# Compiler Design Lab

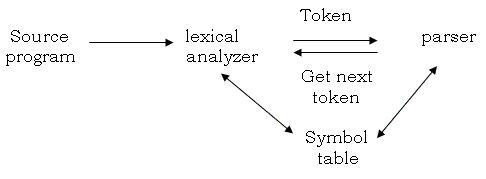
**Aim:**

**Design and implement a lexical analyzer using LEX**

To design and implement a lexical analyzer for given language using LEX tool.

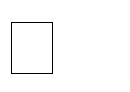
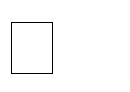
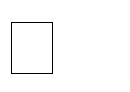
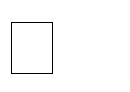
**Description:**

Compiler is responsible for converting high level language in machine language. There are several phases involved in this and lexical analysis is the first phase.

Lexical analyzer reads the characters from source code and convert it into tokens.

Different tokens or lexemes are:

Keywords Identifiers Operators Constants



Consider an example.

**c = a + b;**

After lexical analysis a symbol table is generated as given below.

Token Type

c identifier

= operator

1. identifier

+ operator

1. identifier

; separator

**Program:**

// Program name as “lexicalfile.l”

%{

#include<stdio.h>

%}

delim [\t]

ws {delim}+ letter [A-Za-z] digit [0-9]

id {letter}({letter}|{digit})\*

num {digit}+(\.{digit}+)?(E[+/-]?{digit}+)?

%%

ws {printf("no action");}

if|else|then {{printf("%s is a keyword",yytext);} // TYPE 32 KEYWORDS

{id} {printf("%s is a identifier",yytext);}

{num} {printf(" it is a number");}

"<" {printf("it is a relational operator less than");}

"<=" {printf("it is a relational operator less than or equal");} ">" {printf("it is a relational operator greater than");}

">=" {printf("it is a relational operator greater than");} "==" {printf("it is a relational operator equal");}

"<>" {printf("it is a relational operator not equal");}

%%

main()

{

yylex();

}

**Result**

lex lexicalfile.l gcc lex.yy.c -ll

if

if is a keyword number

number is a identifier 254

It is a number

<>

it is a relational operator not equal

Expt:3.a

**Aim:**

**Program to recognize a valid arithmetic expression**

To implement Program to recognize a valid arithmetic expression that uses operator +, – , \* and /.

**Program:**

#### Lex code ( l.lex )

%{#include "y.tab.h"

%}

%%

[0-9]+ { return NUMBER; }

[a-zA-Z][a-zA-Z0-9]\* { return ID; }

\n { return 0; }

. { return yytext[0]; }

%%

#### YACC code ( y.yacc)

%{

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

%}

%token NUMBER ID

%left '+''-''\*''/'

%%

exp : exp'+'exp

| exp'-'exp

| exp'\*'exp ((8-7)\*(9/3))

| exp'/'exp

| '('exp')'

| NUMBER

| ID ;

%%

int main(int argc, char \*argv[]) { printf("Enter the expression: "); yyparse();

printf("Valid Expression!\n"); return 0;

}

int yyerror() {

printf("Invalid Expression!\n"); exit(1);

}

**Execution steps:**

yacc -dy yacpg.y

lex arith.l

gcc lex.yy.c y.tab.c -lfl

./a.out

**Output:**

./a.out

Enter the expression: 2 + 3 Valid Expression!

./a.out

Enter the expression: 2 3 - Invalid Expression!

Expt:3.b

**Aim:**

**Program to recognize a valid identifier**

To design and implement program to recognize a valid variable which starts with a letter followed by any number of letters or digits..

**Description:**

Step1: Start the program Step2: Reading an expression

Step3: Checking the validating of the given expression according to the rule using yacc. Step4: Using expression rule print the result of the given values

Step5: Stop the program

**Program:**

/\*Definition Section\*/

%{

#include<stdio.h>

%}

%%

([a-zA-Z][0-9])+|[a-zA-Z]\* {printf("Identifier\n");}

%%

int yywrap()

{

return 1;

}

int main(void)

{

yylex(); return 0;

}

**Output:**

Input: var

Output: Identifier

Input: 123

Output: Not a Identifier

Expt:3.c

**Aim:**

**IMPLEMENTATION OF A CALCULATOR**

To implement a calculator using LEX and YACC.

**Description:**

Step1: A Yacc source program has three parts as follows: Declarations %% translation rules %% supporting C routines

Step2: Declarations Section: This section contains entries that:

* 1. Include standard I/O header file.
  2. Define global variables.
  3. Define the list rule as the place to start processing.
  4. Define the tokens used by the parser. v. Define the operators and their precedence.

Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

Step4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.

Step5: Main- The required main program that calls the yyparse subroutine to start the program.

Step6: yyerror(s) -This error-handling subroutine only prints a syntax error message.

Step7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.

Step8: calc.lex contains the rules to generate these tokens from the input stream.

**Program:**

%{

#define YYSTYPE double #include "y.tab.h"

#include <stdlib.h>

%}

white [ \t]+ digit [0-9] integer {digit}+

%%

{white} { }

{integer} { yylval=atof(yytext); return NUMBER;

}

"+" return PLUS;

"-" return MINUS; "\*" return MULT;

"/" return DIVIDE; "(" return LEFT; ")" return RIGHT; "\n" return END;

%%

%{

#include <math.h> #include <stdio.h>

#include <stdlib.h> #define YYSTYPE double

%}

%token NUMBER

%token PLUS MINUS MULT DIVIDE

%token LEFT RIGHT

%token END

%left PLUS MINUS TIMES DIVIDE

%s start Input Line Expression

%%

Input:

| Input Line

;

Line:

END

| Expression END { printf("Result: %f\n", $$); }

;

Expression:

NUMBER {$$=$1; }

| Expression PLUS Expression { $$=$1+$3; }

| Expression MINUS Expression { $$=$1-$3; }

| Expression MULT Expression { $$=$1\*$3; }

| Expression DIVIDE Expression { $$=$1/$3; }

%%

int yyerror(char \*s) { printf("%s\n", s);

}

int main() {

if (yyparse())

fprintf(stderr, "Successful parsing.\n"); else

fprintf(stderr, "error found.\n");

}

**Output**

yacc -dy a.y lex a.l

gcc lex.yy.c y.tab.c -lfl

./a.out 4+7

11

xpt:4

**Write program to find ε – closure of all states of any given NFA with ε transition.**

**Aim:**

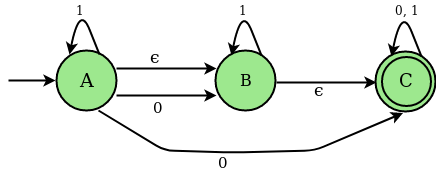
To Convert the BNF rules into YACC form and write code to generate abstract syntax tree

**Description:**

**Description:**

***Non-determinestic Finite Automata (NFA) :*** NFA is a finite automaton where for some cases when a single input is given to a single state, the machine goes to more than 1 states,

i.e. some of the moves cannot be uniquely determined by the present state and the present input symbol.



Q → Finite non-empty set of states.

∑ → Finite non-empty set of input symbols.

∂ → Transitional Function. q0 → Beginning state.

F → Final State

**NFA with (null) or ∈ move :** If any finite automata contains ε (null) move or transaction, then that finite automata is called NFA with ∈ moves

Example :

Consider the following figure of NFA with ∈ move :

An NFA can be represented as M = { Q, ∑, ∂, q0, F}

**Transition state table for the above NFA**

|  |  |  |  |
| --- | --- | --- | --- |
| **STATES** | **0** | **1** | **EPSILON** |
| **A** | **B, C** | **A** | **B** |
| **B** | **–** | **B** | **C** |
| **C** | **C** | **C** | **–** |

Epsilon (∈ ) – closure : Epsilon closure for a given state X is a set of states which can be reached from the states X with only (null) or ε moves including the state X itself. In other words, ε-closure for a state can be obtained by union operation of the ε-closure of the states which can be reached from X with a single ε move in recursive manner.

