E      *i$     Reduced: E->E*E
$E*     i$      Shift
$E*i    $       Shift
$E*E    $       Reduced: E->i
$E      $       Reduced: E->E*E
$E$             Shift
$E$             Shift
```

Fig 7.1 – Op Parser

Practical – 8

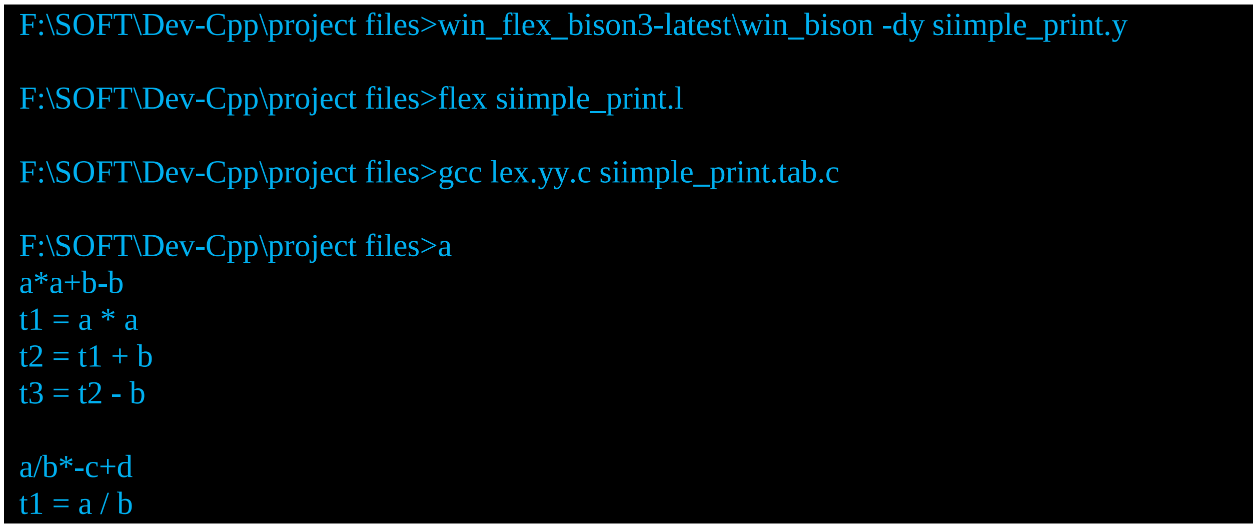
AIM: Generate 3-tuple intermediate code for given infix expression.

simple_print.l

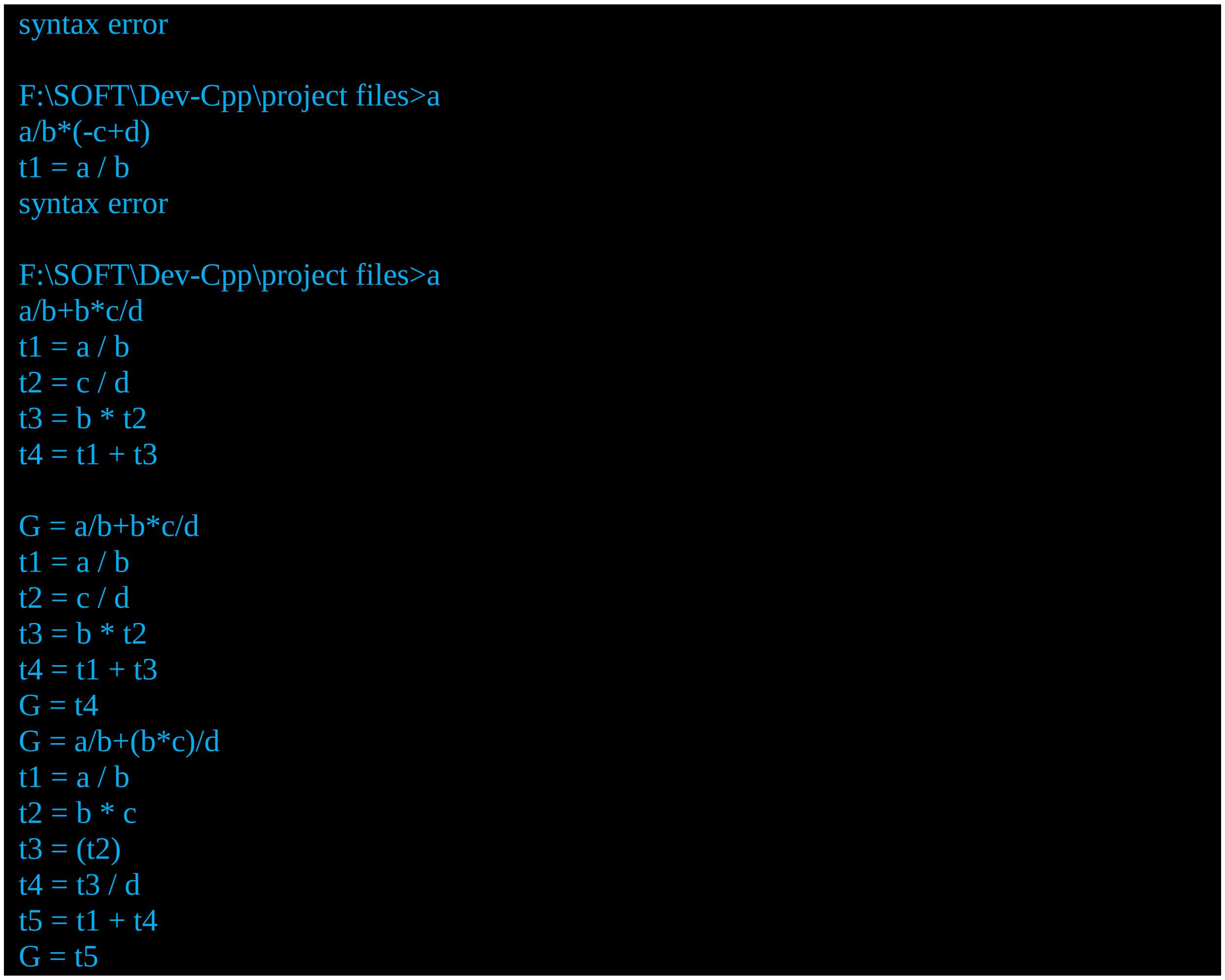
```
/* Simple print the three address code until exit command is hit */
%option noyywrap
%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "siimple_print.tab.h"

