Laboratory Exercises

This resource will enable the readers to

- design and implement a lexical analyser (*scanner*) for the given lexical specification of tokens of the language
- design and implement the syntax analyser (*parser*) for the given syntactic specification of the structure of the language
- generate the intermediate code on successful parsing of the source code
- generate the machine code for an assumed target machine
- do a mini project covering all the features of the un-optimized compiler for a restricted language

LIST OF EXERCISES

1. Develop the code for converting the given infix expression into a postfix one.
2. Find the non-deterministic finite automata from the regular expression (RE) given in postfix form.
3. Construct the deterministic finite state automata (DFA) from the given non-deterministic finite state automata (NFA).
4. Construct a predictive parsing table from the given grammar rules, and use this table to parse the given sentence.
5. Construct the LR(0) parsing table from the given grammar rules.
6. Use the Lex tool for the separation of token from a text.
7. Simulate a numerical calculator using the tools Lex and Yacc.
8. Construct a hash table for the storage and retrieval of the symbol.
9. Generate the intermediate code (quadruples) from the given lexical and syntactical specification of a programming language using Lex and Yacc tools.
10. Generate the machine code from the given intermediate code.
11. Do a mini project on the compiler incorporating all the stages of a compiling process for a simple language.

Hints

1. Routines such as stack, first, and follow will appear in one place only.
2. All the exercises must be prepared with a minimum of three files: (a) header file, (b) implementation file, and (c) application file, where the main function will be present. (For convenience of display, all the files are merged.) The idea behind using three files is to differentiate between the developer and user sides. In the developer

side, all the three files could be made visible. In the user side, only the header file and application files could be made visible, which means that the object code of the implementation file could be given to the user along with the interface given in the header file, hiding the source file, if required.

3. It is recommended not to use any input/output (I/O) functions (`scanf` and `printf`) in the implementation file because the user is not aware of the contents of the file. If any error message should be displayed to the user, the error code can be returned from the implementation, which can be mapped with the table of error message.

These hints may be followed when the exercises are made in a deliverable form. However, for more clarity and better understanding, we have combined all the functions with appropriate `printf` functions.

EXERCISES IN DETAIL

EXERCISE 1 Develop the code for converting the given infix expression into a postfix expression.

Aim

To convert the infix expression into a postfix expression.

Theory

Postfix expression is one of the intermediate codes to be used in the translation of a source code into a machine code. For example, an expression such as 'a + b * c' is converted into 'abc*+'. In postfix expression, the operators are moved to the position that is next to the required operands. To convert the infix expression into a postfix expression, three data structures are used: (a) a stack, (b) input buffer, and (c) output buffer. During the evaluation of the expression, some portions of the expressions are evaluated first and the other portions are evaluated next, depending on the priority of the operator.

Steps

1. Create a buffer (array) to hold both infix and postfix expressions.
2. Create an operator stack.
3. Fix priorities for the operators in the infix string and stack.
4. For every operand or operator read from the infix expression string; do Step 5.
5. If it is operand, place it in postfix string.
6. Else either push the operator onto stack or move the contents of stack to postfix string depending on the priority of the operator compared with that of the stack operator.

Modules

1. Performing stack-related operations
   (a) Creating and disposing the stack
   (b) Checking the empty and full status of the stack for pop and push operations
   (c) Push, pop, and peep at top operations
2. Determining the priority of an operator

3. Deciding whether the element is an operator or an operand
4. Converting an infix expression into a postfix expression

Required Data Structure

1. Buffer for infix and postfix expressions
2. Operator array
3. Infix and stack priority of the operators

Algorithm

The following algorithm outlines the main procedure of converting the infix expression into a postfix expression.

Algorithm `infix2postfix()`

```
// Input: Infix expression
// Output: Postfix expression
{
        defineOperatorPriorities();
        x = readInfix();
        if(isOperator(x))
        {
                ip = findInfixPriority()
                sp = findStackPriority()
                if(ip > sp)
                        push(stack, x);
                else
                {

                    while(sp > ip)
                        // Keep moving operator y on stack to postfix string.
                        add(postfix, y) // y should not be any parenthesis.
                    push(s, x);
                }
        }
        else
                add(postfix, x)

}
```

The following C program code list consists of

1. includes and macrodefinitions
2. data declarations
3. function prototype declarations
4. definitions of the functions
5. main function

The following the code list, the outputs for the various test cases are presented.

Program listing

```
// Include and macrodefinitions
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<string.h>
#define EMPTYSTACK -1
#define MAXSIZE 50
// Data declarations
typedef char ElementType;
typedef struct StackRecord
{
        int capacity;
        int top;
        ElementType *array;
}StackRecord;

typedef StackRecord *Stack;

ElementType op[] = {'*', '+', '&', '|', '(',')', '#'};
int sp[] = {5, 4, 3, 2, 1, -1, -2};
int ip[] = {5, 4, 3, 2, 5, 0, -1};

// Function prototype declarations
void push(Stack s, ElementType x);
ElementType top(Stack s);
void pop(Stack s);
ElementType topAndPop(Stack s);
Stack createStack(int mexLength);
void disposeStack(Stack s);
int checkEmpty(Stack s);
int checkFull(Stack s);

int prior(ElementType x, int y);
int isOperator(ElementType x);
ElementType* in2post(ElementType *infix);

// Function definitions
Stack createStack(int maxElements)
```

```
{
        Stack s;
        s = (struct StackRecord*)malloc(sizeof(struct StackRecord));
        s→capacity = maxElements;
        s→array = (ElementType *)malloc(MAXSIZE*sizeof(ElementType));
        s→top = 0;
        s→array[0] = '#';
        return s;
}

void push(Stack s, ElementType x)
{
        s→top = s→top + 1;
        s→array[s→top] = x;
}

ElementType top(Stack s)
{
        ElementType x;
        x = s→array[s→top];
        return x;
}

void pop(Stack s)
{
        s→top = s→top – 1;
}

ElementType topAndPop(Stack s)
{
        ElementType x;
        x = s→array[s→top];
        s→top = s→top – 1;
        return x;
}

int checkEmpty(Stack s)
{
        if(s→top == 0||s→top == 0)
        {
                return –1;
        }
```

```
        return 1;
}

int checkFull(Stack s)
{
        if(s→top == s→capacity)
        {
                return 1;
        }
        return 0;
}

int prior(ElementType x, int y)
{
        int pt;
        int ii;

        for(ii = 0;ii<;7;ii++)
        {
                if(x == op[ii])
                {
                        if(y == 0)
                        {
                                pt = sp[ii];
                        }
                        else
                        {
                                pt = ip[ii];
                        }
                }
        }
        return pt;
}

int isOperator(ElementType x)
{

        int ck = 0, ii;
        for(ii = 0;ii < 7;ii++)
        {
                if(x == op[ii])
                {
```

```
                        ck = 1;
                        return ck;
                }
        }
        return ck;
}

ElementType* in2post(ElementType *infix)
{
        int i, j, maxElements;
        ElementType *postfix;
        ElementType x, y, ip, sp, h;
        Stack s;
        maxElements = strlen(infix);
        postfix = (ElementType*)malloc(maxElements*sizeof(ElementType));
        if(postfix == NULL)
        {
                printf("Memory allocation problem\n");
                exit(-1);
        }
        s = createStack(maxElements);
        for(i = 0;i < maxElements;i++)
        {
                x = infix[i];
                if(isOperator(x))
                {
                        y = top(s);
                        ip = prior(x, 1);   // Infix priority
                        sp = prior(y, 0);   // Stack priority
                        if(ip > sp)      // ip > SP
                        {
                                push(s, x);
                        }
                        else
                        {
                                while(sp > ip)
                                {
                                        y = top(s);
                                        pop(s);
                                        sp = prior(y, 0);
                                        if(y != ')'&& y != '('
                                        {
                                                postfix[j++] = y;
```

```
                                        }
                                        h = checkEmpty(s);
                                        if(h == -1)
                                                break;
                                }
                                push(s, x);
                        }
                }
                else
                {
                        postfix[j++] = x;
                }
        }
        postfix[j] = '\0';
        return postfix;

        }
main()
{
        ElementType infix[MAXSIZE],*postfix;
        printf("\nEnter the infix expression\t");
        scanf("%s", infix);
        postfix = in2post(infix);
        printf("infix: %s is converted into postfix:%s\n", infix, postfix);
}
```

Output

Test case 1

```
Enter the infix expression a + b*c#
infix: a + b*c# is converted into postfix: abc*+
```

Test case 2

```
Enter the infix expression a * b+c#
infix: a * b+c# is converted into postfix: ab*c+
```

Test case 3

```
Enter the infix expression a + b*(c + d)#
infix: a + b*(c + d)# is converted into postfix: abcd +*+
```

Test case 4

```
Enter the infix expression (a + c)*b + d#
infix: (a + c) * b + d is converted into postfix: ac+b*d+
```

Exercise for the Students

The aforementioned demonstration assumes only character type for the operators and operands. Extend it to operands of type strings.

EXERCISE 2 Find the non-deterministic finite automata from the regular expression given in postfix form.

Aim

To convert the RE into non-deterministic finite automata and display the transition table.

Theory

Any lexical specification can be given in an RE. To separate the token, the token recognizer has to be designed, which starts from the conversion of RE into NFA. The output of this exercise can be used as the input for the next exercise, that is, the conversion of NFA into DFA.