For the above example ∈ closure are as follows :

∈ closure(A) : {A, B, C}

∈ closure(B) : {B, C}

∈ closure(C) : {C}

**Deterministic Finite Automata (DFA) :** DFA is a finite automata where, for all cases, when a single input is given to a single state, the machine goes to a single state, i.e., all the moves of the machine can be uniquely determined by the present state and the present input symbol.

**Steps to Convert NFA with ε-move to DFA :**

**Step 1 :** Take ∈ closure for the beginning state of NFA as beginning state of DFA.

**Step 2 :** Find the states that can be traversed from the present for each input symbol

(union of transition value and their closures for each states of NFA present in current state of DFA).

**Step 3 :** If any new state is found take it as current state and repeat step 2. **Step 4 :** Do repeat Step 2 and Step 3 until no new state present in DFA transition table. **Step 5 :** Mark the states of DFA which contains final state of NFA as final states of DFA.

**Program:**

#include <stdio.h> #include <stdlib.h> #include <string.h> #define MAX\_LEN 100

char NFA\_FILE[MAX\_LEN];

char buffer[MAX\_LEN]; int zz = 0;

// Structure to store DFA states and their

// status ( i.e new entry or already present) struct DFA {

char \*states; int count;

} dfa;

int last\_index = 0; FILE \*fp;

int symbols;

/\* reset the hash map\*/

void reset(int ar[], int size) { int i;

// reset all the values of

// the mapping array to zero for (i = 0; i < size; i++) { ar[i] = 0;

}

}

// Check which States are present in the e-closure

/\* map the states of NFA to a hash set\*/ void check(int ar[], char S[]) {

int i, j;

// To parse the individual states of NFA int len = strlen(S);

for (i = 0; i < len; i++) {

// Set hash map for the position

// of the states which is found j = ((int)(S[i]) - 65);

ar[j]++;

}

}

// To find new Closure States

void state(int ar[], int size, char S[]) { int j, k = 0;

// Combine multiple states of NFA

// to create new states of DFA for (j = 0; j < size; j++) {

if (ar[j] != 0)

S[k++] = (char)(65 + j);

}

// mark the end of the state S[k] = '\0';

}

// To pick the next closure from closure set int closure(int ar[], int size) {

int i;

// check new closure is present or not for (i = 0; i < size; i++) {

if (ar[i] == 1)

***svnce*** 25

### return i;

}

return (100);

}

// Check new DFA states can be

// entered in DFA table or not

int indexing(struct DFA \*dfa) { int i;

for (i = 0; i < last\_index; i++) { if (dfa[i].count == 0)

return 1;

}

return -1;

}

/\* To Display epsilon closure\*/

void Display\_closure(int states, int closure\_ar[], char \*closure\_table[],

char \*NFA\_TABLE[][symbols + 1], char \*DFA\_TABLE[][symbols]) { int i;

for (i = 0; i < states; i++) { reset(closure\_ar, states); closure\_ar[i] = 2;

// to neglect blank entry

if (strcmp(&NFA\_TABLE[i][symbols], "-") != 0) {

// copy the NFA transition state to buffer strcpy(buffer, &NFA\_TABLE[i][symbols]); check(closure\_ar, buffer);

int z = closure(closure\_ar, states);

// till closure get completely saturated while (z != 100)

{

if (strcmp(&NFA\_TABLE[z][symbols], "-") != 0) { strcpy(buffer, &NFA\_TABLE[z][symbols]);

// call the check function check(closure\_ar, buffer);

}

closure\_ar[z]++;

z = closure(closure\_ar, states);

}

}

// print the e closure for every states of NFA printf("\n e-Closure (%c) :\t", (char)(65 + i)); bzero((void \*)buffer, MAX\_LEN); state(closure\_ar, states, buffer); strcpy(&closure\_table[i], buffer); printf("%s\n", &closure\_table[i]);

/\* To check New States in DFA \*/

int new\_states(struct DFA \*dfa, char S[]) { int i;

// To check the current state is already

// being used as a DFA state or not in

// DFA transition table

for (i = 0; i < last\_index; i++) {

if (strcmp(&dfa[i].states, S) == 0) return 0;

}

// push the new strcpy(&dfa[last\_index++].states, S);

// set the count for new states entered

// to zero

dfa[last\_index - 1].count = 0; return 1;

}

// Transition function from NFA to DFA

// (generally union of closure operation )

void trans(char S[], int M, char \*clsr\_t[], int st, char \*NFT[][symbols + 1], char TB[]) {

int len = strlen(S); int i, j, k, g;

int arr[st]; int sz; reset(arr, st);

char temp[MAX\_LEN], temp2[MAX\_LEN]; char \*buff;

// Transition function from NFA to DFA for (i = 0; i < len; i++) {

j = ((int)(S[i] - 65));

strcpy(temp, &NFT[j][M]);

if (strcmp(temp, "-") != 0) { sz = strlen(temp);

g = 0;

while (g < sz) {

k = ((int)(temp[g] - 65)); strcpy(temp2, &clsr\_t[k]); check(arr, temp2);

g++;

}

bzero((void \*)temp, MAX\_LEN); state(arr, st, temp);

***svnce*** 28

### if (temp[0] != '\0') { strcpy(TB, temp);

} else strcpy(TB, "-");

}

/\* Display DFA transition state table\*/

void Display\_DFA(int last\_index, struct DFA \*dfa\_states, char \*DFA\_TABLE[][symbols]) {

int i, j; printf("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\ n\n");

printf("\t\t DFA TRANSITION STATE TABLE \t\t \n\n"); printf("\n STATES OF DFA :\t\t");

for (i = 1; i < last\_index; i++) printf("%s, ", &dfa\_states[i].states); printf("\n");

printf("\n GIVEN SYMBOLS FOR DFA: \t");

for (i = 0; i < symbols; i++) printf("%d, ", i);

printf("\n\n"); printf("STATES\t");

for (i = 0; i < symbols; i++) printf("|%d\t", i);

printf("\n");

// display the DFA transition state table printf(" + \n"); for (i = 0; i < zz; i++) {

printf("%s\t", &dfa\_states[i + 1].states); for (j = 0; j < symbols; j++) { printf("|%s \t", &DFA\_TABLE[i][j]);

}

printf("\n");

}

}

// Driver Code int main() { int i, j, states;

char T\_buf[MAX\_LEN];

// creating an array dfa structures

struct DFA \*dfa\_states = malloc(MAX\_LEN \* (sizeof(dfa))); states = 6, symbols = 2;

printf("\n STATES OF NFA :\t\t"); for (i = 0; i < states; i++) printf("%c, ", (char)(65 + i)); printf("\n");

printf("\n GIVEN SYMBOLS FOR NFA: \t");

for (i = 0; i < symbols; i++) printf("%d, ", i);

printf("eps");

printf("\n\n");

char \*NFA\_TABLE[states][symbols + 1];

// Hard coded input for NFA table

char \*DFA\_TABLE[MAX\_LEN][symbols]; strcpy(&NFA\_TABLE[0][0], "FC");

strcpy(&NFA\_TABLE[0][1], "-");

strcpy(&NFA\_TABLE[0][2], "BF");

strcpy(&NFA\_TABLE[1][0], "-");

strcpy(&NFA\_TABLE[1][1], "C");

strcpy(&NFA\_TABLE[1][2], "-");

strcpy(&NFA\_TABLE[2][0], "-");

strcpy(&NFA\_TABLE[2][1], "-");

strcpy(&NFA\_TABLE[2][2], "D");

strcpy(&NFA\_TABLE[3][0], "E");

strcpy(&NFA\_TABLE[3][1], "A");

strcpy(&NFA\_TABLE[3][2], "-");

strcpy(&NFA\_TABLE[4][0], "A");

strcpy(&NFA\_TABLE[4][1], "-");

strcpy(&NFA\_TABLE[4][2], "BF");

strcpy(&NFA\_TABLE[5][0], "-");

strcpy(&NFA\_TABLE[5][1], "-");

strcpy(&NFA\_TABLE[5][2], "-");

printf("\n NFA STATE TRANSITION TABLE \n\n\n"); printf("STATES\t");

for (i = 0; i < symbols; i++) printf("|%d\t", i);

printf("eps\n");