void yyerror(char*);
extern YYSTYPE yylval;


%}
DIGITS   [0-9]+
DOUBLE  {DIGITS}(\.{DIGITS})?
NAME    [a-zA-Z]
%%
[ \t]+   { }
{DOUBLE} {
                      strcpy(yylval.cvar,yytext);
                      return DOUBLE;
                }
[-+*/=]  {
                   return *yytext;
             }
"("      {
                      return *yytext;
                }
")"      {
                      return *yytext;
                }
{NAME}   {
                      strcpy(yylval.cvar,yytext);
                      return NAME;
                }
\n           {
                      return *yytext;
                }

exit     {
                      return 0;
                }
.        {
                      char msg[25];
                      sprintf(msg," <%s>","invalid character",yytext);
                      yyerror(msg);
                }
%
```

simple_print.y

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

int yylex(void);
int t_count = 1;
char * str;


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

char* getTemp(int i)
{
    char *ret = (char*) malloc(15);
    sprintf(ret, "t%d", i);
        return ret;
}



%}

%union { char cvar[5]; }
%token <cvar> DOUBLE
%token <cvar> NAME
%token '\n'

%type <cvar> expr
%type <cvar> term

%right '='
%left '+' '-'
%left '*'
%left '/'
%left '(' ')'


%%
program: line program { } | line       { } ;
line: expr '\n' {       t_count =1;     printf("\t\n",$1); }
        | NAME '=' expr '\n' {
                                                t_count-=1;
                                                str = getTemp(t_count);
                                                strcpy($3,str);
                                                printf("%s = %s\n",$1,$3);
                                                t_count=1;
                                                };
expr: expr '+' expr {
                                        str = getTemp(t_count);
                                        strcpy($$,str);
                                        printf("%s = %s + %s\n",$$,$1,$3);
                                        t_count++;
                                        }
        | expr '-' expr {
                                        strcpy($$,getTemp(t_count));
                                        t_count++;
                                        printf("%s = %s - %s \n",$$,$1,$3);
                                        }
        | expr '*' expr {
                                        strcpy($$,getTemp(t_count));
                                        t_count++;
                                        printf("%s = %s * %s \n",$$,$1,$3);
                                        }
        | expr '/' expr {
                                        strcpy($$,getTemp(t_count));
                                        t_count++;
                                        printf("%s = %s / %s \n",$$,$1,$3);
                                        }
        | term                 {
                                        strcpy($$, $1);
                                        }
        | '(' expr ')'  {
                                        strcpy($$,getTemp(t_count));
                                        t_count++;
                                        printf("%s = (%s) \n",$$,$2);
                                        };
term: NAME { strcpy($$,$1);} | DOUBLE {       strcpy($$,$1);};
%%


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



