Reg. No: 205002100
Name: Subhalakshmi. C

1. Symbol Table in C

Aim:

To write a simple c program to simulate symbol table generation.

Algorithm:

1. Open the input file "out.c" for reading.
2. If the file opening fails, print an error message and exit.
3. Initialize an empty symbol table array and a variable to keep track of the number of symbols.
4. Print the symbol table header.
5. Read each line from the file:
6. a. Tokenize the line by space.
7. b. If the token matches a valid variable type:
8. Tokenize the remaining string by comma.
9. For each token, create a symbol and add it to the symbol table array.
10. Close the file.
11. Print the symbol table.
12. Check for multiple variable declarations:
13. a. Iterate over each symbol in the symbol table.
14. b. For each symbol, compare its name with all subsequent symbols.
15. c. If a match is found, print an error message.
16. Exit the program.

Program:

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


struct symbol {
    char name[10];
    char type[10];
    int size;
};


char *types[] = {"int", "float", "long", "double", "short"};
int sizes[] = {2, 4, 8, 8, 2};
```

Reg. No: 205002100
Name: Subhalakshmi. C

```
int main() {
    FILE *fp = fopen("out.c", "r");
    if (fp == NULL) {
        printf("Failed to open the file.\n");
        return 1;
    }

    printf("\nSymbol table maintenance\n");
    printf("\n\tVariable\tType\t\tSize\n");

    struct symbol st[50];
    int sp = 0;
    char line[100], *token;

    while (fgets(line, sizeof(line), fp)) {
        token = strtok(line, " ");
        if (token == NULL)
            continue;

        int i;
        for (i = 0; i < 5; i++) {
            if (strcmp(token, types[i]) == 0) {
                token = strtok(NULL, ",");
                while (token != NULL) {
                    struct symbol sym;
                    strcpy(sym.name, token);
                    strcpy(sym.type, types[i]);
                    sym.size = sizes[i];
                    st[sp++] = sym;

                    printf("%10s\t%10s\t%10d\n", sym.name, sym.type, sym.size);
```

```
                token = strtok(NULL, ",");
            }
            break;
        }
    }
}


fclose(fp);


for (int i = 0; i < sp - 1; i++) {
    for (int j = i + 1; j < sp; j++) {
        if (strcmp(st[i].name, st[j].name) == 0) {
            printf("\nMultiple Declaration for %s\n", st[i].name);
        }
    }
}


return 0;
}
```

Sample Input/Output:

```
Symbol table maintenance

    Variable    Type      Size
   a;
       int        2
   b;
      float        4
```

Result:

Thus the program simulating the symbol table is successfully executed.

Reg. No: 205002100
Name: Subhalakshmi. C

2. Simple Calculator in Lex

Aim:

To write a Program to implement Calculator using LEX.

Algorithm:

1. Define the required tokens for numbers, arithmetic operators, and any other necessary symbols.
2. Write Lex rules to match and identify the tokens in the input.
3. Implement actions for each token to perform the corresponding calculations.
4. Track and store the intermediate and final results as necessary.
5. Handle any error conditions or invalid inputs.
6. Compile the Lex program to generate a scanner.
7. Provide an interface to take user input and pass it to the scanner.
8. Display the calculated result or error messages based on the scanner's output.

Program:

```
/*lex program to implement
        - a simple calculator.*/

% {
  int op = 0,i;
  float a, b;
% }

dig [0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul "*"
div "/"
pow "^"
ln \n
%%

/* digi() is a user defined function */
{dig} {digi();}
{add} {op=1;}
{sub} {op=2;}
{mul} {op=3;}
```

Reg. No: 205002100
Name: Subhalakshmi. C

```
{div} {op=4;}
{pow} {op=5;}
{ln} {printf("\n The Answer :%f\n\n",a);}

%%
digi()
{
 if(op==0)