Steps

1. Initialize the transition table.
2. Read the RE in postfix form and allocate space for the transition table.
3. Develop the code for the five operations such as
   (a) concatenation (.)
   (b) alternate operation (|)
   (c) closure operations *, +, ?

Modules

1. Initialization of the table (the table is indexed by state number in row and operands and epsilon in column)

   initNFATransTable

2. Conversion of RE to NFA (make entries in the transition table for the given state and symbols)

   reg2nfa

3. Display of the table

   displayTransTable

4. Storage of the NFA transition table in the file for the next NFA-to-DFA conversion operation

   writeNFA

Algorithm

Refer Fig. 3.8 in Section 3.4.2, which describes the Thompson algorithm and the subsequent discussions with examples.

Source Code Development

The *makefile* (project file) has the following files compiled and linked, developed in a Linux environment.

```
scan:     scan.o nfa.o stack.o
          g++ scan.o nfa.o stack.o -o scan
scan.o:   scan.cc nfa.h stack.h decl.h
          g++ -c scan.cc
nfa.o:    nfa.cc nfa.h stack.h decl.h
          g++ -c nfa.cc
stack.o:  stack.cc stack.h decl.h
          g++ -c stack.cc
```

Data Structure and Prototype Declarations

```
#include<stdlib.h>
#include<malloc.h>
#include<string.h>
typedef char ElementType;
#define NOOPERANDS 50
#define NOOPERATORS 50
typedef long int Pointer;
class NFA
{
private:
        int startState;
        int finalState;
        int noOperands;
        int noSymbols;
        int noOperators;
        int maxStates;
        int noStates;
        int count;
        ElementType operators[NOOPERATORS];
        ElementType operands[NOOPERANDS];
        int **transTable;

        void initNFATransTable(int rows, int cols);
        int checkOperator(ElementType element);
        int isMember(ElementType e);
        void findNoChars(ElementType reg[]);
        int findOpdIndex(ElementType r, ElementType reg[]);
public:
        NFA();
        void reg2nfa(ElementType reg[]);
        void displayTransTable();
```

```
        void writeNFA(FILE *fp);

};
```

Source code for implementation of RE-to-NFA conversion operations

```
NFA::NFA()
{
        int i, j, n;
        startState = 0;
        finalState = 0;
        noOperands = 0;
        noSymbols = 0;
        noOperators = 0;
        maxStates = 0;
        noStates = 0;
        count = 0;
        for(i = 0;i < NOOPERATORS;i++)
                operators[i] = ' ';
        for(i = 0;i < NOOPERANDS;i++)
                operands[i] = ' ';
}

int NFA::checkOperator(ElementType element)
{
        int flag = 0;

        int k;
        switch(element)
        {
                case '*':
                case '+':
                case '?':
                        return 1;

                case '.':
                case '|':
                        return 2;
                default: return 0;
        }
}
```

```
int NFA::isMember(ElementType e)
{
        int i;
        for(i = 0;i < noOperands;i++)
                if(operands[i] == e)
                {
                        printf("%c %c\n", operands[i], e);
                        return 1;
                }
        return 0;
}

void NFA::findNoChars(ElementType reg[])
{
        int i = 0;
        for(i = 0;reg[i] != '\0';i++)
        {
                if(reg[i]>= 'a'&&reg[i] <= 'z')
                {
                        if(!isMember(reg[i]))
                        {
                                operands[noOperands] = reg[i];
                                        noOperands++;
                        }
                        count++;
                }
                else if(reg[i] == '*'||reg[i] == '|'||reg[i] == '+'||reg[i]
                        == '?')
                {
                        operators[noOperators];
                                noOperators++;
                        count++;
                }
        }
}

void NFA::initNFATransTable(int rows, int cols)
{
        int i, j, n;

        transTable = (int**)malloc(rows*sizeof(Pointer));
```

```
        for(n = 0;n < rows;n++)
        {
                transTable[n] = (int*)malloc(cols*sizeof(Pointer));
        }

        for(i = 0;i < rows;i++)
        {
                for(j = 0;j < cols;j++)
                {

                        transTable[i][j] = -1;

                }
        }
}

void NFA::displayTransTable()
{
        int i, j;
        printf("\n\n\tResultant e-NFA\n\n");
        printf("-------------------------------\n");
        printf("State\t");
        for(i = 0;i < noSymbols;i++)
                printf("%c   ", operands[i]);
        printf("eps1  eps2\n\n");
        printf("-------------------------------\n");
        for(i = 0;i < noStates;i++)
        {
                printf("%d\t", i);
                        for(j = 0;j < noSymbols;j++)
                                if(transTable[i][j] == -1)
                        printf("-\t");
                else
                        printf("%d\t", transTable[i][j]);
                printf("\n");
        }
        printf("-------------------------------\n");
}

void NFA::writeNFA(FILE *fp)
{
        int i, j;
```

```
        fprintf(fp,"%d\t%d\t%d\t%d\t",  noStates,  noSymbols,  startState,
finalState);
        for(i = 0;i < noOperands;i++)
                fprintf(fp,"   %c\t", operands[i]);
        fprintf(fp,"\n");
        for(i = 0;i < noStates;i++)
        {
                // printf("%d:  ");
                for(j = 0;j < noSymbols;j++)
                                fprintf(fp," %d\t", transTable[i][j]);
                fprintf(fp,"\n");
        }
}

int NFA::findOpdIndex(ElementType r, ElementType reg[])
{
        int o, x;
        // x = noofoperands(reg);
        for(o = 0;o < noOperands;o++)
        {
                if(r == operands[o])
                        return(o);
        }
        return -1;
}

void NFA::reg2nfa(ElementType reg[])
{
        int index = -1, opdIndex = 1;
        int  i  =  0,  secondStartState,  secondAcceptState,  firstAcceptState,
firstStartState;
        Stack s;
        #define fe noOperands
        #define se noOperands + 1

        findNoChars(reg);
        maxStates = 2*count;
        noSymbols = noOperands + 2;  // To include epsilon

        initNFATransTable(maxStates, noSymbols);

        s.createStack(2*count);
        for(i = 0;reg[i] != '\0';i++)
```

```
{
        if(checkOperator(reg[i]) == 0)  // Operand
        {
                index++;
                startState = index;
                s.push(index);
                opdIndex = findOpdIndex(reg[i], reg);
                transTable[index][opdIndex] = index + 1;
                index++;
                finalState = index;
                s.push(index);
        }
        else if(reg[i] == '|')
        {
                secondAcceptState = s.topAndPop();// SECOND ACCEPTING
                secondStartState = s.topAndPop();// SECOND START
                firstAcceptState = s.topAndPop();// FIRST ACCEPTING
                firstStartState = s.topAndPop();// FIRST START
                        index++;
                startState = index;
                s.push(index);
                transTable[index][fe] = firstStartState;
                transTable[index][se] = secondStartState;
                index++;
                finalState = index;
                transTable[firstAcceptState][fe] = index;
                transTable[secondAcceptState][fe] = index;
                s.push(index);
        }
        else if(reg[i] == '.')
        {
                secondAcceptState = s.topAndPop();
                secondStartState = s.topAndPop();
                firstAcceptState = s.topAndPop();
                firstStartState = s.topAndPop();
                startState = firstStartState;  //
                transTable[firstAcceptState][fe] = secondStartState;
                s.push(firstStartState);
                s.push(secondAcceptState);
        }
        else if(reg[i] == '*')
        {
```

```
                        index++;
                        startState = index;   //
                        firstAcceptState = s.topAndPop();
                        firstStartState = s.topAndPop();
                        s.push(index);
                        transTable[index][se] = firstStartState;
                        transTable[index][fe] = ++index;
                        finalState = index;   //
                        transTable[firstAcceptState][fe] = firstStartState;
                        transTable[firstAcceptState][se] = index;
                        s.push(index);
                }
                else if(reg[i] == '+')
                {
                        firstAcceptState = s.topAndPop();
                        firstStartState = s.topAndPop();
                        index++;
                        startState = index;//
                        transTable[index][fe] = firstStartState;
                        s.push(index);
                        index++;
                        finalState = index;//
                        s.push(index);
                        transTable[firstAcceptState][fe] = index;
                        transTable[firstAcceptState][se] = firstStartState;
                }
                else if(reg[i] == '?')
                {
                        firstAcceptState = s.topAndPop();
                        firstStartState = s.topAndPop();
                        index++;
                        startState = index; //
                        s.push(index);
                        transTable[index][fe] = firstStartState;
                        transTable[index][se] = ++index;
                        finalState = index;
                        transTable[firstAcceptState][fe] = index;
                        s.push(index);
                }
        }
        noStates = index + 1;
}
```

Main function calling the routines for the conversion of RE into NFA

```
#include "decl.h"
#include "stack.h"
#include "nfa.h"
int main()
{
        NFA nfa;
        char reg[50];
        int nofc, nfod;
        int st, ft;
        FILE *fp;
        char fname[20];
        printf("Enter regular expression in postfix form\n");
        scanf("%s", reg);
        printf("Reg expression is: %s\n", reg);

        nfa.reg2nfa(reg);
        nfa.displayTransTable();

        printf("Enter the file name for storing NFA states\n");
        scanf("%s", fname);
        fp = fopen(fname,"w");
        if(NULL == fp)
        {
                printf("%s:Error in file open.. Could not write NFA into
file\n", fname);
                exit(-1);
        }
        nfa.writeNFA(fp);
        return 1;

}
```

Output

Case 1: Output for the regular expression alb

The following numbers indicate the total number of NFA states, number of symbols, start state, final state, and the operands.

```
6    4    4    5    a    b
```

Transition table

NFA	a	b	ε1	ε2
0	1	-1	-1	-1
1	-1	-1	5	-1
2	-1	3	-1	-1
3	-1	-1	5	-1
4	-1	-1	0	2
5	-1	-1	-1	-1

-1 indicates no transition.