```
F:\SOFT\Dev-Cpp\project files>win_flex_bison3-latest\win_bison -dy siimple_print.y

F:\SOFT\Dev-Cpp\project files>flex siimple_print.l

F:\SOFT\Dev-Cpp\project files>gcc lex.yy.c siimple_print.tab.c

F:\SOFT\Dev-Cpp\project files>a
a*a+b-b
t1 = a * a
t2 = t1 + b
t3 = t2 - b

a/b*-c+d
t1 = a / b
```



```
syntax error

F:\SOFT\Dev-Cpp\project files>a
a/b*(-c+d)
t1 = a / b
syntax error

F:\SOFT\Dev-Cpp\project files>a
a/b+b*c/d
t1 = a / b
t2 = c / d
t3 = b * t2
t4 = t1 + t3

G = a/b+b*c/d
t1 = a / b
t2 = c / d
t3 = b * t2
t4 = t1 + t3
G = t4
G = a/b+(b*c)/d
t1 = a / b
t2 = b * c
t3 = (t2)
t4 = t3 / d
t5 = t1 + t4
G = t5
```

Practical – 9

AIM: Extract Predecessor and Successor from given Control Flow Graph.

Let's take example of below program,

```
int input(x,y){
        if(x==0) return 0;
        else {
                x=10;
                if(x<(y-5)) {
                        y=y-1;
                }
                else {
                        x = y + 5;
                }
                z = y – x;
                return z;
        }
}
```

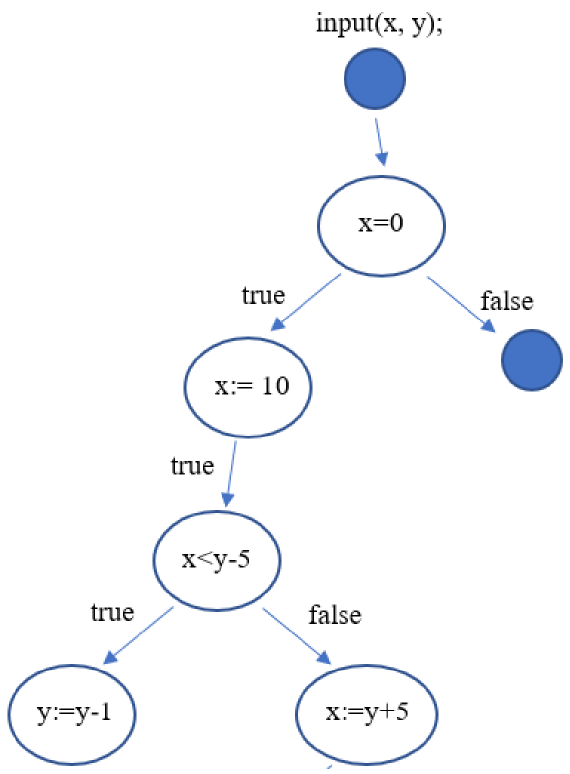


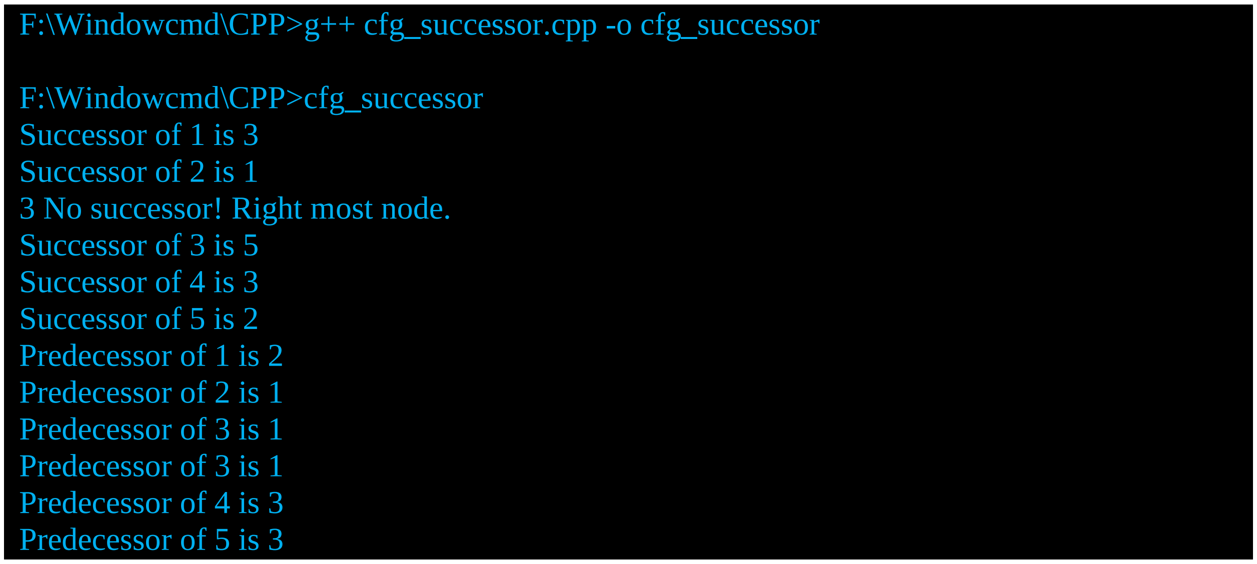
Fig 9.1 CFG for above program

Above graph can be represent as

```
          1
         / \
        2   3
       /
      4
     / \
    5   6
```

```
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
        int data;
        struct Node *left, *right;
};

// Temporary node for case 2
Node* temp = new Node;

// Utility function to create a new tree node
Node* newNode(int data)
{
        Node *temp = new Node;
        temp->data = data;
        temp->left = temp->right = NULL;
        return temp;
}

// function to find left most node in a tree
Node* leftMostNode(Node* node)
{
        while (node != NULL && node->left != NULL)
                node = node->left;