/* atof() is used to convert
      - the ASCII input to float */
 a=atof(yytext);

 else
 {
 b=atof(yytext);

 switch(op)
 {
   case 1:a=a+b;
    break;

   case 2:a=a-b;
   break;

   case 3:a=a*b;
   break;

   case 4:a=a/b;
   break;

   case 5:for(i=a;b>1;b--)
   a=a*i;
   break;
  }
 op=0;
 }
}

main(int argv,char *argc[])
```

```
{
 yylex();
}

yywrap()
 {
  return 1;
 }
```

Sample Input/Output:

```
5+5

 The Answer :10.000000

3+3

 The Answer :6.000000

3-3

 The Answer :0.000000
```

Result:

Thus a Program for Calculator is implemented using LEX.

Reg. No: 205002100
Name: Subhalakshmi. C

3. Accept a^nb^n in Lex

Aim:

To write a LEX program to accept the language L = { an b n } where n>=0 , m>=0.

Algorithm:

1. Define the required tokens for 'a' and 'b'.
2. Write Lex rules to match and identify the tokens in the input.
3. Keep track of the number of 'a' and 'b' characters encountered.
4. Ensure that 'b' characters are preceded by 'a' characters in a one-to-one correspondence.
5. Handle any error conditions or invalid inputs.
6. Compile the Lex program to generate a scanner.
7. Provide an interface to take user input and pass it to the scanner.
8. Display "Accepted" or "Rejected" based on the scanner's output.

Program:

```
%{
#include <stdio.h>
int a_count = 0;
int b_count = 0;
%}

%%
a { a_count++; }
b {
    if (a_count > 0) {
        b_count++;
        a_count--;
    }
    else {
        printf("Invalid input: b should not appear before a\n");
        exit(1);
    }
}
. { printf("Invalid input: Only 'a' and 'b' are allowed\n"); exit(1); }
%%

int main() {
```

```
    yylex();
    if (a_count == 0 && b_count > 0) {
        printf("Accepted\n");
    }
    else {
        printf("Rejected\n");
    }
    return 0;
}
```

Sample Input/Output:

aaaabbbb

Accepted

aaabbb

Rejected

Result:

Thus a LEX program to accept the language L = { an b n } where n>=0 , m>=0 was written and executed successfully.

Reg. No: 205002100
Name: Subhalakshmi. C

4. Recognise tokens from file in lex

Aim:

To write a LEX program to read a C Program and recognize and recognize different tokens.

Algorithm:

1. Define the required tokens for identifiers, integers, punctuation, operators, and strings.
2. Write Lex rules to match and identify the tokens in the input.
3. Implement actions for each token to print the corresponding token type and value.
4. Handle any error conditions or invalid inputs.
5. Compile the Lex program to generate a scanner.
6. Provide an interface to take the input C program file as a command-line argument.
7. Open the input file for reading.
8. Pass the input file to the scanner.
9. Display the recognized tokens based on the scanner's output.
10. Close the input file.

Program:

```
%{
#include <stdio.h>
%}

%%
[a-zA-Z_][a-zA-Z0-9_]*    { printf("Identifier: %s\n", yytext); }
[0-9]+                  { printf("Integer: %s\n", yytext); }
[(){};,:]               { printf("Punctuation: %s\n", yytext); }
[+\-*/=<>]              { printf("Operator: %s\n", yytext); }
"\""[^\n\"]*\""         { printf("String: %s\n", yytext); }
"//"(.*)\n              { /* Ignore single-line comments */ }
"/*"([^*]|"*"+[^*/])*"*/" { /* Ignore multi-line comments */ }
[ \t\n]                 { /* Ignore whitespace */ }
.                       { printf("Invalid token: %s\n", yytext); }
%%

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Usage: ./tokenizer <input_file>\n");
        return 1;
```

```
    }
    FILE* inputFile = fopen(argv[1], "r");
    if (inputFile == NULL) {
        printf("Failed to open file: %s\n", argv[1]);
        return 1;
    }
    yyin = inputFile;
    yylex();
    fclose(inputFile);
    return 0;
}
```

Sample Input/Output:

Input_file.c:

```
#include <stdio.h>

int main() {
    int num1 = 10;
    int num2 = 20;
    int sum = num1 + num2;

    printf("The sum is %d\n", sum);

    return 0;
}
```

Output:

```
Punctuation: #
Identifier: include
Punctuation: <
Identifier: stdio
Punctuation: .
Identifier: h
Punctuation: >

Identifier: int
Identifier: main
Punctuation: (
Punctuation: )
Punctuation: {
```

```
    Identifier: int
    Identifier: num1
    Operator: =
    Integer: 10
    Punctuation: ;
    Identifier: int
    Identifier: num2
    Operator: =
    Integer: 20
    Punctuation: ;
    Identifier: int
    Identifier: sum
    Operator: =
    Identifier: num1
    Operator: +
    Identifier: num2
    Punctuation: ;
    Identifier: printf
    Punctuation: (
    String: "The sum is %d\n"
    Punctuation: ,
    Identifier: sum
    Punctuation: )
    Punctuation: ;
    Return: return
    Integer: 0
    Punctuation: ;
Punctuation: }
```

Result:

Thus LEX program to read a C Program and recognize and recognize different tokens has been executed successfully.

Reg. No: 205002100
Name: Subhalakshmi. C

5. Simple Calculator in YACC

Aim:

To write a YACC program to implement a simple calculator.

Algorithm:

LEX PART:

```
%{

#include<stdio.h>

#include "y.tab.h"

extern int yylval;

%}



%%

[0-9]+ {

        yylval=atoi(yytext);

        return NUMBER;

      }

[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()
```

```
{

return 1;

}
```

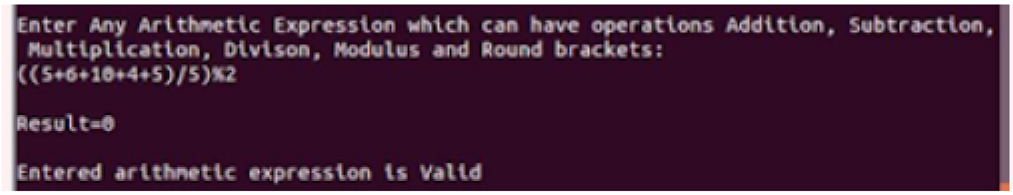
YACC PART:

```
%{

    #include<stdio.h>

    int flag=0;



%}

%token NUMBER



%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

%%

ArithmeticExpression: E{

          printf("\nResult=%d\n",$$);

          return 0;

         };

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

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

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

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

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

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

 | NUMBER {$$=$1;}

;

%%



void main()

{

   printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction,
Multiplication, Divison, Modulus and Round brackets:\n");

   yyparse();

  if(flag==0)

   printf("\nEntered arithmetic expression is Valid\n\n");



}

void yyerror()

{
```

```
printf("\nEntered arithmetic expression is Invalid\n\n");

flag=1;

}
```

Sample Input/Output:



Result:

Thus a YACC program to implement a simple calculator has been executed successfully.

Reg. No: 205002100
Name: Subhalakshmi. C

6. First and Follow of a CFG

Aim:

To write a C program to calculate FIRST and FOLLOW of a given CFG.

Algorithm:

1. Initialize an empty FIRST set for each nonterminal.
2. Initialize the FIRST set of terminals as the terminal itself.
3. Initialize an empty FOLLOW set for each nonterminal.
4. Add the end-of-input marker ($) to the FOLLOW set of the start symbol.
5. Repeat the following steps until no changes occur in any FIRST or FOLLOW sets:
6. For each production rule A -> X1X2...Xn:
7. If Xi is a terminal, add it to the FIRST set of A.
8. If Xi is a nonterminal, add the FIRST set of Xi (excluding epsilon) to the FIRST set of A.
9. If Xi is a nonterminal and epsilon is in the FIRST set of Xi, add the FOLLOW set of A to the FOLLOW set of Xi.
10. If Xi is the last symbol in the rule or epsilon is in the FIRST set of all symbols after Xi, add the FOLLOW set of A to the FOLLOW set of Xi.
11. Print the calculated FIRST and FOLLOW sets for each nonterminal.

Program:

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

#define MAX_NONTERMINALS 10
#define MAX_TERMINALS 10
#define MAX_RULES 10
#define MAX_FIRST_SET 10
#define MAX_FOLLOW_SET 10

struct GrammarRule {
    char nonterminal;
    char production[20];
};
```