Case 2: Output for the RE ab

```
4   4   3   a   b
```

Transition table

NFA	a	b	ε1	ε2
0	1	-1	-1	-1
1	-1	-1	2	-1
2	-1	3	-1	-1
3	-1	-1	-1	-1

Exercise for the Students

Use this program and verify the transition table for the RE (a|b)abb.

EXERCISE 3 Construct the deterministic finite state automata from the given non-deterministic finite state automata.

Aim

To convert the given NFA into a DFA (not a optimized one).

Theory

Non-deterministic automata is a finite state machine, which can have two transitions for the same event, making decision making for the transition of states difficult. It also has $\in$-transitions. To design the lexical analyser, first the REs are converted into NFA, and then the NFA has to be converted into DFA. The resultant DFA will have unique transitions and will not have $\in$-transitions.

Steps

1. Read the NFA transition table available from the previous exercises, or we can create our own transition table by working out from the given RE.
2. Initialize all the required data structures for
   (a) NFA transition table
   (b) DFA transition table
   (c) Set of states (SOS)
3. Start with the initial state of NFA.
4. Find the closure (may give more than one state and hence we call it as Set of State (SOS)) of this initial state to get the initial state of DFA

5. Find the transitions from each state of DFA for each operand and find the e-closure of this state. Make entry of the resultant set of states in the DFA transition table.
6. Repeat Step 5 till all the states (SOS) of DFA are processed for the possible transitions.

Modules

1. Reading the transition of NFA and initializing the DFA transition table

   `readNFA`
   `initDFATables`
2. Finding e-closure of a given state(s)

   `eClosure`
3. Moving from one state of DFA to another state of DFA for an operand

   `move`
4. Labelling the subset construction module, which constructs the sub of states of the NFA states, to be the states of DFA

   `constructSubset`
5. Other modules are for reading the transition table of NFA, checking the duplicates of the set of states, sorting the set of states, displaying the table, and so on.

Algorithm

For the details of the modules *constructSubset*, *eClosure*, and *move*, refer the algorithms shown in Fig. 3.26.

Data structures and prototype declarations

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<string.h>
#define UNCHECKED 0
#define CHECKED 1
#define EMPTYSTACK -1
#define MAXSTACKSIZE 50
#define NOOPERANDS 50
#define NOOPERATORS 50
class SOS
{
public:
        int *sos;
        int nos;
        SOS(int);
        void dispSOS();
};
```

```
class DFA
{
private:
        int maxDFAStates;
        int noDFAStates;
        int noNFAStates;
        int noSymbols;
        int nfaStartState;
        int nfaFinalState;
        int noOperands;
        char operands[NOOPERANDS];
        int **nfaTransTable;
        int **dfaTransTable;
        int *dfaStatesChecked;
public:
        DFA();
        SOS **dfaSOS;
        SOS* eClosure(SOS*);
        SOS* move(SOS* ss, char b);
        void constructSubset();
        int isPresent(int a[], int n, int x);
        void readNFA(FILE *fp);
        void initDFATables();
        int findOpdIndex(char r);
        void displayNFATransTable();
        void displayDFATransTable();
        SOS* eClosure(SOS*);
        int checkDFAStates(SOS *ss);
        int isSOSEqual(SOS *ss1, SOS *ss2);
        SOS *sort(SOS *ss);
};
```

Source code listing

```
#include "decl.h"
#include "cstack.h"
#include "istack.h"
#include "dfa.h"
SOS::SOS(int ns)
{
        int i;
```

```
        nos = ns;
        sos = (int*)malloc(sizeof(long int)*nos);
        for(i = 0;i < nos;i++)
        sos[i] = -1;
}
void SOS::dispSOS()
{
        int i;
        printf("[ ");
        for(i = 0;i < nos;i++)
                if(sos[i] != -1)
                        printf("%d ", sos[i]);
        printf("]");
}

DFA::DFA()
{
        noDFAStates = 0;
}

SOS* DFA::sort(SOS *ss)
{
        int i, j, t;
        for(i = 0;i < ss→nos - 1;i++)
                for(j = i + 1;j < ss→nos;j++)
                        if(ss→sos[i] > ss→sos[j])
                        {
                                t = ss→sos[i];
                                ss→sos[i] = ss→sos[j];
                                ss→sos[j] = t;
                        }
        return ss;
}

int DFA::isSOSEqual(SOS *ss1, SOS *ss2)
{
        int i;
        for(i = 0;i < ss1→nos;i++)
                if(ss1→sos[i] != ss2→sos[i])
                        return 0;
        return 1;
}
```

```
int DFA::checkDFAStates(SOS *ss)
{
        int i;
        for(i = 0;i < noDFAStates;i++)
                if(isSOSEqual(ss, dfaSOS[i]))
                        return i;
        return -1;
}

void DFA::constructSubset()
{
        SOS *ss0, *ss, *ss1,*ss2;
        char b;
        int dfaState, opdIdx, retState, i;
        noDFAStates = 0;
        ss0 = new SOS(noNFAStates);
        ss0→sos[0] = nfaStartState;
        ss = eClosure(ss0);
        dfaSOS[noDFAStates++] = ss;

        for(dfaState = 0;dfaState < noDFAStates;dfaState++)
        {
                ss = dfaSOS[dfaState];
                if(dfaStatesChecked[dfaState] == UNCHECKED)
                {
                        for(opdIdx = 0;opdIdx <; noOperands;opdIdx++)
                        {
                                b = operands[opdIdx];
                                ss1 = move(ss, b);
                                ss2 = eClosure(ss1);
                                if(ss2→nos == 0)
                                {
                                        dfaTransTable[dfaState][opdIdx] = -1;
                                        continue;
                                }
                                retState = checkDFAStates(ss2);

                                if(retState == -1)
                                {
                                        dfaSOS[noDFAStates++] = ss2;
                                        dfaTransTable[dfaState][opdIdx] =
noDFAStates - 1;
                                }
```

```
                                else
                                        dfaTransTable[dfaState][opdIdx] =
retState;

                        }
                        dfaStatesChecked[dfaState] = CHECKED;
                }

        }
}

SOS* DFA::eClosure(SOS* ss)
{
        IStack s;
        SOS *tss;
        int nos;
        int p, q;
        int i, stCount = 0;
        // Push all the states onto stack.
        nos = ss→nos;
        tss = new SOS(nos);
        for(i = 0;i < ss→nos;i++)
                if(ss->sos[i] != -1)
                {
                        s.push(ss→sos[i]);
                }
        while(!(s.checkempty()))
        {
                q = s.topAndPop();
                tss→sos[stCount++] = q;
                p = nfaTransTable[q][noOperands];   // To get epsilon1

                if(p != -1 && (!isPresent(tss→sos, stCount, p) ) )
                        s.push(p);
                p = nfaTransTable[q][noOperands + 1];   // To get epsilon2

                if(p != -1 && (!isPresent(tss→sos, stCount, p) ) )
                        s.push(p);
        }
        tss→nos = stCount;
        return sort(tss);
}
```

```
SOS* DFA::move(SOS *ss, char b)
{
        SOS *ss1, *ss2;    // ss are ss1 are the set of states with nos as
the number of states.

        int i, st = -1, k = 0, state = -1;
        int opdIdx;
        int nos;
        nos = ss→nos;
        ss1 = new SOS(nos);
        ss2 = new SOS(nos);
        opdIdx = findOpdIndex(b);
        if(opdIdx != -1)
                for(i = 0;i < nos;i++)
                {
                        state = ss→sos[i];
                        st = nfaTransTable[state][opdIdx];
                        if(st != -1 && (!isPresent(ss1→sos, k, st) ) )
// Check for duplicates.
                        {
                                ss1→sos[k] = st;
                                k++;
                        }
                }
        return ss1;
}

int DFA::isPresent(int a[], int n, int x)
{
        int i;
        for(i = 0;i < n;i++)
                if(a[i] == x)
                        return 1;
        return 0;
}
void DFA::displayDFATransTable()
{
        int i, j, k;
        printf("\n\n\tResultant DFA\n\n");
        printf("-------------------------\n");

        printf("State\t");
        for(i = 0;i < noOperands;i++)
```

```
                printf("\t%c ", operands[i]);
        printf("\n------------------------\n");
        for(i = 0;i < noDFAStates;i++)
        {
                dfaSOS[i]->dispSOS();
                for(j = 0;j < noOperands;j++)
                {
                        if(dfaTransTable[i][j] != -1)
                                printf("\t%d ", dfaTransTable[i][j]);
                        else
                                printf("\t-  ");
                }
                printf("\n");
        }
        printf("-------------------------\n");
}

void DFA::readNFA(FILE *fp)
{
        int i, j;
        char ch;
        fscanf(fp,"%d%d%d%d",&noNFAStates,&noSymbols,&nfaStartState,&nfaFi
nalState);
        noOperands = noSymbols - 2;
        maxDFAStates = noNFAStates;

        for(i = 0;i < noOperands;i++)
        {
                fscanf(fp," %c",&operands[i]);
                // printf("%c\t", operands[i]);
        }

        nfaTransTable = (int**)malloc(noNFAStates*sizeof(long int));
        for(i = 0;i < noNFAStates;i++)
        {
                nfaTransTable[i] = (int*)malloc(noSymbols*sizeof(long int));
        }
        for(i = 0;i < noNFAStates;i++)
        {
                for(j = 0;j < noSymbols;j++)
                        fscanf(fp,"%d",&nfaTransTable[i][j]);
        }
}
```