// Displaying the matrix of NFA transition table printf(" + \n"); for (i = 0; i < states; i++) {

printf("%c\t", (char)(65 + i)); for (j = 0; j <= symbols; j++) {

printf("|%s \t", &NFA\_TABLE[i][j]);

}

printf("\n");

}

int closure\_ar[states];

char \*closure\_table[states];

Display\_closure(states, closure\_ar, closure\_table, NFA\_TABLE, DFA\_TABLE);

strcpy(&dfa\_states[last\_index++].states, "-"); dfa\_states[last\_index - 1].count = 1; bzero((void \*)buffer, MAX\_LEN); strcpy(buffer, &closure\_table[0]);

strcpy(&dfa\_states[last\_index++].states, buffer); int Sm = 1, ind = 1;

int start\_index = 1;

// Filling up the DFA table with transition values

// Till new states can be entered in DFA table while (ind != -1) { dfa\_states[start\_index].count = 1;

Sm = 0;

for (i = 0; i < symbols; i++) {

trans(buffer, i, closure\_table, states, NFA\_TABLE, T\_buf);

// storing the new DFA state in buffer strcpy(&DFA\_TABLE[zz][i], T\_buf);

// parameter to control new states

Sm = Sm + new\_states(dfa\_states, T\_buf);

}

ind = indexing(dfa\_states); if (ind != -1)

strcpy(buffer, &dfa\_states[++start\_index].states); zz++;

}

// display the DFA TABLE Display\_DFA(last\_index, dfa\_states, DFA\_TABLE); return 0;

}

**Input :**

9

2

* - BH
* - CE

D - -

* - G
* F -
* - G
* - BH

I - -

- - -

**Output :**

STATES OF NFA : A, B, C, D, E, F, G, H, I, GIVEN SYMBOLS FOR NFA: 0, 1, eps NFA STATE TRANSITION TABLE

STATES |0 |1 eps

+ A |- |- |BH

B |- |- |CE C |D |- |- D |- |- |G E |- |F |- F |- |- |G

G |- |- |BH H |I |- |-

I |- |- |-

e-Closure (A) : ABCEH e-Closure (B) : BCE

e-Closure (C) : C

e-Closure (D) : BCDEGH e-Closure (E) : E

e-Closure (F) : BCEFGH e-Closure (G) : BCEGH e-Closure (H) : H

e-Closure (I) : I

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DFA TRANSITION STATE TABLE

STATES OF DFA : ABCEH, BCDEGHI, BCEFGH, GIVEN SYMBOLS FOR DFA: 0, 1,

STATES |0 |1

+ ABCEH |BCDEGHI |BCEFGH BCDEGHI |BCDEGHI |BCEFGH BCEFGH |BCDEGHI |BCEFGH

Expt:5

**PROGRAM TO CONVERT NFA to DFA**

**Aim:**

To Convert NFA to DFA

**Description:**

Let **X = (Qx, ∑, δx, q0, Fx)** be an NDFA which accepts the language L(X). We have to design an equivalent DFA **Y = (Qy, ∑, δy, q0, Fy)** such that **L(Y) = L(X)**. The following procedure converts the NDFA to its equivalent DFA −

***Algorithm***

**Input** − An NDFA

**Output** − An equivalent DFA

**Step 1** − Create state table from the given NDFA.

**Step 2** − Create a blank state table under possible input alphabets for the equivalent DFA.

**Step 3** − Mark the start state of the DFA by q0 (Same as the NDFA).

**Step 4** − Find out the combination of States {Q0, Q1,... , Qn} for each possible input alphabet.

**Step 5** − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

**Step 6** − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

**Program:**

#include <cstdio> #include <fstream> #include <iostream> #include <bitset> #include <vector> #include <cstring> #include <cstdlib> #include <algorithm> #include <queue> #include <set>

#define MAX\_NFA\_STATES 10

#define MAX\_ALPHABET\_SIZE 10 using namespace std;

// Representation of an NFA state class NFAstate

{

public:

int transitions[MAX\_ALPHABET\_SIZE][MAX\_NFA\_STATES]; NFAstate()

{

for (int i = 0; i < MAX\_ALPHABET\_SIZE; i++) for (int j = 0; j < MAX\_NFA\_STATES; j++) transitions[i][j] = -1;

}

}\*NFAstates;

// Representation of a DFA state struct DFAstate

{

bool finalState;

bitset<MAX\_NFA\_STATES> constituentNFAstates; bitset<MAX\_NFA\_STATES> transitions[MAX\_ALPHABET\_SIZE]; int symbolicTransitions[MAX\_ALPHABET\_SIZE];

***svnce*** 45

### };

set<int> NFA\_finalStates; vector<int> DFA\_finalStates; vector<DFAstate\*> DFAstates; queue<int> incompleteDFAstates;

int N, M; // N -> No. of stattes, M -> Size of input alphabet

// finds the epsilon closure of the NFA state "state" and stores it into "closure" void epsilonClosure(int state, bitset<MAX\_NFA\_STATES> &closure)

{

for (int i = 0; i < N && NFAstates[state].transitions[0][i] != -1; i++) if (closure[NFAstates[state].transitions[0][i]] == 0)

{

closure[NFAstates[state].transitions[0][i]] = 1; epsilonClosure(NFAstates[state].transitions[0][i], closure);

}

}

// finds the epsilon closure of a set of NFA states "state" and stores it into "closure"

void epsilonClosure(bitset<MAX\_NFA\_STATES> state, bitset<MAX\_NFA\_STATES> &closure)

{

for (int i = 0; i < N; i++) if (state[i] == 1) epsilonClosure(i, closure);

}

// returns a bitset representing the set of states the NFA could be in after moving

// from state X on input symbol A

void NFAmove(int X, int A, bitset<MAX\_NFA\_STATES> &Y)

{

for (int i = 0; i < N && NFAstates[X].transitions[A][i] != -1; i++) Y[NFAstates[X].transitions[A][i]] = 1;

}

// returns a bitset representing the set of states the NFA could be in after moving

// from the set of states X on input symbol A

void NFAmove(bitset<MAX\_NFA\_STATES> X, int A, bitset<MAX\_NFA\_STATES> &Y)

{

for (int i = 0; i < N; i++) if (X[i] == 1)

NFAmove(i, A, Y);

}

int main()

{

int i, j, X, Y, A, T, F, D;

// read in the underlying NFA ifstream fin("NFA.txt");

fin >> N >> M;

NFAstates = new NFAstate[N]; fin >> F;

for (i = 0; i < F; i++)

{fin >> X;

NFA\_finalStates.insert(X);

} fin >> T; while (T--)

{

fin >> X >> A >> Y;

for (i = 0; i < Y; i++)

{

fin >> j; NFAstates[X].transitions[A][i] = j;

}

}

fin.close();

// construct the corresponding DFA D = 1;

DFAstates.push\_back(new DFAstate); DFAstates[0]->constituentNFAstates[0] = 1;

epsilonClosure(0, DFAstates[0]->constituentNFAstates); for (j = 0; j < N; j++)

if (DFAstates[0]->constituentNFAstates[j] == 1 && NFA\_finalStates.find(

1. != NFA\_finalStates.end())

{

DFAstates[0]->finalState = true; DFA\_finalStates.push\_back(0); break;

}

incompleteDFAstates.push(0);

while (!incompleteDFAstates.empty())

{

X = incompleteDFAstates.front(); incompleteDFAstates.pop();

for (i = 1; i <= M; i++)

{

NFAmove(DFAstates[X]->constituentNFAstates, i, DFAstates[X]->transitions[i]); epsilonClosure(DFAstates[X]->transitions[i], DFAstates[X]->transitions[i]);

for (j = 0; j < D; j++)

if (DFAstates[X]->transitions[i]