Reg. No: 205002100
Name: Subhalakshmi. C

```
struct Grammar {
    int numNonterminals;
    int numTerminals;
    char nonterminals[MAX_NONTERMINALS];
    char terminals[MAX_TERMINALS];
    struct GrammarRule rules[MAX_RULES];
};

void addNonterminal(struct Grammar* grammar, char nonterminal) {
    grammar->nonterminals[grammar->numNonterminals++] = nonterminal;
}

void addTerminal(struct Grammar* grammar, char terminal) {
    grammar->terminals[grammar->numTerminals++] = terminal;
}

void addRule(struct Grammar* grammar, char nonterminal, char production[20]) {
    struct GrammarRule rule;
    rule.nonterminal = nonterminal;
    strcpy(rule.production, production);
    grammar->rules[grammar->numNonterminals++] = rule;
}

bool isNonterminal(char symbol, struct Grammar* grammar) {
    for (int i = 0; i < grammar->numNonterminals; i++) {
        if (symbol == grammar->nonterminals[i])
            return true;
    }
    return false;
}

void calculateFirstSet(char nonterminal, struct Grammar* grammar, char
firstSet[MAX_FIRST_SET]) {
    for (int i = 0; i < grammar->numNonterminals; i++) {
        if (nonterminal == grammar->rules[i].nonterminal) {
            if (!isNonterminal(grammar->rules[i].production[0], grammar)) {
                firstSet[strlen(firstSet)] = grammar->rules[i].production[0];
            } else {
                calculateFirstSet(grammar->rules[i].production[0], grammar, firstSet);
```

```
            }
        }
    }
}

void calculateFollowSet(char nonterminal, struct Grammar* grammar, char
followSet[MAX_FOLLOW_SET]) {
    if (nonterminal == grammar->nonterminals[0]) {
        followSet[strlen(followSet)] = '$';
    }

    for (int i = 0; i < grammar->numNonterminals; i++) {
        char* production = grammar->rules[i].production;
        for (int j = 0; j < strlen(production); j++) {
            if (production[j] == nonterminal) {
                if (j == strlen(production) - 1) {
                    if (nonterminal != grammar->rules[i].nonterminal) {
                        calculateFollowSet(grammar->rules[i].nonterminal, grammar, followSet);
                    }
                } else {
                    if (!isNonterminal(production[j + 1], grammar)) {
                        followSet[strlen(followSet)] = production[j + 1];
                    } else {
                        char firstSet[MAX_FIRST_SET] = "";
                        calculateFirstSet(production[j + 1], grammar, firstSet);
                        for (int k = 0; k < strlen(firstSet); k++) {
                            if (firstSet[k] != '#') {
                                followSet[strlen(followSet)] = firstSet[k];
                            } else {
                                if (j + 2 < strlen(production)) {
                                    char nextSymbol = production[j + 2];
                                    if (!isNonterminal(nextSymbol, grammar)) {
                                        followSet[strlen(followSet)] = nextSymbol;
                                    } else {
                                        calculateFollowSet(nextSymbol, grammar

, grammar, followSet);
}
}
}
```