        return node;
}

// function to find right most node in a tree
Node* rightMostNode(Node* node)
{
        while (node != NULL && node->right != NULL)
                node = node->right;
        return node;
}

// recursive function to find the Successor
// when the right child of node x is NULL
Node* findInorderRecursive(Node* root, Node* x )
{
        if (!root)
                return NULL;
        if (root==x || (temp = findInorderRecursive(root->left,x)) ||
                                  (temp = findInorderRecursive(root->right,x)))
        {
                if (temp)
                {
                        if (root->left == temp)
                        {
                                cout << "Successor of " << x->data;
                                cout << " is "<< root->data << "\n";
                                return NULL;
                        }
                }

                return root;
        }

        return NULL;
}

/* This function return the maximum node in tree rooted at node root */
Node *findMaximum(Node *root){
    if(!root)
        return NULL;

    while(root->right) root = root->right;
    return root;
}
void inorderPredecessor(Node *root, Node *K){

    Node *predecessor   = NULL;
    Node *current       = root;

    if(!root)
        return;

    while(current && current->data != K->data){

        if(current->data >K->data){
            current= current->left;
        }
        else{
            predecessor = current;
            current = current->right;
        }
    }
    if(current && current->left){
        predecessor = findMaximum(current->left);
    }
    if(predecessor !=  NULL){
        cout<<"Predecessor of "<<K->data<<" is ";
               cout<<predecessor->data<<"\n";
    }
    else{
        cout << K->data<<" No predecessor!.\n";
    }
}
```