```
void DFA::displayNFATransTable()
{
        int i, j;
        printf("\n\n\tGiven e-NFA\n\n");
        printf("-------------------------------\n");
        printf("State\t");
        for(i = 0;i < noOperands;i++)
                printf("%c   ", operands[i]);
        printf("eps1  eps2\n\n");
        printf("-------------------------------\n");
        for(i = 0;i < noNFAStates;i++)
        {
        printf("%d\t", i);
                for(j = 0;j < noSymbols;j++)
                        if(nfaTransTable[i][j] == -1)
                        printf("-\t");
                else
                        printf("%d\t", nfaTransTable[i][j]);
                printf("\n");
        }
        printf("-------------------------------\n");
}

void DFA::initDFATables()
{
        int i, j;

        SOS *ss;
        dfaTransTable = (int**)malloc(maxDFAStates*sizeof(long int));
        dfaStatesChecked = (int *)malloc(maxDFAStates*sizeof(long int));
        for(i = 0;i < maxDFAStates;i++)
        {
                dfaTransTable[i] = (int*)malloc(noOperands*sizeof(long int));
                dfaStatesChecked[i] = UNCHECKED;
        }

        for(i = 0;i < maxDFAStates;i++)
        {
                for(j = 0;j < noOperands;j++)
                {
                        dfaTransTable[i][j] = -1;
                }
```

```
        }
        dfaSOS = (SOS**)malloc(maxDFAStates*sizeof(long int));

        for(i = 0;i < maxDFAStates;i++)
        {
                dfaSOS[i] = new SOS(noNFAStates);
        }
}

// Find the index of the operand.

int DFA::findOpdIndex(char r)
{
        int o;
        for(o = 0;o < noOperands;o++)
        {
                if(r == operands[o])
                        return(o);
        }
        return -1;
}
```

Output

Case 1: Output for the non-deterministic finite state automata table obtained from RE(a|b)

```
Given e-NFA
```

State	a	b	eps1	eps2
0	1	–	–	–
1	–	–	5	–
2	–	3	–	–
3	–	–	5	–
4	–	–	0	2
5	–	–	–	–

```
Resultant DFA
```

State	a	b
[0 2 4]	1	2
[1 5]	–	–
[3 5]	–	–

– indicates that there are no transitions.

Case 2: Output for the non-deterministic finite state automata table obtained from RE(a|b)

```
Given e-NFA
```

State	a	b	eps1	eps2
0	1	–	–	–
1	–	–	2	–
2	–	3	–	–
3	–	–	–	–

```
Resultant DFA
```

State	a	b
[0]	1	–
[1 2]	2	–
[3]	–	–

– indicates that there are no transitions.

Exercises for the Students

1. Verify the output for the given NFA.
2. Use this DFA transition table for recognizing simple words.
3. Use this DFA to recognize the tokens of various control statements such as if, goto, while, and for.
4. Use this DFA to identify the token representing the integers, real numbers, and numbers with exponents.

EXERCISE 4 Construct a predictive parsing table and parse the sentence.

Aim

To construct the predictive parser and parse the given sentence using top-down parsing.

Theory

Predictive parser is a recursive descent parser without backtracking, which follows top-down parsing approach. Non-recursive grammar is given as an input to the predictive parser. The main components of the predictive parser are stack, input buffer, parsing table, and parsing algorithm. In this exercise, the parsing table is constructed from the given grammar rules and used to parse the sentence for the syntactic checking of the source code. If the parsing is successful, the intermediate code is generated; otherwise an error message is displayed and the compiling process is terminated. The stack has the non-terminal that is expanded to match its contents with the input string. The parsing table is row-wise indexed by the non-terminal and column-wise indexed by the terminal. The cell in the parsing table has a rule that is used to produce the right-hand side of the production rule. If multiple rules are present in the cell, the grammar is ambiguous and has to be reworked.

Steps

1. First read the grammar and identify the terminal and the non-terminal along with the rule head.
2. Identify the first and follow.
3. For each rule, place the rule in the predictive parsing table row indexed by the non-terminal associated with the rule head and column indexed by the first of this non-terminal.
4. If only epsilon is there in the first of the non-terminal place, the rule in the predictive parsing table row is indexed by the non-terminal associated with the rule head and column indexed by the follows of this non-terminal.
5. Read the sentence and separate them into tokens.
6. Start with the start symbol and parse the sentence by repeatedly looking at the predictive parsing and replacing the non-terminal in the left-hand side of the rule by its right-hand side and matching with the tokens in the sentence.
7. Report any error if empty entries are encountered in the parsing table during the parsing process.

Modules

1. Reading the grammar rules
   ```
   readRules
   ```
2. Finding the first and follow of the non-terminals
   ```
   computeAllFirst
   computeFirst
   computeAllFollow
   ```
3. Constructing the predictive parsing table
   ```
   constructTable
   ```
4. Reading the sentence to be parsed
   ```
   readInput
   ```
5. Recognizing the sentence
   ```
   recognize
   ```
6. Other modules for displaying the table and intermediate results, checking the duplicates, mapping of the non-terminal and rule number, and so on.

Algorithm

1. Refer Section 4.4.3 and look at the algorithm *constructPredictiveTable, first, follow* for the construction of the predictive parsing table.
2. Refer the algorithm *Predictive_parser* in Section 4.4.3 for the recognition of the sentence using the predictive parser.

Data structure and prototype declarations

```
#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<stdlib.h>
#define MAXNAMESIZE 20
#define MAXSTACKSIZE 100
#define SENTENCESIZE 30
#define MAXRULES 100
#define MAXNTS 20
#define MAXTS 20
#define RH 1            // Rule head
#define RB 2            // Rule body
#define UNKNOWN -1
#define T 1
#define NT 2
#define testc {printf("\nTest and continue\n");}
#define teste {printf("\nTest and exit..\n");exit(-1);}
#include "sstack.h"
#include "sstack.cc"
typedef struct LNode
{
        char name[MAXNAMESIZE];
        int tOrNt;   // Terminal or non-terminal
        int rhOrRb;  // Rule head or rule body
        struct LNode *next;
} LNode;
typedef LNode *PtrToLNode ;
typedef PtrToLNode List;
class First
{
public:
        int ntIdx;
        int count;
        int maxTs;    // Maximum number of terminals
        char **names;
        First(int);
        void initFirst();
};

class PredParser
```

```
{
        int noRules;
        int noNTs;              // Number of non-terminals
        int noTs;               // Number of terminals
        char **input;
        char startSymbol[MAXNAMESIZE];
        int noInWords;
        int **parsingTable;
        char nts[MAXRULES][MAXNAMESIZE];
        char uniqueNTs[MAXNTS][MAXNAMESIZE]; // List of non-terminals
        char uniqueTs[MAXTS][MAXNAMESIZE];   // List of terminals
        int **ntRuleNumberMapping;
        List *rules;
        char ***first;
        char ***follow;
        int *firstCount;
        int *followCount;
        int *rulesVisited;
        char **leftHeadNames;
        char ***grammarSymbols;  // Grammar symbols in the right hand of rule
        int **grammarSymbolsType; // Grammar symbols type
        int *grammarSymbolCount;
public:
        PredParser();
        void readRules(FILE *);
        void readInput(FILE *);
        void dispInput();
        void constructTable();
        int isMemberInUniqueNTs(char *str);
        int isMemberInUniqueTs(char *str);
        int isPresent(int ntIdx, char *name, int count);
        int isPresentFollow(int ntIdx, char *name, int count);
        void addRuleIndexToNT(int ruleNo, char *ntName);
        void dispNTRuleNumberMapping();
        int indexOfNT(char *str);
        int indexOfT(char *str);
        void computeAllFirst();
        First computeFirst(char *);
        void computeAllFollow();
        void initRulesVisited();
        void dispRules();
        void dispNTs();
        void dispTs();
```

```
        void dispFirst();
        void dispFollow();
        void dispParsingTable();
        int recognize();
};
```

Source code listing

The implementation of the main modules and output are shown here.

Main function for constructing the parsing table and recognizing the sentence:

```
int main()
{
      int parsingStatus;
      PredParser pParser;
      FILE *fp,*fp1;
      char fname[30];
      fp = fopen("rules.txt","r");
      if(NULL == fp)
      {
          printf("%s:Error in file open.. Could not read rules from file\n");
          exit(-1);
      }
      fp1 = fopen("input.txt","r");
      if(NULL == fp1)
      {
          printf("%s:Error in file open.. Could not read sentence from file\n");
          exit(-1);
      }

      pParser.readRules(fp);
      pParser.readInput(fp1);
      pParser.dispInput();
      pParser.constructTable();
      pParser.dispNTs();
      pParser.dispTs();
      pParser.dispFirst();
      pParser.dispFollow();
      pParser.dispRules();
      pParser.dispParsingTable();
      pParser.dispNTRuleNumberMapping();
```

```
        parsingStatus = pParser.recognize();
        if(parsingStatus == 1)
            printf("Successful Parsing\n");
        else
            printf("Error in Parsing\n");

        return 1;
}
```

Output

The grammar specification (non-left recursive) is given as follows:

E : T E1;
E1 : + T E1;
E1 : eps;
T : F T1;
T1 : * F T1;
T1 : eps;
F : (E) ;
F : ***id***;
F : ***const***;

The output of the predictive parser along with the intermediate results is as follows:

List of non-terminals

E E1 T T1 F

List of terminals

\+ * ( ) ***id*** ***const*** #
First(E):→(***id const***
First(E1):→+ eps
First(T):→(***id const***
First(T1):→* eps
First(F):→(***id const***
Follow(E):→#)
Follow(E1):→#)
Follow(T):→+ #)
Follow(T1):→+ #)
Follow(F):→* + #)

Given grammar with rule number

0: E→ T E1
1: E1→ + T E1
2: E1→ eps
3: T→ F T1
4: T1→ * F T1
5: T1→ eps

6: F→ (E)
7: F→ ***id***
8: F→ ***const***

Parsing table

NT	+	*	(	)	id	const	#
E	-	-	0	-	0	0	-
E1	1	-	-	2	-	-	2
T	-	-	3	-	3	3	-
T1	5	4	-	5	-	-	5
F	-	-	6	-	7	8	-

All the entries in the parsing table are rule numbers to be referred during the parsing. For efficient storage, the rule names are labelled and dealt inside the program.

Non-terminal and rule number mapping

E→0
E1→1 2
T→3
T1→4 5
F→6 7 8

The code is tested with the following arithmetic expression:

(***id*** + ***id*** * ***id*** + ***id*** + ***id*** * ***id*** * ***id***) #

This expression is parsed successfully.

Exercises for the Students

1. Check the validity of the parsing table.
2. Use this parsing table to test some other arithmetic expressions.

EXERCISE 5 Construct the LR(0) parsing table from the given grammar rules.

Aim

To construct the LR(0) items and the **action** and **goto** tables.

Theory

LR parser is a bottom-up parser. L stands for left to right and R for the rightmost derivations in reverse. LR parser starts processing the given sentence of the language and keeps reducing the sentence till the start symbol is obtained. This parser detects the handle of the grammar in the sentence and replaces it with the left-hand side of the rule (non-terminal). LR parser consists of (a) input buffer, (b) stack, (c) parsing table, and (d) parsing algorithm. The parsing table is constructed from the given grammar and used for the recognition of the given sentence. The content of the input buffering is the input sentence to be parsed. The contents of the stack are the grammar symbol and the current state of the parser. The state on the top of the parser reflects the current status of the parser, and appropriate actions such as shift, reduce, accept, and error are taken.

Steps

1. Read the grammar rules (augmented grammar rules).
2. Compute first and follow of the non-terminals.
3. Create appropriate data structure for holding the rules and allocate space in the memory. (We have used Boolean vector to store the presence of the rules in the set of items and position vector to store the information about the dot position in the rule. This representation simplifies the storage process.)
4. Start with the augmented start symbol and find the closure of all the possible items. (Item refers to a grammar rule, and set of items refers to the possible derived items from the rule.)
5. Construct the set of items by deriving from the initial set of items by making a transition for each non-terminal and terminal.
6. Place the transitions from one set of items to another state for the non-terminal in the gotos table and terminal transition in the action table (for shift action).
7. If dot is at the last position of the rule, it is the rule for reduce action. Place this reduce action in the action table row indexed by the current state and column indexed by the follow of the rule head.
8. All empty entries are error.
9. Place accept state when the dot appears at the end of the symbol.

Modules

1. Reading grammar rules and allocating space in the memory
2. Computing first and follow (as done in a predictive parsing table)
3. Constructing a subset
4. Display modules
5. Other modules for checking duplicates state

Algorithm

Refer Section 4.5.3 for the algorithms.

```
ClosureLR0
GotoLR0
ConstructLR0ItemsSet
ConsturctSLRTable
```

Hints on the Development

The process involves the construction of the set of items. Hence, large numbers of rules are replicated with the dot positions at different locations. Hence in our program, we have represented the LR(0) item by two vectors (the dimension of the vector is the number of rules in the given problem). The first vector used is a binary vector, which tells whether the rule is present in the given LR(0) items. Another vector shows the position of the dot in the rule. There can be a case where one rule is associated with two positions in LR(0) item. This exercise illustrates such a case. Hence a structure called *POSITION* is used to hold two dot positions (first and second). We have used the word *items* to indicate all the

LR(0) items. The word *item* is used to represent any one LR(0) item. The position represents the place at which the dot is positioned to represent the state of parsing.

Data structures and prototype declaration

```
#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<stdlib.h>
#define MAXNAMESIZE 20
#define MAXSTACKSIZE 100
#define SENTENCESIZE 30
#define MAXRULES 100
#define MAXITEMS 100
#define MAXNTS 20
#define MAXTS 20
#define RH 1            // Rule head
#define RB 2            // Rule body
#define UNKNOWN -1
#define T 1
#define NT 2
#include "sstack.h"
#include "sstack.cc"
typedef struct LNode
{
        char name[MAXNAMESIZE];
        int tOrNt;   // Terminal or non-terminal
        int rhOrRb;  // Rule head or rule body
        struct LNode *next;
} LNode;

typedef LNode *PtrToLNode ;
typedef PtrToLNode List;

class First
{
public:
        int ntIdx;
        int count;
        int maxTs;
        char **names;
        First(int);
        void initFirst();
```

```
};

typedef struct POSITIONS
{
        int first;
        int second;
}POSITIONS;

typedef struct POSITION
{
        int first;
        int second;
}POSITION;

typedef struct ACTION
{
        char action;
        int state;
}ACTION;

class LRParser
{

        int noRules;
        int noItems;
        int noNTs;
        int noTs;
        int **gotos;
        ACTION **actions;
        char **input;
        int itemsCount;
        char startSymbol[MAXNAMESIZE];
        int noInWords;
        int **parsingTable;
        char nts[MAXRULES][MAXNAMESIZE];
        char uniqueNTs[MAXNTS][MAXNAMESIZE];
        char uniqueTs[MAXTS][MAXNAMESIZE];
        int **ntRuleNumberMapping;
        int **items;
        int *item;
        POSITIONS **positions;
        POSITION *position;
        List *rules;
```

```
        char ***first;
        char ***follow;
        int *firstCount;
        int *followCount;
        int *rulesVisited;
        int *itemsVisited;
        char **leftHeadNames;
        char ***grammarSymbols;
        int **grammarSymbolsType;
        int *grammarSymbolCount;
public:
        LRParser();
        void initItem();
        int validRule(int *);
        void initItemsProcessed();
        int isRulePresent(int, POSITION);
        int isItemPresent(int *itm, POSITION *posi);
        void readRules(FILE *);
        void readInput(FILE *);
        void dispInput();
        void constructTable();
        void constructSetOfItems();
        int isMemberInUniqueNTs(char *str);
        int isMemberInUniqueTs(char *str);
        int isPresent(int ntIdx, char *name, int count);
        int isPresentFollow(int ntIdx, char *name, int count);
        void addRuleIndexToNT(int ruleNo, char *ntName);
        void dispNTRuleNumberMapping();
        void dispActions();
        void dispGotos();
        int indexOfNT(char *str);
        int indexOfT(char *str);
        void computeAllFirst();
        First computeFirst(char *);
        void computeAllFollow();
        void closure(int rno, int p);
        void initRulesVisited();
        int allItemsAreVisited();
        void itemsProcessed();
        void dispRules();
        void dispNTs();
        void dispTs();
        void dispFirst();
```

```
        void dispFollow();
        void dispGrammarSymbols();
        void dispParsingTable();
        void dispItems();
        void dispItem(int *);
        void dispPosition(POSITION *);
        int recognize();
};
```

The implementation of the main modules and output are shown here.

Main function for constructing the parsing table

```
int main()
{
      int parsingStatus;
      LRParser lrParser;
      FILE *fp,*fp1;
      char fname[30];
      fp = fopen("rules1.txt","r");
      if(NULL == fp)
      {
            printf("Error in file open.. Could not read rules from file\n");
            exit(-1);
      }
      fp1 = fopen("input.txt","r");
      if(NULL == fp1)
      {
            printf("Error in file open.. Could not read sentence from file\n");
            exit(-1);
      }

      lrParser.readRules(fp);
      lrParser.readInput(fp1);
      lrParser.dispInput();
      lrParser.dispNTs();
      lrParser.dispTs();

      lrParser.dispRules();
      lrParser.dispParsingTable();
      lrParser.dispNTRuleNumberMapping();
```

```
        lrParser.dispGrammarSymbols();
        lrParser.constructSetOfItems();
        lrParser.dispFirst();
        lrParser.dispFollow();
        lrParser.constructTable();
        lrParser.dispItems();
        lrParser.dispActions();
        lrParser.dispGotos();

        return 1;
}
```

Output

Grammar specification:

E1 : E;
E : E + T;
E : T;
T : T * F;
T : F;
F : (E);
F : ***id***;

The program generates the following results with the intermediate code as follows:

List of non-terminals

E1 E T F

List of terminals

\+ * ( ) ***id*** #

Rules

```
0: E1→ E                    4: T→ F
1: E→ E + T                 5: F→ (E)
2: E→ T                     6: F→ id
3: T→ T * F
```

Mapping the non-terminal and the rules

```
E1→0      E→1 2      T→3 4      F→5 6
```

First

```
First(E1):→(id
First(E):→(id
First(T):→(id
First(F):→(id
```

Follow

```
Follow(E1):→#
Follow(E):→+ ) #
Follow(T):→* + ) #
Follow(F):→* + ) #
```

Set of items

```
ITEM: I0
        E1→.E
        E→.E + T
        E→.T
        T→.T * F
        T→.F
        F→.(E)
        F→.id

ITEM : I1
        E1→E .
        E→E .+ T

ITEM : I2
        E→T .
        T→T .* F

ITEM : I3
        T→F .