== DFAstates[j]->constituentNFAstates)

{

DFAstates[X]->symbolicTransitions[i] = j; break;

}

if (j == D)

{

DFAstates[X]->symbolicTransitions[i] = D; DFAstates.push\_back(new DFAstate); DFAstates[D]->constituentNFAstates

= DFAstates[X]->transitions[i]; for (j = 0; j < N; j++)

if (DFAstates[D]->constituentNFAstates[j] == 1

&& NFA\_finalStates.find(j) != NFA\_finalStates.end())

{

DFAstates[D]->finalState = true; DFA\_finalStates.push\_back(D); break;

}

incompleteDFAstates.push(D); D++;

} } }

// write out the corresponding DFA ofstream fout("DFA.txt");

fout << D << " " << M << "\n" << DFA\_finalStates.size(); for (vector<int>::iterator it = DFA\_finalStates.begin(); it

!= DFA\_finalStates.end(); it++) fout << " " << \*it;

fout << "\n";

for (i = 0; i < D; i++)

{

for (j = 1; j <= M; j++) fout << i << " " << j << " "

<< DFAstates[i]->symbolicTransitions[j] << "\n";

}

fout.close(); return 0;

}

**Output:**

./a.out Input file

NFA.txt 4 2

2 0 1

4

0 1 2 1 2

1 1 2 1 2

2 2 2 1 3

3 1 2 1 2

Output file

DFA.txt 4 2

3 0 1 3

0 1 1

0 2 2

1 1 1

1 2 3

2 1 2

2 2 2

3 1 1

3 2 2

Expt:6

**OPERATOR PRECEDENCE PARSING**

**Aim:**

To develop an operator precedence parser for a given language.

**Description:**

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

* No R.H.S. of any production has a∈.
* No two non-terminals are adjacent.

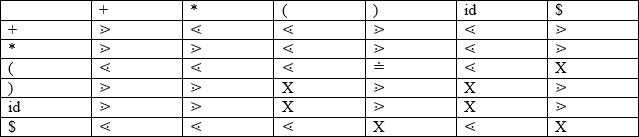
Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

**There are the three operator precedence relations:**

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b". a ≐ b means that the terminal "a" and "b" both have same precedence.

**Precedence table:**



**Parsing Action**

* Both end of the given input string, add the $ symbol.
* Now scan the input string from left right until the ⋗ is encountered.
* Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
* Everything between left most ⋖ and right most ⋗ is a handle.
* $ on $ means parsing is successful.

**Program:**

#include<stdio.h> #include<conio.h> void main()

{

char stack[20],ip[20],opt[10][10][1],ter[10]; inti,j,k,n,top=0,row,col;

clrscr(); for(i=0;i<10;i++)

{

stack[i]=NULL; ip[i]=NULL;

for(j=0;j<10;j++)

{ opt[i][j][1]=NULL;

}

}

printf("Enter the no.of terminals:"); scanf("%d",&n);

printf("\nEnter the terminals:"); scanf("%s",ter);

printf("\nEnter the table values:\n"); for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

printf("Enter the value for %c %c:",ter[i],ter[j]); scanf("%s",opt[i][j]);

}

}

printf("\nOPERATOR PRECEDENCE TABLE:\n");

for(i=0;i<n;i++)

{

printf("\t%c",ter[i]);

}

printf("\n printf("\n");

");

for(i=0;i<n;i++)

{

printf("\n%c |",ter[i]); for(j=0;j<n;j++)

{

printf("\t%c",opt[i][j][0]);

}

}

stack[top]='$';

printf("\n\nEnter the input string(append with $):"); scanf("%s",ip);

i=0;

printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n"); printf("\n%s\t\t\t%s\t\t\t",stack,ip);

while(i<=strlen(ip))

{

for(k=0;k<n;k++)

{

if(stack[top]==ter[k]) row=k; if(ip[i]==ter[k]) col=k;

}

if((stack[top]=='$')&&(ip[i]=='$'))

{

printf("String is ACCEPTED"); break;

}

else if((opt[row][col][0]=='<') ||(opt[row][col][0]=='='))

{

stack[++top]=opt[row][col][0]; stack[++top]=ip[i];

ip[i]=' ';

printf("Shift %c",ip[i]); i++;

}

else

{

if(opt[row][col][0]=='>')

{

while(stack[top]!='<')

{

--top;

}

top=top-1; printf("Reduce");

}

else

{

printf("\nString is not accepted"); break;

}

}

printf("\n"); printf("%s\t\t\t%s\t\t\t",stack,ip);

}

getch();

}

**OUTPUT**

Enter the no.of terminals:4 Enter the terminals:i+\*$ Enter the table values:

Enter the value for i i:- Enter the value for i +:> Enter the value for i \*:> Enter the value for i $:> Enter the value for + i:< Enter the value for + +:> Enter the value for + \*:< Enter the value for + $:> Enter the value for \* i:< Enter the value for \* +:> Enter the value for \* \*:> Enter the value for \* $:> Enter the value for $ i:< Enter the value for $ +:< Enter the value for $ \*:< Enter the value for $ $:-

OPERATOR PRECEDENCE TABLE:

i + \* $

i | - > > >

+ | < > < >

\* | < > > >

$ | < < < -

Enter the input string(append with $):i+i\*i$

STACK INPUT STRING ACTION

$ i+i\*i$ Shift

$<i +i\*i$ Reduce

$<i +i\*i$ Shift

$<+ i\*i$ Shift

$<+<i \*i$ Reduce

$<+<i \*i$ Shift

$<+<\* i$ Shift

$<+<\*<i $ Reduce

$<+<\*<i $ Reduce

$<+<\*<i $ Reduce

$<+<\*<i $ String is ACCEPTED

Expt:7

**FIRST AND FOLLOW OF ANY GIVEN GRAMMAR.**

**Aim:**

To develop a program to find simulate first and follow of any given grammar.

**Description:**

**first(x) for all grammar symbols x**

Apply following rules:

* 1. If X is terminal, FIRST(X) = {X}.
  2. If X → ε is a production, then add ε to FIRST(X).
  3. If X is a non-terminal, and X → Y1 Y2 … Yk is a production, and ε is in all of FIRST(Y1), …, FIRST(Yk), then add ε to FIRST(X).
  4. If X is a non-terminal, and X → Y1 Y2 … Yk is a production, then add a to FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1), …, FIRST(Yi-1).
  5. Applying rules 1 and 2 is obvious. Applying rules 3 and 4 for FIRST(Y1 Y2 … Yk) can be done as follows:
  6. Add all the non-ε symbols of FIRST(Y1) to FIRST(Y1 Y2 … Yk). If ε ∈ FIRST(Y1), add all the non-ε symbols of FIRST(Y2). If ε ∈ FIRST(Y1) and ε ∈ FIRST(Y2), add all the non-ε symbols of FIRST(Y3), and so on. Finally, add ε to FIRST(Y1 Y2 … Yk) if ε ∈ FIRST(Yi), for all 1 ≤ i ≤ k.

## Example:

Consider the following grammar.

E → E + T | T

T → T \* F | F F → (E) | id

Grammar after removing left recursion:

E → TX

1. → +TX | ε T → FY
2. → \*FY | ε F → (E) | id

For the above grammar, following the above rules, the FIRST sets could be computed as

follows:

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(X) = {+, ε}

FIRST(Y) = {\*, ε}

## follow(a) for all non-terminals a

Apply the following rules:

1. If $ is the input end-marker, and S is the start symbol, $ ∈ FOLLOW(S).
2. If there is a production, A → αBβ, then (FIRST(β) – ε) ⊆ FOLLOW(B).
3. If there is a production, A → αB, or a production A → αBβ, where ε ∈ FIRST(β), then FOLLOW(A) ⊆ FOLLOW(B).