```
// function to find inorder successor of
// a node
void inorderSuccessor(Node* root, Node* x)
{
        // Case1: If right child is not NULL
        if (x->right != NULL)
        {
                Node* inorderSucc = leftMostNode(x->right);
                cout<<"Successor of "<<x->data<<" is ";
                cout<<inorderSucc->data<<"\n";
        }

        // Case2: If right child is NULL
        if (x->right == NULL)
        {
                Node* rightMost = rightMostNode(root);

                // case3: If x is the right most node
                if (rightMost == x)
                        cout << x->data<<" No successor! Right most node.\n";
                else
                        findInorderRecursive(root, x);
        }
}
 int main()
{
        /*
        //         1
        //        / \
        //       2   null
        //      /
        //     3
        //    / \
        //   4   5 */
        Node* root = newNode(1);
        root->left = newNode(2);
        root->right = newNode(3);
        root->left->left = newNode(3);
        root->left->left->left = newNode(4);
        root->left->left->right = newNode(5);


        inorderSuccessor(root, root);
        inorderSuccessor(root, root->left);
        inorderSuccessor(root, root->right);
        inorderSuccessor(root, root->left->left);
        inorderSuccessor(root, root->left->left->left);
        inorderSuccessor(root, root->left->left->right);

        inorderPredecessor(root, root);
        inorderPredecessor(root, root->left);
        inorderPredecessor(root, root->right);
        inorderPredecessor(root, root->left->left);
        inorderPredecessor(root, root->left->left->left);
        inorderPredecessor(root, root->left->left->right);

        return 0;
}
```



```
F:\Windowcmd\CPP>g++ cfg_successor.cpp -o cfg_successor

F:\Windowcmd\CPP>cfg_successor
Successor of 1 is 3
Successor of 2 is 1
3 No successor! Right most node.
Successor of 3 is 5
Successor of 4 is 3
Successor of 5 is 2
Predecessor of 1 is 2
Predecessor of 2 is 1
Predecessor of 3 is 1
Predecessor of 3 is 1
Predecessor of 4 is 3
Predecessor of 5 is 3
```

Fig 9.2 output of cfg graph

Practical – 10

AIM: Study of Learning Basic Block Scheduling Heuristics from optimal data.

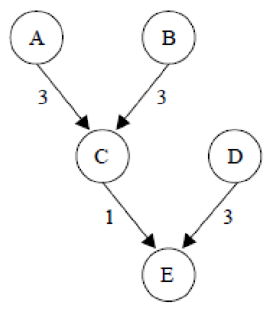
Instruction scheduling is an important step for improving the performance of object code produced by a compiler.

Modern architectures are pipelined and can issue multiple instructions per time cycle. On such processors, the order that the instructions are scheduled can significantly impact performance.

The basic block instruction scheduling problem is to find a minimum length schedule for a basic block a straight-line sequence of code with a single entry point and a single exit point subject to precedence, latency, and resource constraints. Basic block scheduling is important in its own right and also as a building block for scheduling larger groups of instructions such as superblocks. Solving the basic block instruction scheduling problem exactly is known to be difficult, and most compilers use a greedy list scheduling algorithm together with a heuristic for choosing which instruction to schedule next. Such a heuristic usually consists of a set of features and a priority or order in which to test the features.

The heuristic in a production compiler is usually hand-crafted by choosing and testing many different subsets of features and different possible orderings a potentially time-consuming process.

Consider multiple-issue pipelined processors. On such processors, there are multiple functional units and multiple instructions can be issued (begin execution) each clock cycle. Associated with each instruction is a delay or latency between when the instruction is issued and when the result is available for other instructions which use the result. In this paper, we assume that all functional units are fully pipelined and that instructions are typed. Examples of types of instructions are load/store, integer, floating point, and branch instructions. We use the standard labeled directed acyclic graph (DAG) representation of a basic block (see Figure 1(a)). Each node corresponds to an instruction and there is an edge from i to j labeled with a positive integer l (i, j) if j must not be issued until i has executed for l (i, j) cycles.



(a)

```
A    r1 ← a
B    r2 ← b
     NOP
     NOP
C    r1 ← r1 + r2
D    r3 ← c
     NOP
     NOP
E    r1 ← r1 + r3
```

(b)