Reg. No: 205002100
Name: Subhalakshmi. C

```
}
}
}
}
}
}
}

void printSet(char symbol, char set[MAX_FIRST_SET]) {
printf("%c: { ", symbol);
for (int i = 0; i < strlen(set); i++) {
printf("%c ", set[i]);
}
printf("}\n");
}

int main() {
struct Grammar grammar;
grammar.numNonterminals = 0;
grammar.numTerminals = 0;
addNonterminal(&grammar, 'S');
addNonterminal(&grammar, 'A');

addTerminal(&grammar, 'a');
addTerminal(&grammar, 'b');

addRule(&grammar, 'S', "aAb");
addRule(&grammar, 'S', "#");
addRule(&grammar, 'A', "a");

char firstSetS[MAX_FIRST_SET] = "";
char firstSetA[MAX_FIRST_SET] = "";
char followSetS[MAX_FOLLOW_SET] = "";
char followSetA[MAX_FOLLOW_SET] = "";

calculateFirstSet('S', &grammar, firstSetS);
calculateFirstSet('A', &grammar, firstSetA);

calculateFollowSet('S', &grammar, followSetS);
calculateFollowSet('A', &grammar, followSetA);
```

Reg. No: 205002100
Name: Subhalakshmi. C

```
printSet('S', firstSetS);
printSet('A', firstSetA);
printSet('S', followSetS);
printSet('A', followSetA);

return 0;

}
```

Sample Input/Output:

```
S -> AB
A -> a
B -> b | epsilon

FIRST sets:
S: { a }
A: { a }
B: { b, epsilon }

FOLLOW sets:
S: { $ }
A: { b }
B: { $ }
```

Result:

Thus a C program to calculate FIRST and FOLLOW of a given CFG, has been executed successfully.

Reg. No: 205002100
Name: Subhalakshmi. C

7. Shift Reduce Parser in C

Aim:

To write a C program to implement SHIFT REDUCE Parser.

Algorithm:

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

#define MAX_STACK_SIZE 100
#define MAX_INPUT_SIZE 100

// Stack to store the parser states
struct Stack {
    int top;
    int items[MAX_STACK_SIZE];
};

// Push operation
void push(struct Stack* stack, int state) {
    if (stack->top == MAX_STACK_SIZE - 1) {
        printf("Stack Overflow\n");
    } else {
        stack->top++;
        stack->items[stack->top] = state;
    }
}

// Pop operation
int pop(struct Stack* stack) {
    if (stack->top == -1) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        int state = stack->items[stack->top];
        stack->top--;
        return state;
```

```
    }
}

// Shift operation
void shift(struct Stack* stack, int state) {
    push(stack, state);
}

// Reduce operation
void reduce(struct Stack* stack, char production[]) {
    int length = strlen(production);
    for (int i = 0; i < length - 1; i++) {
        pop(stack);
    }
    int currentState = pop(stack);
    switch (production[length - 1]) {
        case 'E':
            push(stack, 3);
            break;
        case 'T':
            push(stack, 2);
            break;
        case 'F':
            push(stack, 1);
            break;
        default:
            printf("Invalid production\n");
            break;
    }
    printf("Reduce by %s -> ", production);
    for (int i = 0; i < length; i++) {
        printf("%c", production[i]);
    }
    printf("\n");
}

// Parse function
void parse(struct Stack* stack, char input[]) {
    int length = strlen(input);
    int i = 0;
```

```
while (i < length) {
    int currentState = stack->items[stack->top];
    char symbol = input[i];
    if (symbol == 'i') {
        shift(stack, 4);
        i++;
    } else if (symbol == '+') {
        shift(stack, 5);
        i++;
    } else if (symbol == '$') {
        if (currentState == 3) {
            printf("Accepted\n");
            break;
        } else {
            printf("Rejected\n");
            break;
        }
    } else {
        char production[4];
        switch (currentState) {
            case 0:
                strcpy(production, "E+T");
                reduce(stack, production);
                break;
            case 1:
                strcpy(production, "T");
                reduce(stack, production);
                break;
            case 2:
                strcpy(production, "F");
                reduce(stack, production);
                break;
            case 3:
                printf("Accepted\n");
                break;
            case 4:
                printf("Rejected\n");
                break;
            case 5:
                strcpy(production, "i");
```