Note that unlike the computation of FIRST sets for non-terminals, where the focus is on what a non-terminal generates, the computation of FOLLOW sets depends upon where the non- terminal appears on the RHS of a production.

## Example:

For the above grammar, the FOLLOW sets can be computed by applying the above rules as follows.

FOLLOW(E) = {$, )}

FOLLOW(E) ⊆ FOLLOW(X) [in other words, FOLLOW(X) contains FOLLOW(E)] Since there is no other rule applicable to FOLLOW(X),

FOLLOW(X) = {$, )}

FOLLOW(T) ⊆ FOLLOW(Y) …. (1)

(FIRST(X) – ε) ⊆ FOLLOW(T) i.e., {+} ⊆ FOLLOW(T) …. (2) Also, since ε ∈ FIRST(X), FOLLOW(E) ⊆ FOLLOW(T)

i.e., {$, )} ⊆ FOLLOW(T) …. (3)

Putting (2) and (3) together, we get:

FOLLOW(T) = {$, ), +}

Since, there is no other rule applying to FOLLOW(Y), from (1), we get: FOLLOW(Y) = {$, ), +}

Since ε ∈ FIRST(Y), FOLLOW(T) ⊆ FOLLOW(F) and FOLLOW(Y) ⊆ FOLLOW(F).

Also, (FIRST(Y) – ε) ⊆ FOLLOW(F). Putting all these together: FOLLOW(F) = FOLLOW(T) ∪ FOLLOW(Y) ∪ (FIRST(Y) – ε) = {$, ), +, \*}

**Program:**

**a) FIRST** #include<stdio.h> #include<ctype.h>

void FIRST(char[],char ); void result(char[],char); intnop;

char prod[10][10]; void main()

{

int i;

char choice; char c;

char res1[20]; clrscr();

printf("How many number of productions ? :"); scanf(" %d",&nop);

printf("enter the production string like E=E+T\n"); for(i=0;i<nop;i++)

{

printf("Enter productions Number %d : ",i+1); scanf(" %s",prod[i]);

}

do

{

printf("\n Find the FIRST of :"); scanf(" %c",&c); memset(res1,’0’,sizeof(res)); FIRST(res1,c);

printf("\n FIRST(%c)= { ",c);

for(i=0;res1[i]!='\0';i++)

printf(" %c ",res1[i]);

***svnce*** 57

### printf("}\n");

printf("press 'y' to continue : "); scanf(" %c",&choice);

}

while(choice=='y'||choice =='Y');

}

void FIRST(char res[],char c)

{

inti,j,k;

char subres[5]; int eps; subres[0]='\0';

res[0]='\0'; memset(res,’0’,sizeof(res));

memset(subres,’0’,sizeof(res)); if(!(isupper(c)))

{

result(res,c); return ;

}

for(i=0;i<nop;i++)

{

if(prod[i][0]==c)

{

if(prod[i][2]=='$')

result(res,'$'); else

{ j=2;

while(prod[i][j]!='\0')

{

eps=0; FIRST(subres,prod[i][j]);

for(k=0;subres[k]!='\0';k++) result(res,subres[k]); for(k=0;subres[k]!='\0';k++) if(subres[k]=='$')

***svnce*** 58

### {

eps=1; break;

}

if(!eps) break; j++;

}

}

}

}

return ;

}

void result(char res[],char val)

{

int k;

for(k=0 ;res[k]!='\0';k++) if(res[k]==val)

return;

res[k]=val; res[k+1]='\0';

}

**OUTPUT:**

How many number of productions ?:8 enter the production string like E=E+T Enter productions Number 1 : E=TX Enter productions Number 2 : X=+TX Enter productions Number 3 : X=$ Enter productions Number 4 : T=FY Enter productions Number 5 : Y=\*FY Enter productions Number 6 : Y=$ Enter productions Number 7 : F=(E) Enter productions Number 8 : F=i Find the FIRST of :X

FIRST(X)= { + $ }

press 'y' to continue : Y Find the FIRST of :F FIRST(F)= { ( i }

press 'y' to continue : Y Find the FIRST of :Y FIRST(Y)= { \* $ }

press 'y' to continue : Y Find the FIRST of :E FIRST(E)= { ( i }

press 'y' to continue : Y Find the FIRST of :T FIRST(T)= { ( i }

press 'y' to continue : N

**FOLLOW**

#include<stdio.h> #include<string.h> intnop,m=0,p,i=0,j=0; char prod[10][10],res[10]; void FOLLOW(char c); void first(char c);

void result(char); void main()

{

int i;

int choice; charc,ch;

printf("Enter the no.of productions: "); scanf("%d", &nop);

printf("enter the production string like E=E+T\n"); for(i=0;i<nop;i++)

{

printf("Enter productions Number %d : ",i+1); scanf(" %s",prod[i]);

}

do

{ m=0;

memset(res,’0’,sizeof(res)); printf("Find FOLLOW of -->"); scanf(" %c",&c);

FOLLOW(c);

printf("FOLLOW(%c) = { ",c); for(i=0;i<m;i++)

printf("%c ",res[i]);

printf(" }\n");

printf("Do you want to continue(Press 1 to continue. )?");

scanf("%d%c",&choice,&ch);

}

while(choice==1);

}

void FOLLOW(char c)

{

if(prod[0][0]==c)

result('$'); for(i=0;i<nop;i++)

{

for(j=2;j<strlen(prod[i]);j++)

{

if(prod[i][j]==c)

{

if(prod[i][j+1]!='\0')

first(prod[i][j+1]); if(prod[i][j+1]=='\0'&&c!=prod[i][0]) FOLLOW(prod[i][0]);

}

}

}

}

void first(char c)

{

int k; if(!(isupper(c))) result(c); for(k=0;k<nop;k++)

{

if(prod[k][0]==c)

{

if(prod[k][2]=='$')

FOLLOW(prod[i][0]);

else if(islower(prod[k][2])) result(prod[k][2]);

else first(prod[k][2]);

}

}

}

void result(char c)

{

int i;

for( i=0;i<=m;i++) if(res[i]==c) return; res[m++]=c;

}

**OUTPUT:**

Enter the no.of productions: 8

enter the production string like E=E+T Enter productions Number 1 : E=TX Enter productions Number 2 : X=+TX Enter productions Number 3 : X=$ Enter productions Number 4 : T=FY Enter productions Number 5 : Y=\*FY Enter productions Number 6 : Y=$ Enter productions Number 7 : F=(E) Enter productions Number 8 : F=i Find FOLLOW of -->X

FOLLOW(X) = { $ ) }

Expt:8

**Construct a recursive descent parser for an expression.**

**Aim:**

To construct a recursive descent parser for an expression.

**Description:**

1. For each non-terminal symbol in the grammar implement corresponding parsing function (so e.g. to parse E we'll have `function E() {}`)
2. If there are several alternative productions for a non-terminal, implement each sub- production as a sub-function, and try them in order If some of the sub-productions succeeds, the main non-terminal is then restore the cursor\* (do backtracking) to the beginning of that sub-production, and try parsing next sub-production.

So for E we going to implement E1 for T, and E2 for T + E, and call E1 and E2 in order from main E:

E = E1 || E2; E1 = T;

E2 = T + E;

1. Implement function that checks for the presence of a needed token at the current cursor position in the source code. We'll call this function term(...)`, and e.g. to check terminal "a" in the source stream at cursor position, we do term("a"). After the check (positive or negative)

it also advances the cursor to the next token.

### **Program:** #include<stdio.h> #include<string.h> char input[10];

int i=0,error=0; void E();

void T(); voidEprime(); voidTprime(); void F();

void main()

{

clrscr();

printf("Enter an arithmetic expression :\n"); gets(input);

E();

if(strlen(input)==i&&error==0) printf("\nAccepted..!!!");

else printf("\nRejected..!!!"); getch();

}

void E()

{ T();

Eprime();

}

voidEprime()

{

if(input[i]=='+')

{ i++; T();

***svnce*** 63

### Eprime();

}

}

void T()

{ F();

Tprime();

}

voidTprime()

{

if(input[i]=='\*')

{ i++; F();

Tprime();

}

}

void F()

{

if(input[i]=='(')

{ i++; E();

if(input[i]==')') i++;

}

else if(isalpha(input[i]))

{ i++;

while(isalnum(input[i])||input[i]=='\_') i++;

}

else error=1;

} OUTPUT 1)

Enter an arithmetic expression : sum+month\*interest Accepted..!!!

