PRACTICAL – 6

Implementation of a parser for an expression grammar using LEX and YACC.

Lex file (lexer.l):

```
%{
#include "y.tab.h"
%}

%%

[0-9]+      { yylval.num = atoi(yytext); return NUM; }
[-+*/()\n]  { return *yytext; }
[ \t]       ; /* skip whitespace */

.           { fprintf(stderr, "Invalid character: %s\n", yytext); }

%%

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

Yacc file (parser.y):

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

%token NUM

%%
expression : expression '+' expression
           | expression '-' expression
           | expression '*' expression
           | expression '/' expression
```

```
        | '(' expression ')'
        | NUM
        ;

%%

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

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

output

mathematica Copy code

```mathematica
Enter an arithmetic expression: 2 + * 3
```

Output:

vbnet Copy code

```vbnet
Error: syntax error
```

PRACTICAL 7

Generate three address codes for a simple program using LEX and YACC.

C Program (arithmetic.c):

```
#include <stdio.h>

int main() {
    int temp_count = 0;
    char input[100];
    printf("Enter an arithmetic expression: ");
    scanf("%[^\n]", input);
    printf("Input expression: %s\n", input);

    return 0;
}
```

Lex file (lexer.l):

```
%{
#include "y.tab.h"
%}

%%

[0-9]+      { yylval.num = atoi(yytext); return NUM; }
[-+*/()\n]  { return *yytext; }
[ \t]       ; /* skip whitespace */
.           { fprintf(stderr, "Invalid character: %s\n", yytext); }
%%
int yywrap() {
    return 1;
}
```

Yacc file (parser.y):

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

%token NUM

%%
expression : expression '+' expression
        {
           printf("t%d = t%d + t%d;\n", ++temp_count, $1, $3);
           $$ = temp_count;
        }
        | expression '-' expression
        {
           printf("t%d = t%d - t%d;\n", ++temp_count, $1, $3);
           $$ = temp_count;
        }
        | expression '*' expression
        {
           printf("t%d = t%d * t%d;\n", ++temp_count, $1, $3);
           $$ = temp_count;
        }
        | expression '/' expression
        {
           printf("t%d = t%d / t%d;\n", ++temp_count, $1, $3);
           $$ = temp_count;
        }
        | '(' expression ')'
        {
           $$ = $2;
        }
        | NUM
        {
           printf("t%d = %d;\n", ++temp_count, $1);
           $$ = temp_count;
        }
        ;
```

```
%%

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

int temp_count = 0;

int main() {
    printf("Enter an arithmetic expression: ");
    yyparse();
    return 0;
}
```

OUTPUT

mathematica Copy code

```
Enter an arithmetic expression: 2 + * 3
```

Output:

vbnet Copy code

```
Error: syntax error
```

PRACTICAL 8

Generate and populate appropriate Symbol Table.

C Program (symbol_table.c):

```
#include <stdio.h>
#include <string.h>
typedef struct {
    char name[50];
    int value;
} Symbol;

// Symbol table
Symbol symbolTable[100];
int symbolCount = 0;
void addToSymbolTable(char name[], int value) {
    Symbol symbol;
    strcpy(symbol.name, name);
    symbol.value = value;
    symbolTable[symbolCount++] = symbol;
}
int findSymbol(char name[]) {
    for (int i = 0; i < symbolCount; ++i) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            return symbolTable[i].value;
        }
    }
    return -1; // Symbol not found
}

int main() {
    char input[100];
    char name[50];
    int value;
```

```
    printf("Enter variable declarations and assignments (e.g., variable = value):\n");
    while (fgets(input, sizeof(input), stdin) != NULL) {
        sscanf(input, "%s = %d", name, &value);
        addToSymbolTable(name, value);
    }
    printf("\nSymbol Table:\n");
    for (int i = 0; i < symbolCount; ++i) {
        printf("%s = %d\n", symbolTable[i].name, symbolTable[i].value);
    }
    char searchName[50];
    printf("\nEnter a variable name to search in the symbol table: ");
    scanf("%s", searchName);

    int searchResult = findSymbol(searchName);
    if (searchResult != -1) {
        printf("Value of %s: %d\n", searchName, searchResult);
    } else {
        printf("Symbol %s not found in the symbol table.\n", searchName);
    }

    return 0;
}
```