ITEM : I4
        E→.E + T
        E→.T
        T→.T * F
        T→.F
        F→.(.E)
        F→.id

ITEM : I5
        F→id .
ITEM : I6
        E→E + .T
        T→.T * F
        T→.F
```

```
        F→.(E)
        F→.id
ITEM : I7
        T→T * .F
        F→.(E)
        F→.id

ITEM : I8
        E→E .+ T
        F→(E .)

ITEM : I9
        E→E + T .
        T→T .* F

ITEM : I10
        T→T * F .

ITEM : I11
        F→( E ) .
```

Parsing table—Action

State	+	×	(	)	id	#
0:	–	–	s4	–	s5	–
1:	s6	–	–	–	–	acc
2:	r2	s7	–	r2	–	r2
3:	r4	r4	–	r4	–	r4
4:	–	–	–	–	s5	–
5:	r6	r6	–	r6	–	r6
6:	–	–	s4	–	s5	–
7:	–	–	s4	–	s5	–
8:	s6	–	–	s11	–	–
9:	r1	s7	–	r1	–	r1
10:	r3	r3	–	r3	–	r3
11:	r5	r5	–	r5	–	r5

Parsing Table—Goto

State	E	T	F
0:	1	2	3
1:	–	–	–
2:	–	–	–
3:	–	–	–
4:	8	2	3
5:	–	–	–
6:	–	9	3
7:	–	–	10
8:	–	–	–
9:	–	–	–
10:	–	–	–
11:	–	–	–

(– indicates error entries)

Exercises for the Students

1. Verify the construction of the set of items and the LR(0) parsing table.
2. In this exercise, the parsing table is constructed. Use the parsing table to parse an arithmetic expression as done for the predictive parser. Refer the algorithm *LR_Parsing* in Section 4.5.3.

EXERCISE 6 Use the Lex tool for the separation of token from a text.

Aim

To use the Lex tool to implement a lexical analyser.

Theory

The compilation process goes through a sequence of steps in a *predefined manner*. From the given lexical and syntactical specifications, the scanning and parsing operations can be carried out, as shown in Exercises 2–5. However, the process is repetitive in nature, and hence it is automated by developing a tool. The Lex tool is used to scan through the text and separate the tokens depending on the lexical specification in the file. If this Lex tool is used in the compilation process, the C program embedded into the action part of the lexical specification file returns the tokens to the parser. Lex tool can be used for other purposes such as text formatting, word separation, and frequency count of characters, words, and sentences. The output of the Lex tool is lex.yy.c, which consists of the scanning routines.

Steps

1. Specify the lexical patterns in the Lex specification file (file name extension is .l).

2. Compile the lexical specification file using the tool Lex, which produces a file lex.yy.c.
3. Include lex.yy.c in a C file and compile using C compiler producing an executable version of the scanner program.
4. Call the scanner program in C program using the function yylex().

Modules

1. Auxiliary definitions (if any)
2. Lexical pattern
3. The actions for displaying the tokens
4. The actions for returning the token to the parser

A sample lexical specification file, as reproduced from Exercise 3.23, is shown as follows with actions to display the type of operator:

```
%START BM
%%
[a-zA-Z][0-9a-zA-Z]*   {
                               BEGIN BM;
                               printf("%s is an id\n", yytext);
                       }
"+"                    {
                               BEGIN 0;
                               printf("+ is a binary operator\n");
                       }
"*"                    {
                               BEGIN 0;
                               printf("* is a binary operator\n");
                       }
[0-9]+                 {
                               BEGIN BM;
                               printf("%s is a constant\n");
                       }
 "-"                   {
                               BEGIN 0;
                               printf("- is a binary operator\n");
                       }
"-"                    {
                               printf("- is an unary operator\n");
                       }
%%
```

The introduction to the Lex tool with a simple program is available in Section 3.5.5 and the details of the usage of the Lex tool are given in Chapter 6 and Appendix A for representing the entire C language. Refer Example 3.24 for the count of the characters, words, and lines in a file and Example 3.25 for the replacement of the lower case letters by upper case letters.

EXERCISE 7 Simulate a numerical calculator using the tools Lex and Yacc.

Aim

To develop a calculator using Lex and Yacc tools.

Theory

The calculator program involves constant arithemetic expressions. If a syntactic specification of a constant arithemetic expression is given, the parser program implemented using Lex and Yacc parses the structure of the arithemetic expression and executes the semantic actions. In this exercise, the nature of the semantic actions is to evaluate the expression and display the result. It is similar to the interpreter of the language. It does not produce machine code. Any ambiguity issues in the grammar has to be resolved while specifying the grammar rules. The Lex tool generates a C file 'lex.yy.c' and the Yacc tool generates the C file 'y.tab.c'. These two files linked along with the user's file having the *main* function performs the appropriate calculator functions depending on the arithmetic sentence given to the program.

Steps

1. Specify the structure of the number and operators using lexical specification file.
2. Specify the structure of the constant arithmetic expression.
3. Compile the lexical and syntactic specification files using the tools Lex and Yacc.
4. Include the lex.yy.c and y.tab.c in a C program file and compile.
5. Call the function `yyparse()`, which in turn calls the `yylex()` for the token scanning.
6. Place the appropriate actions to do the arithmetic operations and display the results.

Modules

1. Lexical specification module
2. Syntax specification module
3. Actions module in Lex and Yacc files

A simple calculator program using Lex and Yacc, as reproduced from Example 4.19 is as follows:

Lexical specifications

```
%{
extern int yylval;
%}
%%
```

```
"+"     {    return plus;}
"*"     {    return mul;}
"\n"    {    return newline;}
"("     {    return openp;}
")"     {    return closep;}
[0-9]+       {
             yylval = atoi(yytext);
             return number;
        }
%%
yywrap()
{
        printf("eof reached\n");
        return 1;
}
```

Syntax specifications

```
%{
#include<stdio.h>
%}
%token plus mul newline
%token number
%%
lines   : lines line | line
        ;
line    : E newline          {printf("%d\n", $1);}
        ;
E       : E plus T           {$$ = $1 + $3;}
          | T                {$$ = $1;}
        ;
T       : T mul F            {$$ = $1 * $3;}
          |F                 {$$ = $1;}
        ;
F       : openp E closep     {$$ = $2;}
           |number           {$$ = $1;}
        ;
%%
yyerror()
{
        printf("error occurred\n");
        exit(-1);
}
```

Calling the parser

```
#include"y.tab.c"
#include"lex.yy.c"
int main()
{
        yyparse();
        return 1;
}
```

Exercise for the Students

Extend the above-mentioned program to simulate a scientific calculator.

EXERCISE 8 Construct a hash table for the storage and retrieval of the symbol.

Aim

To manage the symbols in the program and implement the various operations associated with the symbol table.

Theory

Numerous symbols (identifiers) are used in the declarative part of the statements and their use is referred in the executable statements. Hence a storage is required for storing and retrieving the symbols. The logical storage for storing these symbols can be a *set* of symbols, where *set* refers to the collection of elements without duplication. The set is viewed as a table, and hence we call it a symbol table. However, the data structure used for implementing the set (symbol table) varies as it can be a binary search tree or a B-Tree, or a Hash Map, and so on. The operations involved with the symbol table in compiler perspective are addition of symbols and reference of the symbol.

Steps

1. Read the symbols in the program during the scanning and parsing processes.
2. Define the hash and initialize the hash symbol table.
3. Add the symbol in the hash table (addition of the symbol will be done when the symbols are declared).
4. Look for the symbol in the hash table (look up or find the symbol will happen when the symbol is accessed inside the program in the executable statement).
5. If any value is to be assigned, assign the value to the symbol in the symbol table.

Modules

1. Initialization of the symbol table
2. Addition of a symbol
3. Finding of a symbol
4. Display of the symbol
5. Assignment of values to the symbol in the table

Data structures and prototype declaration

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define true 1
#define false 0
#define teste {printf("Test and exit\n");exit(-1);}
#define test {printf("Test\n");}
#define NAMESIZE 300
#define CHAR 1
#define INT 2
#define FLOAT 3
#define LONG 5
#define DOUBLE 4
#define STRING 6
typedef struct Symbol
{
        short int type;
        short int width;
        char name[40];
        union
        {
                char cVal;
                int iVal;
                float fVal;
                long lVal;
                double dVal;
        }value;
        char *sVal;
}Symbol;
typedef struct Node
{
        Symbol symbol;
        struct Node *next;
} Node;
typedef struct Node *Position;
typedef Position List;

class HSet
{
```

```
private:
        int hsize;
        List *HashTable;
public:
        HSet(int);
        int  addSymbol(Symbol symbol);
        int findHash(char*);
        void dispSymbolTable();
        void dispNode(Node *p);
        Node *findSymbol(char *name);
        int assignValue(char *name, char *value);
};
```