2)

Enter an arithmetic expression : sum+avg\*+interest Rejected..!!!

Expt:9

**SHIFT REDUCE PARSER.**

**Aim:**

To construct a Shift Reduce Parser for a given language.

**Description:**

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.

This parser requires some data structures i.e. A input buffer for storing the input string.

A stack for storing and accessing the production rules.

**Basic Operations –**

Shift: This involves moving of symbols from input buffer onto the stack.

Reduce: If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.

Accept: If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.

Error: This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

**Program:**

#include"stdio.h" #include"stdlib.h" #include"string.h"

char ip\_sym[15],stack[15]; int ip\_ptr=0,st\_ptr=0,len,i; char temp[2],temp2[2]; char act[15];

void check(); void main()

{

printf("\n\t\t SHIFT REDUCE PARSER\n"); printf("\n GRAMMER\n");

printf("\n E->E+E\n E->E/E");

printf("\n E->E\*E\n E->a/b"); printf("\n enter the input symbol:\t"); gets(ip\_sym);

printf("\n\t stack implementation table"); printf("\n stack\t\t input symbol\t\t action"); printf("\n \t\t \t\t \n"); printf("\n $\t\t%s$\t\t\t--",ip\_sym); strcpy(act,"shift ");

temp[0]=ip\_sym[ip\_ptr]; temp[1]='\0'; strcat(act,temp); len=strlen(ip\_sym); for(i=0;i<=len-1;i++)

{

stack[st\_ptr]=ip\_sym[ip\_ptr]; stack[st\_ptr+1]='\0'; ip\_sym[ip\_ptr]=' ';

ip\_ptr++;

printf("\n $%s\t\t%s$\t\t\t%s",stack,ip\_sym,act); strcpy(act,"shift ");

temp[0]=ip\_sym[ip\_ptr]; temp[1]='\0'; strcat(act,temp); check();

st\_ptr++;

}

check();

}

void check()

{

int flag=0; temp2[0]=stack[st\_ptr]; temp2[1]='\0'; if((isalpha(temp2[0])))

{

stack[st\_ptr]='E';

printf("\n $%s\t\t%s$\t\t\tE->%s",stack,ip\_sym,temp2); flag=1;

}

if((!strcmp(temp2,"+"))||(!strcmp(temp2,"\*"))||(!strcmp(temp2,"/")))

{

flag=1;

}

if((!strcmp(stack,"E+E"))||(!strcmp(stack,"E/E"))||(!strcmp(stack,"E\*E")))

{

if(!strcmp(stack,"E+E"))

{

strcpy(stack,"E");

printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip\_sym);

}

else if(!strcmp(stack,"E/E"))

{

strcpy(stack,"E");

printf("\n $%s\t\t %s$\t\t\tE->E/E",stack,ip\_sym);

}

else

{

strcpy(stack,"E");

printf("\n $%s\t\t%s$\t\t\tE->E\*E",stack,ip\_sym);

}

flag=1; st\_ptr=0;

}

if(!strcmp(stack,"E")&&ip\_ptr==len)

{

printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip\_sym); exit(0);

}

if(flag==0)

{

printf("\n $%s\t\t%s$\t\t\tReject",stack,ip\_sym); exit(0);

}

return;

}

**OUTPUT:**

1)

SHIFT REDUCE PARSER GRAMMER E->E+E

E->E/E E->E\*E E->E-E

E->id

enter the input symbol: a+b\*c stack implementation table stack input symbol action

$ a+b\*c$ --

$a +b\*c$ shift a

$E +b\*c$ E->a

$E+ b\*c$ shift +

$E+b \*c$ shift b

$E+E \*c$ E->b

$E \*c$ E->E+E

$E\* c$ shift \*

$E\*c $ shift c

$E\*E $ E->c

$E $ E->E\*E

$E $ ACCEPT

1. SHIFT REDUCE PARSER GRAMMER E->E+E

E->E/E E->E\*E E->E-E

E->id

enter the input symbol: a+b\*+c stack implementation table stack input symbol action

$ a+b\*+c$ --

$a +b\*+c$ shift a

$E +b\*+c$ E->a

$E+ b\*+c$ shift +

$E+b \*+c$ shift b

$E+E \*+c$ E->b

$E \*+c$ E->E+E

$E\* +c$ shift \*

$E\*+ c$ shift +

$E\*+c $ shift c

$E\*+E $ E->c

$E\*+E reject

Expt:10

**LOOP UNROLLING.**

**Aim:**

To develop Program to perform loop unrolling.

**Description:**

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program’s speed by eliminating loop control instruction and loop test instructions.

**Advantages:**

Increases program efficiency. Reduces loop overhead.

If statements in loop are not dependent on each other, they can be executed in parallel.

**Program:**

#include<stdio.h> int main()

{

int i; for(i=0;i<10;i+=2)

{

printf("fun(%d)\n",i+1);

printf("fun(%d)\n",i+2);

}

}

**OUTPUT**

fun(1)

fun(2)

fun(3)

fun(4)

fun(5)

fun(6)

fun(7)

fun(8)

fun(9) fun(10)

Expt:11

**CONSTANT PROPAGATION**

**Aim:**

To develop Program to perform constant propagation.

**Description:**

Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

Example:

In the code fragment below, the value of x can be propagated to the use of x.

x = 3;

y = x + 4;

Below is the code fragment after constant propagation and constant folding. x = 3;

y = 7;

**Program**

#include<stdio.h> int main()

{

int x, y, z; x = 10;

y = x + 45;

z = y + 4;

printf("The value of z = %d", z); return 0;

}

**OUTPUT:**

$ vi test.c

$ cc –c –S test.c

$ vi test.s //before optimization assembly code

main:

pushl %ebp

movl %esp, %ebp andl $-16, %esp subl $32, %esp movl $10, 20(%esp)

movl 20(%esp), %eax addl $45, %eax

movl %eax, 24(%esp) movl 24(%esp), %eax addl $4, %eax

movl %eax, 28(%esp) movl $.LC0, %eax movl 28(%esp), %edx movl %edx, 4(%esp) movl %eax, (%esp) call printf

movl $0, %eax leave

ret

$ cc –c –S -O2 test.c

$ vi test.s //after optimization assembly code main:

pushl %ebp

movl %esp, %ebp andl $-16, %esp subl $16, %esp movl $59, 4(%esp) movl $.LC0, (%esp) call printf

xorl %eax, %eax leave

ret

Expt:12

**A THREE-ADDRESS CODE**

**Aim:**

To Implement three address code generator.

**Description:**

A statement involving no more than three references(two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

Example – The three address code for the expression a + b \* c + d :

T 1 = b \* c

T 2 = a + T 1 T 3 = T 2 + d

T 1 , T 2 , T 3 are temporary variables.

**Program**

%{

#include"y.tab.h"

%}

%%

[0-9]+ {yylval.dval=(\*yytext);return NUM;}

\n return 0;

. return yytext[0];

%%

%{

#include<stdio.h> #include<stdlib.h> char p='A';

%}

%union {char dval;}

%token NUM sign

%left '+''-'

%left '\*''/'

%type <dval> S

%type <dval> E

%%

S:E {printf("X=%c\n",$$);}

E:NUM {}|E'+'E {printf("%c=%c+%c\n",p,$1,$3);$$=p;p++;}

|E'-'E {printf("%c=%c-%c\n",p,$1,$3);$$=p;p++;}

|E'\*'E {printf("%c=%c\*%c\n",p,$1,$3);$$=p;p++;}

|E'/'E {printf("%c=%c/%c\n",p,$1,$3);$$=p;p++;}

%%

int main()

{

printf("Enter the expression : "); yyparse();

printf("Expression is valid\n"); return 0;

}

int yyerror()

{

printf("Expression is invalid \n"); exit(0);

}

**Sample input and output:**

>>yacc -dy 3\_address\_code.y

>>lex 3\_address\_code.l

>>gcc lex.yy.c y.tab.c -lfl

>>./a.out

Enter the expression : 2+3\*6-9/2 A=3\*6

B=2+A C=9/2 D=B-C X=D

Expt:13

**BACK END OF THE COMPILER**

**Aim:**

To Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.