```
A    r1 ← a
B    r2 ← b
D    r3 ← c
     NOP
C    r1 ← r1 + r2
E    r1 ← r1 + r3
```

(c)

Figure 1: (a) Dependency DAG associated with the instructions to evaluate $(a+b)+c$ on a processor where loads from memory have a latency of 3 cycles and integer operations have a latency of 1 cycle; (b) a schedule; (c) a better schedule.

Given a labeled dependency DAG for a basic block, a schedule for a multiple-issue processor specifies an issue or start time for each instruction or node such that the latency constraints are satisfied and the resource constraints are satisfied. The latter are satisfied if, at every time cycle, the number of instructions of each type issued at that cycle does not exceed the number of functional units that can execute instructions of that type. The length of a schedule is the number of cycles needed for the schedule to complete; i.e., each instruction has been issued at its start time and, for each instruction with no successors, enough cycles have elapsed that the result for the instruction is available. The basic block instruction scheduling problem is to construct a schedule with minimum length.

Instruction scheduling for basic blocks is known to be NP-complete for realistic architectures. The most popular method for scheduling basic blocks continues to be list scheduling. A list scheduler takes a set of instructions as represented by a dependency DAG and builds a schedule using a best-first greedy heuristic. A list scheduler generates the schedule by determining all instructions that can be scheduled at that time step, called the ready list, and uses the heuristic to determine the best instruction on the list. The selected instruction is then added to the partial schedule and the scheduler determines if any new instructions can be added to the ready list. The heuristic in a list scheduler generally consists of a set of features and an order for testing the features. Some standard features are as follows. The path length from a node I to a node j in a DAG is the maximum number of edges along any path from i to j. The critical-path distance from a node i to a node j in a DAG is the maximum sum of the latencies.

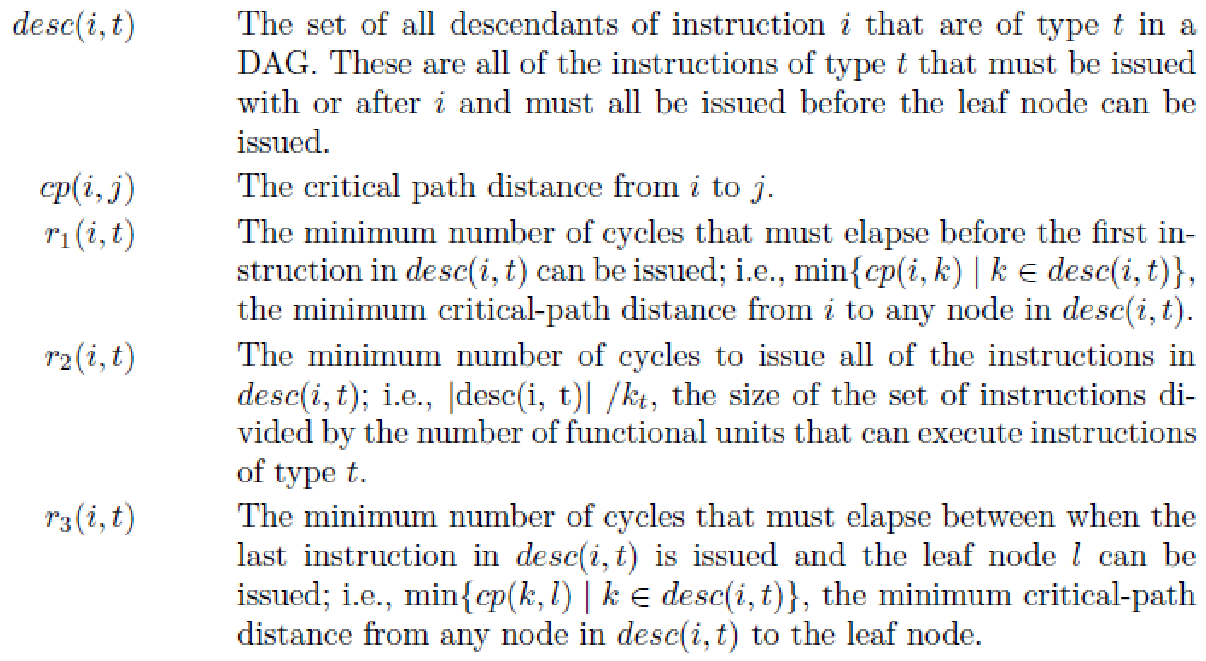
Learning a Heuristic

To use supervised learning techniques to construct a list scheduling heuristic, the problem first has to be phrased as a classification problem. Moss et al. note that to choose the best instruction on a ready list to schedule next, it is sufficient to be able to compare two instructions i and j and return true if i should be scheduled before j; and false otherwise. We thus have a binary classification problem. The choice of distinguishing features is critical to successfully learning a classifier. We began with almost 60 features that we felt would be promising. These included all of the classic features surveyed by Smotherman et al. except for the features having to do with register pressure and a few features that were irrelevant or whose roles were better performed by other features. These classic features all measure properties of an instruction on the ready list. We also included features that measured properties of the DAG to be scheduled (such as the source language and the number of instructions of each type) and of the architecture
(Such as the number of functional units of each type).