Implementation of symbol table manipulation routines

```
HSet::HSet(int size)
{
int ii;
        hsize = size;
        HashTable = (List*)malloc(sizeof(List) * hsize);
        if(HashTable == NULL) {printf("Memory allocation problem\n");
exit(-1);}
        for(ii = 0;ii < hsize;ii++)
        {
                HashTable[ii] = (Node*)malloc(sizeof(Node));
                if(HashTable[ii] == NULL) {printf("error in memory allocation
for hash table\n");exit(-1);}
                HashTable[ii]→next = NULL;
        }
}
void HSet::dispSymbolTable()
{
List L;
int ii;
        for(ii = 0;ii < hsize;ii++)
        {
                L = HashTable[ii];
                while(L→next != NULL)
                {
                        dispNode(L→next);
                        L = L→next;
                }
```

```
        }
}

void HSet::dispNode(Node *p)
{
        printf("%d\t%s\t", p→symbol.type, p→symbol.name);
        switch(p→symbol.type)
        {
                case CHAR:
                        printf("%c ", p→symbol.value.cVal);
                        break;
                case INT:
                        printf("%d ", p→symbol.value.iVal);
                        break;
                case FLOAT:
                        printf("%f ", p→symbol.value.fVal);
                        break;
                case LONG:
                        printf("%ld ", p→symbol.value.lVal);
                        break;
                case DOUBLE:
                        printf("%lf ", p→symbol.value.dVal);
                        break;
                case STRING:
                        printf("%s", p→symbol.sVal);
                        break;
        }
        printf("\n");
}

Node* HSet::findSymbol(char *name)
{
List L;
int idx;
        idx = findHash(name);
        L = HashTable[idx];
        while(L→next != NULL)
        {
                if(strcmp(name, L→next→symbol.name) == 0)
                        return L→next;
                L = L→next;
        }
        return NULL;
```

```
}

int HSet::assignValue(char *name, char value[])
{
        Node *p;
        char *str;
        p = findSymbol(name);
        if(p! = NULL)
        {
                switch(p→symbol.type)
                {
                        case CHAR:
                                p→symbol.value.cVal = *value;
                                break;
                        case INT:
                                p→symbol.value.iVal = atoi(value);
                                break;
                        case FLOAT:
                                p→symbol.value.fVal = atof(value);
                                break;
                        case LONG:
                                p→symbol.value.lVal = atol(value);
                                break;
                        case DOUBLE:
                                p→symbol.value.dVal = atof(value);
                                break;
                        case STRING:
                                str = (char*)malloc(strlen(value));
                                strcpy(str, value);
                                p→symbol.sVal = str;
                                break;
                }
        }
        else
        {
                printf("%s : Not found\n", name);
                return -1;
        }
        return 1;
}

int HSet::findHash(char *name)
{
```

```
int hashVal = 0;
        while(*name != '\0')
                hashVal += *name++;
        return hashVal % hsize;
}

int HSet::addSymbol(Symbol symbol)
{
Node *p,*sp;
int ret, idx;
char *str;
        idx = findHash(symbol.name);
        p = HashTable[idx];
        while(p→next != NULL)
        {
                ret = strcmp(symbol.name, p→next→symbol.name);
                if(ret == 0)
                {
                        printf("Symbol: %s is redefined\n", symbol.name);
                        return -1;
                }
                p = p->next;
        }
        sp = (Node*)malloc(sizeof(Node));
        if(NULL == sp)
        {
                printf("Error in memory allocations\n");
                exit(-1);
        }
        sp→symbol.type = symbol.type;
        strcpy(sp→symbol.name, symbol.name);
        sp→next = NULL;
        p→next = sp;
        return 1;
}
```

Using symbol table manipulation routines from the main program

```
int main()
{
HSet symbols(50);
Symbol sym;
```

```
int i;

        sym.type = FLOAT;
        strcpy(sym.name,"area");
        symbols.addSymbol(sym);
        symbols.assignValue("area","10.0");
        sym.type = INT;
        strcpy(sym.name,"perimeter");
        symbols.addSymbol(sym);
        symbols.assignValue("perimeter","200");
        sym.type = STRING;
        strcpy(sym.name,"College");
        symbols.addSymbol(sym);
        symbols.assignValue("College","Mepco Schlenk Engineering College");
        sym.type = STRING;
        strcpy(sym.name,"House");
        symbols.addSymbol(sym);
        symbols.assignValue("House","BG1 Staff Quarters");
        symbols.dispSymbolTable();
        return 1;
}
```

Output

```
3    area    10.000000
6    House   BG1 Staff Quarters
2    perimeter    200
6    College Mepco Schlenk Engineering College
```

Exercise for the Students

Replace the dummy attributes given for the symbol with the actual compiler-based symbol attributes.

EXERCISE 9 Generate the intermediate code (quadruples) from the given lexical and syntactical specifications for a programming language using Lex and Yacc tools.

Aim

To develop the semantic routine to generate the intermediate code.

Theory

The intermediate code bridges the syntax and semantics of the source code with the machine code. Some of the popular intermediate codes in the perspective of a compiler are quadruple, triple, indirect triple, postfix expression, and syntax tree. The standardized intermediate code will enable the programming languages to be ported to many computer platforms. Java byte code and common intermediate language (CIL) from Microsoft

are the examples of the standard intermediate code. To generate the intermediate code, semantic actions (implemented in an programming languages) must be written to convert the programming language constructs into the appropriate intermediate code. From the intermediate code, the machine code could be generated by mapping it to the machine code.

Steps

1. Identify the language constructs.
2. Write the syntactic specification and present it to the parser (a Lex- and Yacc-based parser may be used).
3. Compile the lexical and syntactic specifications of the selected language constructs such as arithmetic expression, control statement, and declarative statement.
4. Attach the semantic routines for each grammar rule so that the intermediate code could be generated.
5. Read the source program.
6. Parse the program.
7. Convert the source code into an intermediate code.

Modules

1. Lexical module
2. Syntactic module
3. Semantic routines module
4. Output module

In this exercise, we will highlight the generation of an intermediate code for the arithmetic expression. Only the modules related to the intermediate code generation are highlighted. For complete details, refer Chapter 6 and Appendix A.

The *data structure* for the quadruple is as follows:

```
typedef struct Quad{
        char *label;
        int operator;
        Attr *operand1;
        Attr * operand2;
        Attr *result;
        struct Quad *nextQuad;
}Quad;
```

The prototype declarations for the addition of a quadruple to the intermediate code are as follows:

```
void addCode(Quad *quadListHeader, char *label, int operator,  Attr operand1,
Attr operand2, Attr result);
```

The initialization of the code data structure is done as follows:

```
void createQuadList(Quad **quadListHeader)
{
        *quadListHeader = (Quad *)malloc(sizeof(Quad));
```

```
        (*quadListHeader)→label = NULL;
        (*quadListHeader)→operator = –1;
        (*quadListHeader)→operand1 = NULL;
        (*quadListHeader)→operand2 = NULL;
        (*quadListHeader)→result = NULL;
        (*quadListHeader)→nextQuad = NULL;
}
```

The routines for the implementation of the addition of quadruple are as follows:

```
void addCode(Quad *quadListHeader, char *label, int operator, char *operand1,
char *operand2, char *result)
{
        Quad *temp,*ptr;
        temp = quadListHeader;
        while(temp→nextQuad! = NULL)
                temp = temp→nextQuad;
        ptr = (Quad*)malloc(sizeof(Quad));
        ptr→label = (char *)malloc(strlen(label) + 1);
        ptr→operand1 = (char *)malloc(strlen(operand1) + 1);
        ptr→operand2 = (char *)malloc(strlen(operand2) + 1);
        ptr→result = (char *)malloc(strlen(result) + 1);
        ptr→nextQuad = NULL;
        strcpy(ptr→label, label);
        if(strcmp(label," ")! = 0)
                strcpy(label," ");
        ptr→operator = operator;
        strcpy(ptr→operand1, operand1);
        strcpy(ptr→operand2, operand2);
        strcpy(ptr→result, result);
        temp->nextQuad = ptr;
}
```

Whenever an expression is encountered, a temporary variable is created. The module for creating the temporary variable is as follows:

```
void createTemp(char temp[])
{
        static int i = 0;
        char no[5];
        i++;
```

```
        itoa(i, no);
        strcpy(temp,"t");
        strcat(temp, no);
}
```