### **Program:** #include<stdio.h> #include<conio.h> #include<string.h> #include<ctype.h> #include<graphics.h> typedef struct

{ char var[10]; int alive;

}

regist;

regist preg[10];

void substring(char exp[],int st,int end)

{ int i,j=0;

char dup[10]=""; for(i=st;i<end;i++) dup[j++]=exp[i];

dup[j]='0'; strcpy(exp,dup);

} int getregister(char var[])

{ int i; for(i=0;i<10;i++)

{

if(preg[i].alive==0)

{s trcpy(preg[i].var,var); break;

}}

return(i);

}

void getvar(char exp[],char v[])

{ int i,j=0;

char var[10]=""; for(i=0;exp[i]!='\0';i++) if(isalpha(exp[i]))

var[j++]=exp[i]; else

break; strcpy(v,var);

}

void main()

{ char basic[10][10],var[10][10],fstr[10],op; int i,j,k,reg,vc,flag=0;

clrscr();

printf("\nEnter the Three Address Code:\n"); for(i=0;;i++)

{

gets(basic[i]); if(strcmp(basic[i],"exit")==0) break;

}

printf("\nThe Equivalent Assembly Code is:\n"); for(j=0;j<i;j++)

{

getvar(basic[j],var[vc++]); strcpy(fstr,var[vc-1]);

substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j])); getvar(basic[j],var[vc++]);

reg=getregister(var[vc-1]); if(preg[reg].alive==0)

{

printf("\nMov R%d,%s",reg,var[vc-1]); preg[reg].alive=1;

}

op=basic[j][strlen(var[vc-1])]; substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j])); getvar(basic[j],var[vc++]);

switch(op)

{ case '+': printf("\nAdd"); break;

case '-': printf("\nSub"); break; case '\*': printf("\nMul"); break; case '/': printf("\nDiv"); break;

}

flag=1; for(k=0;k<=reg;k++)

{ if(strcmp(preg[k].var,var[vc-1])==0)

{

printf("R%d, R%d",k,reg); preg[k].alive=0;

flag=0; break;

}} if(flag)

{

printf(" %s,R%d",var[vc-1],reg); printf("\nMov %s,R%d",fstr,reg);

}s

trcpy(preg[reg].var,var[vc-3]); getch();

}}

**INPUT & OUTPUT:**

Enter the Three Address Code: a=b+c

c=a\*c exit

The Equivalent Assembly Code is: Mov R0,b

Add c,R0 Mov a,R0 Mov R1,a Mul c,R1 Mov c,R1

**ADDITIONAL LAB PROGRAMS**

Expt:14

**Aim:**

**Implementation of DFA using LEX**

To implement a DFA which accepts even number of 0’s and 1’s.

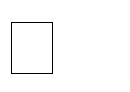
**Finite Automata**

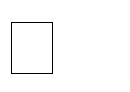
A finite automaton is an abstract, mathematical machine, also known as a finite state machine, with the following components:

* 1. A set of states S
  2. A set of input symbols E (the alphabet)
  3. A transition function move(state, symbol) : new state(s)
  4. A start state S0
  5. A set of final states F

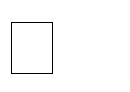
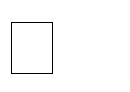
For a deterministic finite automaton (DFA), the function move(state, symbol) goes to at most one state, and symbol is never epsilon.

Finite automata correspond in a 1:1 relationship to transition diagrams; from any transition diagram one can write down the formal automaton in terms of items #1-#5 above, and vice versa. To draw the transition diagram for a finite automaton:

 draw a circle for each state s in S; put a label inside the circles to identify each state by number or name

 draw an arrow between Si and Sj, labeled with x whenever the transition says to move(Si,

x) : Sj

 draw a "wedgie" into the start state S0 to identify it  draw a second circle inside each of the final states in F

**Program:**

%{

%}

A

initial

%s A B C 1

%%

<INITIAL>0 BEGIN B; 0 0 0 0

<INITIAL>1 BEGIN A; 1

<A>0 BEGIN C;

<A>1 BEGIN INITIAL; 1

B

C

<B>0 BEGIN INITIAL;

<B>1 BEGIN C;

<C>0 BEGIN A; 1100

<C>1 BEGIN B;

<INITIAL>\n BEGIN INITIAL; printf("Accepted\n");

<A>\n BEGIN A; {printf("NOT ACCEPTED\n");}{return 0;}

<B>\n BEGIN B; {printf("NOT ACCEPTED\n");}{return 0;}

<C>\n BEGIN C; {printf("NOT ACCEPTED\n");}{return 0;}

%%

void main()

{

printf(“Enter the expression”);

yylex();

}

**Output:**

lex prpg11.l gcc lex.yy.c

./a.out

1000

Not Accepted

hello INVALID 01010101

Accepted

Expt:15

**Aim:**

**Convert lowercase characters to upper case in a file**

Program to convert lowercase characters to upper case in a file using LEX

**Program:**

%{

FILE \*fp;

FILE \*fp1;

%}

%%

[a-z] fprintf(fp1,"%c",toupper(yytext[0]));

. fprintf(fp1,"%C",yytext[0]);

%%

int main(int args,char \*\*argv)

{

fp1=fopen("b.txt","w"); if(!fp1)

{

printf("file not exists\n"); return(0);

}

fp=fopen("a.txt","r"); if(!fp)

{

printf("file not exists\n"); return(0);

}

yyin=fp1; yyin=fp; yylex(); return(0);

}

**Output:**

Contents of a.txt hello world

After execution of program contents of b.txt

HELLO WORLD

Expt:16

**Aim:**

**PROGRAM TO ACCEPT THE LANGUAGE anb**

To implement a program to accept the language anb n>0

**Program**

%{

#include "y.tab.h" #include<stdlib.h>

%}

%%

“a”{return A;} “b”{return B;}

“\n” return END;

%%

YACC

%{

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

%}

%token A B END

%s stmt T T1 S Eb

%% E aE/a

Stmt:

|T END {printf(“Accepted \n”); exit(0);}

;

T:T1 B;

T1: AT1|A;

%%

int main()

{

printf("Enter the string : ");

yyparse();

printf("valid string\n"); return 0;

}

int yyerror()

{

printf("invalid\n ");

}

## Output:

yacc -dy a.y lex a.l

gcc lex.yy.c y.tab.c -lfl

./a.out

Enter the string: aaab

valid string Enter the string: Aaaaaba invalid

S🡪aSc/A

A🡪bSc/bc

aabbcccc

## OPEN ENDED PROGRAMS

Expt:17

**Implementation of Symbol table generator**

**Aim:**

To Implement a Symbol table generator

**ALGORITHM:**

Step1: Start the program for performing insert, display, delete, search and modify option in symbol table

Step2: Define the structure of the Symbol Table

Step3: Enter the choice for performing the operations in the symbol Table

Step4: If the entered choice is 1, search the symbol table for the symbol to be inserted. If the symbol is already present, it displays “Duplicate Symbol”. Else, insert the symbol and the corresponding address in the symbol table.

Step5: If the entered choice is 2, the symbols present in the symbol table are displayed. Step6: If the entered choice is 3, the symbol to be deleted is searched in the symbol table.

Step7: If it is not found in the symbol table it displays “Label Not found”. Else, the symbol is deleted.