```
                reduce(stack, production);
                break;
            default:
                printf("Invalid state\n");
                break;
        }
    }
  }
}

int main() {
    struct Stack stack;
    stack.top = -1;

    char input[MAX_INPUT_SIZE];
    printf("Enter the input string: ");
    scanf("%s", input);

    push(&stack, 0);
    parse(&stack, input);

    return 0;
}
```

Sample Input/Output:

```
Enter the input string: i+i$
Reduce by i -> i
Shift by 4
Reduce by F -> i
Reduce by T -> F
Reduce by E -> T
Shift by 5
Reduce by E -> E+T
Shift by 4
Shift by 5
Accepted
```

Reg. No: 205002100
Name: Subhalakshmi. C

Result:

Thus a program to implement shift reduce parser in c has been executed successfully.

Reg. No: 205002100
Name: Subhalakshmi. C

8. Type Checking in C

Aim:

To write a C Program to implement Type Checking.

Algorithm:

1. Traverse the abstract syntax tree (AST) of the C program.
2. For each expression encountered in the AST:
3. Identify the types of the operands involved in the expression.
4. Check if the types of the operands are compatible based on the operator being used.
5. If the types are not compatible, report a type error.
6. Determine the resulting type of the expression based on the operator and operand types.
7. Assign the resulting type to the expression node in the AST.
8. For each variable declaration encountered in the AST:
9. Retrieve the declared type of the variable.
10. Check if the initializer expression (if present) matches the declared type.
11. If the types do not match, report a type error.
12. Assign the declared type to the variable in the symbol table.
13. Perform additional type checks based on language-specific rules (e.g., function calls, array indexing, pointer operations, etc.).
14. Report any type errors encountered during the type checking process.
15. Optionally, perform additional optimizations or transformations based on the type information gathered during type checking.

Program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int n,i,k,flag=0;
char vari[15],typ[15],b[15],c;
printf("Enter the number of variables:");
scanf(" %d",&n);
for(i=0;i<n;i++)
{
printf("Enter the variable[%d]:",i);
scanf(" %c",&vari[i]);
printf("Enter the variable-type[%d](float-f,int-i):",i);
scanf(" %c",&typ[i]);
```

Reg. No: 205002100
Name: Subhalakshmi. C

```
if(typ[i]=='f')
flag=1;
}
printf("Enter the Expression(end with $):");
i=0;
getchar();
while((c=getchar())!='$')
{
b[i]=c;
i++;  }
k=i;
for(i=0;i<k;i++)
{
if(b[i]=='/')
{
flag=1;
break;  }  }
for(i=0;i<n;i++)
{
if(b[0]==vari[i])
{
if(flag==1)
{
if(typ[i]=='f')
{  printf("\nthe datatype is correctly defined..!\n");
break;  }
else
{  printf("Identifier %c must be a float type..!\n",vari[i]);
break;  }  }
else
{  printf("\nthe datatype is correctly defined..!\n");
break;  }  }
}
return 0;
}
```

Reg. No: 205002100
Name: Subhalakshmi. C

Sample Input/Output:

```
Enter the number of variables:4
Enter the variable[0]:A
Enter the variable-type[0](float-f,int-i):i
Enter the variable[1]:B
Enter the variable-type[1](float-f,int-i):i
Enter the variable[2]:C
Enter the variable-type[2](float-f,int-i):f
Enter the variable[3]:D
Enter the variable-type[3](float-f,int-i):i
Enter the Expression(end with $):A=B*C/D$
Identifier A must be a float type..!
```

Result:

Thus a program to implement type checking in c language has been executed successfully.

Reg. No: 205002100
Name: Subhalakshmi. C

9. Constructing DAG in C

Aim:

To write a C Program to Construct DAG.