The syntactic specification and semantic actions for the generation of quadruple (intermediate code) are as follows:

```
assignStmt: assignExpr _semicolon
;

Expr     : Expr _plus Expr {
                        createTemp($$);
                        addCode(quadTable, labelpending, PLUS, $1, $3, $$);
                        }
   | Expr _minus Expr{
                        createTemp($$);
                        addCode(quadTable, labelpending, MINUS, $1, $3, $$);
                        }

   | Expr _mul Expr {
                        createTemp($$);
                        addCode(quadTable, labelpending, MUL, $1, $3, $$);
                        }
   | Expr _div Expr {
                        createTemp($$);
                        addCode(quadTable, labelpending, DIV, $1, $3, $$);
                        }
   | Expr _modulo Expr {
                              createTemp($$);
                              addCode(quadTable, labelpending, MOD, $1, $3,
$$);
                        }
   | _uminus Expr       {
                        createTemp($$);
                        addCode(quadTable, labelpending, UMINUS, $2," ", $$);
                        }
   | Expr _lt Expr      {
                        createTemp($$);
                        addCode(quadTable, labelpending, LT, $1, $3, $$);
                        }
```

```
| Expr _le Expr       {
                      createTemp($$);
                      addCode(quadTable, labelpending, LE, $1, $3, $$);
                      }
| Expr _ge Expr       {
                      createTemp($$);
                      addCode(quadTable, labelpending, GE, $1, $3, $$);
                      }
| Expr _gt Expr       {
                      createTemp($$);
                      addCode(quadTable, labelpending, GT, $1, $3, $$);
                      }
| Expr _dequal Expr   {
                      createTemp($$);
                      addCode(quadTable, labelpending, EQ, $1, $3, $$);
                      }
| Expr _unequal Expr  {
                      createTemp($$);
                      addCode(quadTable, labelpending, NE, $1, $3, $$);
                      }
| Expr _or Expr       {
                      createTemp($$);
                      addCode(quadTable, labelpending, OR, $1, $3, $$);
                      }
| Expr _and Expr      {
                      createTemp($$);
                      addCode(quadTable, labelpending, AND, $1, $3, $$);
                      }
| _leftp Expr _rightp {strcpy($$, $2);}
| _id
          {
          if(findSymbolHash($1) == NULL)
          {
                    printf("%s: %d:Error %s: Undeclared
                    Identifier\n", srcFileName, lineNo – 1, $1);
                    errCount++;
          }
          strcpy($$, $1);
          }
| _num
      {
```

```
            itoa($1, str);
            strcpy($$, str);
        }
    ;
```

The routines to display the quadruples are as follows:

```
void printCode(Quad *quadListHeader)
{
      Quad *temp;
      temp = quadListHeader→nextQuad;
      printf(" THE TABLE OF QUADRUPLES ARE\n\n");
      printf("LABEL\tOPER\tOP1\tOP2\tRES\n\n");
      while(temp! = NULL)
      {
          if(strcmp(temp→label," "))
                printf("%s:\t", temp→label);
          else
                printf("\t");
          printf("%s\t%s\t%s\t%s\n", ops[temp→operator], temp→operand1,
          temp→operand2, temp→result);
          temp = temp→nextQuad;
      }
}
```

Output

The input code segment is as follows:

```
noOfYears = 10
interestRate = 0.12
principalAmount = 1000
totalAmount = principalAmount + interestRate * noOfYears
```

The set of quadruples generated are as follows:

```
LABEL   OPER    OP1              OP2          RES
        =       10                            noOfYears
        =       0.12                          interestRate
        =       1000                          principalAmount
        *       interestRate     noOfYears    t1
        +       principalAmount  t1           t2
        =       t2                            totalAmount
```

Exercise for the Students

Write down the semantic routines for generating the intermediate code (quadruple) for a Boolean expression.

EXERCISE 10 Generate the machine code from the given intermediate code.

Aim

To generate the machine for the Intel 8086 architecture for a simple case.

Theory

Machine codes are the ones that are finally loaded into the main memory of the computer for execution. Hence the machine code is dependent on the target machine. To generate the machine code, knowledge of the hardware architecture of the target machine, addressing modes, instruction, and so on is essential. It involves the bit and byte codes of the target machine. The code generation process involves allocation and reallocation of the registers in the processor of the target machine and mapping of the intermediate code with the machine code.

Steps

1. Initialize the register descriptor as an array of characters.
2. Read the quadruple in the format `op`, `operand1`, `operand2`, and `result`.
3. For every quadruple entry, perform the following:
   (a) if an operand is already an available required register, then use Load the operand in one register;
   (b) else;
   (c) obtain the free register and load the operand into the register;
   (d) load the second operand into another register;
   (e) call the appropriate code generation routine depending on the operator.

Algorithm

An outline of the code generation procedure is given Section 8.5.2. The following algorithm presents the code generation process for a simple move and arithmetic operations.

Algorithm `genCode(Q)`

```
// List of quadruples
//returns the 8086 assembly language code equivalent.
{
        for each quadruple Q in quadruples list do
        {
                R1 = register for operand1
                genCode("Mov", R1, operand1)
                R2 = register for operand2
                genCode("Mov", R2, operand2)
```

```
                genCode(operator, R1, R2)
                genCode("Mov"result, R1)
        }
}
```

Source code listing for machine code generation for a simple arithmetic expression

```
#include<stdio.h>
#include<string.h>
#define noop 6

typedef struct quad
{
  char operand1[10];
  char operand2[10];
  char operator[10];
  char result[10];
}quad;

char *oparray[] = {"+","-","*","/","uminus","="};
quad readQuadruples(FILE*);
void genCode(quad);

void add(quad);
void sub(quad);
void mul(quad);
void div(quad);
void neg(quad);
void ass(quad);

void add(quad q1)
{
        printf("MOV AX, %s\n", q1.operand1);
        printf("MOV BX, %s\n", q1.operand2);
        printf("ADD AX, BX\n");
        printf("MOV %s, AX\n", q1.result);
}
void sub(quad q1)
{
        printf("MOV AX, %s\n", q1.operand1);
        printf("MOV BX, %s\n", q1.operand2);
        printf("SUB AX, BX\n");
```

```
        printf("MOV %s, AX\n", q1.result);
}

void mul(quad q1)
{
        printf("MOV AX, %s\n", q1.operand1);
        printf("MOV BX, %s\n", q1.operand2);
        printf("MUL BX\n");
        printf("MOV %s, AX\n", q1.result);
}

void div(quad q1)
{
        printf("MOV AX, %s\n", q1.operand1);
        printf("MOV BX, %s\n", q1.operand2);
        printf("DIV BX\n");
        printf("MOV %s, AX\n", q1.result);
}

void neg(quad q1)
{
        printf("MOV AX, %s\n", q1.operand1);
        printf("NEG AX\n");
        printf("MOV %s, AX\n", q1.result);
}
void assi(quad q1)
{
        printf("MOV AX, %s\n", q1.operand1);
        printf("MOV %s, AX\n", q1.result);
}

void(*opname[])(quad) = {add, sub, mul, div, neg, assi};
quad readQuadruples(FILE*fp)
{
        quad q1;
        fscanf(fp,"%s\t%s\t%s\t%s\n", q1.oprator, q1.operand1, q1.operand2,
q1.re   sult);
        printf("The given quadruple is: %s = %s %s %s\n", q1.result,
q1.operand1, q1.oprator, q1.operand2);
        return q1;
}

void genCode(quad q1)
```

```
{
        void(*ptof)(quad);
        int i;

        // Find the index of the operator in the array of operator names.
        for(i = 0;i < noop;i++)
        if(strcmp(oparray[i], q1.oprator) == 0)
        break;

        if(i == noop)
        {
        printf("Invalid Operator");
        return;
        }
        // Call the semantic routine that has the name of the operator using
        //pointer to functions.
        ptof = opname[i];
        (*ptof)(q1);
}

int main()
{
        FILE *fp;
        quad q1;

        fp = fopen("ari.txt","r");

        do
        {
                q1 = readQuadruples(fp);
                genCode(q1);
        } while(!feof(fp));
        fclose(fp);
}
```

Output

```
The given quadruple is: c = a + b
MOV AX, a
MOV BX, b
ADD AX, BX
MOV c, AX
```

```
The given quadruple is: r = p – q
MOV AX, p
MOV BX, q
SUB AX, BX
MOV r, AX
The given quadruple is: z = x * y
MOV AX, x
MOV BX, y
MUL BX
MOV z, AX
```

We have generated the assembly code for the processor. However, the machine code has to be generated for the given program. Generation of machine code for the 8086 processor is dealt in Section 8.5.2.

Exercise for the Students

Experiment with other expression and other programming language constructs by suitably modifying the program.

FLEX YOUR BRAIN: MINI PROJECT

Consider a language of your choice to do the following task.

1. Identify the tokens and their specifications.
2. Identify the syntactic structure of the language.
3. Specify both lexical and syntactic structures of the language constructs using Lex and Yacc.
4. Adapt the procedure as given in Chapter 6 and Appendix A or your own procedure for writing the semantic routines for the generation of an intermediate code.
5. Consider a simple expression for the conversion of an intermediate code into an assembly code.
6. Use the methods given in Section 8.5.3 for the conversion of the assembly code into a target machine code for the 8086 processor.
7. Identify the exe format of a target processor such as 8086 and fill the header portion.
8. Write the machine code into a binary file following the exe header format.
9. Execute the exe file and debug if required.