Step8: If the entered choice is 5, the symbol to be modified is searched in the symbol table.

**Program**

//Implementation of symbol table #include<stdio.h> #include<ctype.h> #include<stdlib.h> #include<string.h> #include<math.h>

void main()

{

int i=0,j=0,x=0,n; void \*p,\*add[5];

char ch,srch,b[15],d[15],c; printf("Expression terminated by $:"); while((c=getchar())!='$')

{

b[i]=c; i++;

}

n=i-1;

printf("Given Expression:"); i=0;

while(i<=n)

{

printf("%c",b[i]); i++;

}

printf("\n Symbol Table\n"); printf("Symbol \t addr \t type"); while(j<=n)

{

c=b[j]; if(isalpha(toascii(c)))

{

p=malloc(c); add[x]=p;

d[x]=c;

printf("\n%c \t %d \t identifier\n",c,p); x++;

j++;

}

else

{

ch=c;

if(ch=='+'||ch=='-'||ch=='\*'||ch=='=')

{

p=malloc(ch); add[x]=p;

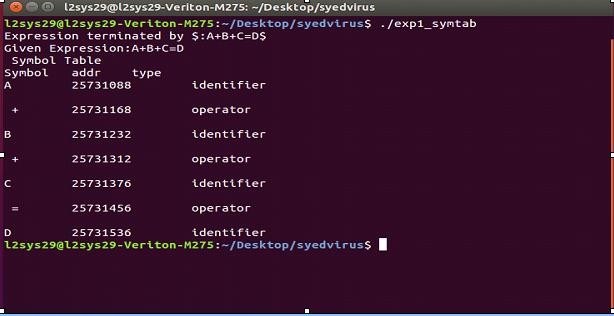
d[x]=ch;

printf("\n %c \t %d \t operator\n",ch,p); x++;

j++;

}}}}

## Output:



**Viva Questions**

1. Compiler translates the source code to
   1. Executable code
   2. Machine code
   3. Binary code
   4. Both B and C ANSWER: D
2. What is the output of lexical analyzer?
   1. A parse tree
   2. A list of tokens
   3. Intermediate code
   4. Machine code ANSWER: B
3. Grammar of the programming is checked at phase of compiler.
   1. Semantic analysis
   2. Syntax analysis
   3. Code optimization
   4. Code generation ANSWER: B
4. Compiler can check \_ error.
   1. Logical
   2. Syntax
   3. Content
   4. Both A and B ANSWER: B
5. A grammar that produces more than one parse tree for some sentence is called as
   1. Ambiguous
   2. Unambiguous
   3. Regular
   4. All of these ANSWER : A
6. Lexical analysis is about breaking a sequence of characters into
   1. Groups
   2. Packets
   3. Lines
   4. Tokens ANSWER : D
7. is the most general phase structured grammar.
   1. Context sensitive
   2. Regular
   3. Context free
   4. All of these

ANSWER: A

1. is considered as a sequence of characters in a token.
   1. Texeme
   2. Pattern
   3. Lexeme
   4. Mexeme ANSWER: C
2. What is the name of the process that determining whether a string of tokens can be generated by a grammar?
   1. Analysing
   2. Recognizing
   3. Translating
   4. Parsing ANSWER : D
3. A is a software utility that translates code written in higher language into a low level language.
   1. Converter
   2. Compiler
   3. Text editor
   4. Code optimizer ANSWER : B
4. Match all items in Group 1 with correct options from those given in Group 2. Group 1 Group 2
5. Regular expression 1.Syntax analysis
6. Pushdown automata 2.Code generation
7. Dataflow analysis 3.Lexical analysis
8. Register allocation 4. Code optimization

A. P-4. Q-1, R-2, S-3

B.P-3, Q-1, R-4, S-2

C.P-3, Q-4, R-1, S-2

D.P-2, Q-1, R-4, S-3 ANSWER : B

1. Which one of the following is a top-down parser?
   1. Recursive descent parser
   2. Operator precedence parser
   3. An LR(k) parser
   4. An LALR(k) parser ANSWER : A
2. Which of the following statements is false?
   1. An unambiguous grammar has same leftmost and rightmost derivation
   2. An LL(1) parser is a top-down parser C.LALR is more powerful than SLR

D.An ambiguous grammar can never be LR(k) for any k ANSWER : A

1. Some code optimizations are carried out on the intermediate code because A.they enhance the portability of the compiler to other target processors
2. program analysis is more accurate on intermediate code than on machine code
3. the information from dataflow analysis cannot otherwise be used for optimization
4. the information from the front end cannot otherwise be used for optimization ANSWER : A
5. One of the purposes of using intermediate code in compilers is to
   1. make parsing and semantic analysis simpler
   2. improve error recovery and error reporting
   3. increase the chances of reusing the machine-independent code optimizer in other compilers
   4. improve the register allocation ANSWER: C
6. In a compiler the data structure responsible for the management of information of the variables and their attributes is
   1. Symbol table
   2. Abstract syntax tree
   3. Parse tree
   4. Semantic stack ANSWER: A
7. Which of the following derivations does a top-down parser use while parsing an input string?

The input is assumed to be scanned in left to right order.

* 1. Leftmost derivation
  2. Leftmost derivation traced out in reverse C.Rightmost derivation

D.Rightmost derivation traced out in reverse ANSWER: A

1. Which one of the following statements is FALSE ?
   1. Context-free grammar can be used to specify both lexical and syntax rules. B.Type checking is done before parsing.

C.High-level language programs can be translated to different Intermediate Representations.

D.Arguments to a function can be passed using the program stack. ANSWER: B

1. A compiler program written in a high level language is called
   1. source program
   2. object program
   3. machine language program
   4. none of these ANSWER:A
2. The number of tokens in the following C statement is printf("i = %d, &i = %x", i, &i);

A.3 B.26 C.10

D.21 ANSWER : A

1. In a compiler, keywords of a language are recognized during A.parsing of the program
2. the code generation
3. the lexical analysis of the program
4. dataflow analysis ANSWER : C
5. Consider the following translation scheme. S → ER R → \*E{print("\*");}

R | ε E → F + E {print("+");} | F F → (S) | id {print(id.value);}

Here id is a token that represents an integer and id.value represents the corresponding integer value. For an input '2 \* 3 + 4', this translation scheme prints

A. 2 \* 3 + 4

B.2 \* +3 4

C.2 3 \* 4 +

D.2 3 4+\* ANSWER : D

1. The grammar A → AA | (A) | ε is not suitable for predictive-parsing because the grammar is
   1. ambiguous B.left-recursive C.right-recursive

D.an operator-grammar ANSWER : B

1. Which of the following grammar rules violate the requirements of an operator grammar ? P, Q, R arenonterminals, and r, s, t are terminals.
2. P → Q R
3. P → Q s R
4. P → ε
5. P → Q t R r
   * 1. only

B.1 and 3 only

C.2 and 3 only

D.3 and 4 only ANSWER : B

1. Consider the translation scheme shown below S → T R

R → + T {print ('+');} R | ε T → num {print (num.val);}

Here num is a token that represents an integer and num.val represents the corresponding integer value. For an input string '9 + 5 + 2', this translation scheme will print

A.9 + 5 + 2

B.9 5 + 2 +

C.9 5 2 + +

D.+ + 9 5 2 ANSWER : B

1. The lexical analysis for a modern computer language such as Java needs the power of which one of the following machine models in a necessary and sufficient sense?
   1. Finite state automata
   2. Deterministic pushdown automata
   3. Non-Deterministic pushdown automata
   4. Turing Machine ANSWER : A

REFERENCE :

1. *Principles of Compiler Design- Alfred Aho and Jeffrey Ullman*
2. *https://github.com/JacobSamro/Compiler-Design-Lab*
3. [*https://w*](http://www.geeksforgeeks.org/compiler-design-tutorials/)*ww.ge*[*eksforgeeks.org/compiler-design-tutorials/*](http://www.geeksforgeeks.org/compiler-design-tutorials/)