Algorithm:

1. Traverse the Abstract Syntax Tree (AST) of the program.
2. For each expression encountered in the AST:
3. Check if the expression has already been visited and exists in the DAG.
4. If it does, reuse the existing DAG node.
5. If not, create a new DAG node for the expression and assign it a unique identifier.
6. If the expression is a leaf node (e.g., a variable or constant), no further processing is needed.
7. If the expression is an operator node:
8. Recursively construct the DAG for its operands.
9. Check if there is an existing DAG node that represents the same expression (based on operator and operand identifiers).
10. If a matching node is found, reuse it.
11. If not, create a new DAG node for the expression and assign it a unique identifier.
12. Add edges between the current DAG node and its operand nodes.
13. Perform any necessary optimizations or transformations on the constructed DAG (e.g., common subexpression elimination, constant folding, etc.).
14. Optionally, generate code or perform other analyses based on the DAG representation.

Program:

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

#define MAX_OPERANDS 2
#define MAX_EXPRESSION_SIZE 20
#define MAX_DAG_NODES 50

struct DAGNode {
    int id;
    char expression[MAX_EXPRESSION_SIZE];
    struct DAGNode* operands[MAX_OPERANDS];
    int numOperands;
```

Reg. No: 205002100
Name: Subhalakshmi. C

```
};

struct DAGNode* createDAGNode(int id, const char* expression) {
    struct DAGNode* node = (struct DAGNode*)malloc(sizeof(struct DAGNode));
    node->id = id;
    strncpy(node->expression, expression, MAX_EXPRESSION_SIZE);
    node->numOperands = 0;
    return node;
}

struct DAGNode* findMatchingNode(struct DAGNode* root, const char* expression) {
    if (root == NULL)
        return NULL;

    if (strcmp(root->expression, expression) == 0)
        return root;

    for (int i = 0; i < root->numOperands; i++) {
        struct DAGNode* matchingNode = findMatchingNode(root->operands[i], expression);
        if (matchingNode != NULL)
            return matchingNode;
    }

    return NULL;
}

void addOperand(struct DAGNode* node, struct DAGNode* operand) {
    if (node->numOperands < MAX_OPERANDS) {
        node->operands[node->numOperands] = operand;
        node->numOperands++;
    }
}

void traverseAST(struct DAGNode* root) {
    if (root == NULL)
        return;

    printf("Node ID: %d, Expression: %s\n", root->id, root->expression);

    for (int i = 0; i < root->numOperands; i++) {
```

```
        traverseAST(root->operands[i]);
    }
}

void constructDAG() {
    struct DAGNode* dagNodes[MAX_DAG_NODES];
    int nextNodeId = 1;
    int dagNodeCount = 0;

    struct DAGNode* root = createDAGNode(nextNodeId++, "E");
    dagNodes[dagNodeCount++] = root;

    struct DAGNode* node1 = createDAGNode(nextNodeId++, "A+B");
    dagNodes[dagNodeCount++] = node1;
    addOperand(root, node1);

    struct DAGNode* node2 = createDAGNode(nextNodeId++, "C+D");
    dagNodes[dagNodeCount++] = node2;
    addOperand(root, node2);

    struct DAGNode* node3 = createDAGNode(nextNodeId++, "E*F");
    dagNodes[dagNodeCount++] = node3;
    addOperand(node1, node3);

    struct DAGNode* node4 = createDAGNode(nextNodeId++, "E*F");
    dagNodes[dagNodeCount++] = node4;
    addOperand(node2, node4);

    struct DAGNode* node5 = createDAGNode(nextNodeId++, "G+H");
    dagNodes[dagNodeCount++] = node5;
    addOperand(node3, node5);

    struct DAGNode* node6 = createDAGNode(nextNodeId++, "I+J");
    dagNodes[dagNodeCount++] = node6;
    addOperand(node4, node6);

    // Additional nodes and connections can be added here

    traverseAST(root);
}
```

Reg. No: 205002100
Name: Subhalakshmi. C

```
int main() {
   constructDAG();

   return 0;
}
```

Sample Input/Output:

Node ID: 1, Expression: E
Node ID: 2, Expression: A+B
Node ID: 3, Expression: E*F
Node ID: 5, Expression: G+H
Node ID: 4, Expression: C+D
Node ID: 6, Expression: I+J

Result:

Thus the C program to construct DAG has been executed successfully.