A more accurate classifier can sometimes be achieved by synthesizing new features from existing basic features. We also included many such synthesized features. Some of the novel features were constructed by applying simple functions to basic features. Examples include comparison of two features, maximum of two features, and the average of several features.

One of the novel features that we constructed, resource-based distance to the leaf node, turned out to be the best feature among all of the features that we studied. Consider the notation shown in Table 1. For convenience of presentation, we are assuming that a DAG has a single leaf node; i.e., we are assuming a fictitious node is added to the DAG and zero latency arcs are added from the leaf nodes to this fictitious node. The resource-based distance from a node i to the leaf node is given by, rb(i) = max{r1 (i, t) + r2 (i, t) + r3 (i, t)}, where we are finding the maximum over all instruction types t. The distance was sometimes improved by “removing” a small number of nodes (between one and three nodes) from desc (i, t). This was done whenever removing these nodes led to an increase in the value of rb (i); i.e., the decrease in r2 (i, t) was more than offset by the increase in r1(i, t) + r3(i, t).

Table 1: Notation for the resource-based distance to leaf node feature.



$desc(i,t)$	The set of all descendants of instruction i that are of type t in a DAG. These are all of the instructions of type t that must be issued with or after i and must all be issued before the leaf node can be issued.
$cp(i,j)$	The critical path distance from i to j.
$r_1(i,t)$	The minimum number of cycles that must elapse before the first instruction in $desc(i,t)$ can be issued; i.e., $\min\{cp(i,k) \mid k \in desc(i,t)\}$, the minimum critical-path distance from i to any node in $desc(i,t)$.
$r_2(i,t)$	The minimum number of cycles to issue all of the instructions in $desc(i,t)$; i.e., $\lvert \mathrm{desc(i,\ t)} \rvert\ /k_t$, the size of the set of instructions divided by the number of functional units that can execute instructions of type t.
$r_3(i,t)$	The minimum number of cycles that must elapse between when the last instruction in $desc(i,t)$ is issued and the leaf node l can be issued; i.e., $\min\{cp(k,l) \mid k \in desc(i,t)\}$, the minimum critical-path distance from any node in $desc(i,t)$ to the leaf node.

An important next step, prior to learning the heuristic, is to filter the features. The goal of filtering is to select the most important features for constructing a good heuristic. Only the selected features are then passed to the learning algorithm and the features identified as irrelevant or redundant are deleted. There are two significant motivations for performing this preprocessing step: the efficiency of the learning process can be improved and the quality of the heuristic that is learned can be improved (many learning methods, decision tree learning included, do poorly in the presence of redundant or irrelevant features). Several feature filtering techniques have been developed (see, for example, and the references therein). In our work, a feature was deleted if both: (i) the accuracy of a single feature decision tree classifier constructed from this feature was no better than random guessing on the validation set; and (ii) the accuracy of all two-featured decision tree classifiers constructed from this feature and each of the other features was no better than or a negligible improvement over random guessing on the validation set. The motivation behind case (ii) is that a feature may not improve classification accuracy by itself, but may by useful together with another feature.