OUTPUT

makefile Copy code

```
variable1 = 10
variable2 = 20
variable3 = 30
```

Output:

mathematica Copy code

```
Symbol Table:
variable1 = 10
variable2 = 20
variable3 = 30

Enter a variable name to search in the symbol table: variable2
Value of variable2: 20
```

PRACTICAL 9

Implementation of simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)

CODE-

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>


typedef struct {

    char* operation;

    int operand1;

    int operand2;

} ExpressionNode;

void constantFolding(ExpressionNode* expression) {

    if (isdigit(expression->operation[0]) && isdigit(expression->operation[2])) {

        int result;

        switch (expression->operation[1]) {

            case '+':

                result = expression->operation[0] - '0' + expression->operation[2] - '0';

                break;

            case '-':

                result = expression->operation[0] - '0' - expression->operation[2] - '0';

                break;

            case '*':

                result = (expression->operation[0] - '0') * (expression->operation[2] - '0');

                break;

            case '/':

                if (expression->operation[2] != '0') {

                    result = (expression->operation[0] - '0') / (expression->operation[2] - '0');

                } else {

                    printf("Error: Division by zero.\n");

                    exit(1);

                }

                break;

        }

        free(expression->operation);
```

```
        expression->operation = malloc(4);
        snprintf(expression->operation, 4, "%d", result);
    } }
void strengthReduction(ExpressionNode* expression) {
    if (expression->operation[1] == '*' && expression->operation[2] == '2') {
        expression->operation[2] = '+';
    )}
void algebraicTransformation(ExpressionNode* expression) {
    if (expression->operation[0] == '(' && expression->operation[3] == ')' &&
        expression->operation[1] == '+' && expression->operation[2] == '0') {
        free(expression->operation);
        expression->operation = malloc(2);
        snprintf(expression->operation, 2, "%c", expression->operation[0]);
    }  }
void printExpression(ExpressionNode* expression) {
    printf("Optimized Expression: %s\n", expression->operation);
}
int main() {
    ExpressionNode expression;
    expression.operation = strdup("(3+0)");

    printf("Original Expression: %s\n", expression.operation);
    constantFolding(&expression);
    strengthReduction(&expression);
    algebraicTransformation(&expression);
    printExpression(&expression);

    free(expression.operation);
  return 0; }
```

OUTPUT

Output:

mathematica   Copy code

```
Original Expression: (3+0)
Optimized Expression: 3
```

PRACTICAL 10

Generate an appropriate Target Code from the given intermediate code assuming suitable processor details.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
    char op;
    int operand;
} Instruction;
void generateTargetCode(Instruction* intermediateCode, int size) {
    printf("Target Code:\n");
    for (int i = 0; i < size; ++i) {
        switch (intermediateCode[i].op) {
            case '+':
                printf("PUSH %d\n", intermediateCode[i].operand);
                printf("PUSH %d\n", intermediateCode[i + 1].operand);
                printf("ADD\n");
                i++;  // Skip the next instruction since it's already used
                break;
            case '-':
                printf("PUSH %d\n", intermediateCode[i].operand);
                printf("PUSH %d\n", intermediateCode[i + 1].operand);
                printf("SUB\n");
                i++;  // Skip the next instruction since it's already used
                break;
            case '*':
                printf("PUSH %d\n", intermediateCode[i].operand);
                printf("PUSH %d\n", intermediateCode[i + 1].operand);
                printf("MUL\n");
                i++;  // Skip the next instruction since it's already used
                break;
            case '/':
                printf("PUSH %d\n", intermediateCode[i].operand);
                printf("PUSH %d\n", intermediateCode[i + 1].operand);
                printf("DIV\n");
                i++;
```

```
            break;
        default:
            printf("Unknown operation: %c\n", intermediateCode[i].op);
            exit(1);
        }
    }
}

int main() {
    Instruction intermediateCode[] = {
        {'*', 3},  // PUSH 3
        {'4', 0},  // PUSH 4
        {'*', 0},  // MUL
        {'2', 0},  // PUSH 2
        {'+', 0},  // ADD
        {'1', 0},  // PUSH 1
        {'-', 0},  // SUB
    };
    int size = sizeof(intermediateCode) / sizeof(intermediateCode[0]);
    generateTargetCode(intermediateCode, size);
    return 0;
}
```

OUTPUT

arithmetic expression: 2 + 3 * 4 – 1

```
Target Code:
PUSH 3
PUSH 4
MUL
PUSH 2
ADD
PUSH 1
SUB
```