CITAR CHENNAI INSTITUTE OF TECHNOLOGY AND APPLIED RESEARCH

CS3501 - Compiler Design Laboratory

Questions

1. Develop a program in C to design a lexical analyzer that recognizes identifiers and constants

2. Implement a symbol table that involves insertion, deletion, search and modify operations using C language

3. Design a program that implements a lexical analyzer that separates token using a LEX tool

4. Use YACC tool to recognize a valid arithmetic expression that uses basic arithmetic operators [+,-,*,/]

5. Design a program to recognize a valid variable which starts with an alphabet followed by any number of digits or alphabets using YACC tool

6. Use LEX and YACC tools to implement a native calculator

7. Design a program to generate a three-address code from a given arithmetic expression

8. Implement a simple type checker that checks the scope of the variables and semantic errors from the given statement

9. Develop a program that optimizes the given input block using Code Optimization Techniques

10. Given an intermediate code as an input. Develop a program that generates the machine code from the given input

11. Generate a valid pattern that recognizes all statements that begins with an Upper-Case Letter followed by five digits or alphabets. Use a YACC tool to do the same

12. Design a lexical analyzer that identifies comments, operators and keywords from a given expression

13. Develop a Program to recognize a valid control structures syntax of C language (For loop, while loop, if else, if-else-if, switch-case, etc.)

14. Develop a Lex program to find out the total number of vowels and consonants from the given input string

15. Develop a program to generate machine code from a given postfix notation

Answers

1. Lexical Analyzer for Identifiers and Constants

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

int isIdentifier(char *str) {
    if (!isalpha(str[0]) && str[0] != '_') return 0;
    for (int i = 1; str[i]; i++)
        if (!isalnum(str[i]) && str[i] != '_') return 0;
    return 1;
}

int isConstant(char *str) {
    for (int i = 0; str[i]; i++)
        if (!isdigit(str[i])) return 0;
    return 1;
}

int main() {
    char tokens[][20] = {"abc", "123", "var1", "45", "test_123", "78"};
    int n = 6;

    printf("Lexical Analysis:\n");
    for (int i = 0; i < n; i++) {
        if (isIdentifier(tokens[i]))
            printf("%s -> Identifier\n", tokens[i]);
        else if (isConstant(tokens[i]))
            printf("%s -> Constant\n", tokens[i]);
    }
    return 0;
}
```

Output:

```
Lexical Analysis:
abc -> Identifier
123 -> Constant
var1 -> Identifier
45 -> Constant
```

test_123 -> Identifier
78 -> Constant

2. Symbol Table Operations

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

struct Symbol {
    char name[20];
    char type[10];
    int value;
};

struct Symbol symTable[10];
int count = 0;

void insert(char *name, char *type, int value) {
    strcpy(symTable[count].name, name);
    strcpy(symTable[count].type, type);
    symTable[count].value = value;
    count++;
}

int search(char *name) {
    for (int i = 0; i < count; i++)
        if (strcmp(symTable[i].name, name) == 0) return i;
    return -1;
}

void delete(char *name) {
    int pos = search(name);
    if (pos != -1) {
        for (int i = pos; i < count-1; i++)
            symTable[i] = symTable[i+1];
        count--;
    }
}

void modify(char *name, int newValue) {
    int pos = search(name);
```

```
    if (pos != -1) symTable[pos].value = newValue;
}

void display() {
    printf("\nSymbol Table:\n");
    for (int i = 0; i < count; i++)
        printf("%s\t%s\t%d\n", symTable[i].name, symTable[i].type, symTable[i].value);
}

int main() {
    insert("x", "int", 10);
    insert("y", "float", 20);
    display();

    printf("\nSearch 'x': %d\n", search("x"));
    modify("x", 100);
    display();
    delete("y");
    display();
    return 0;
}
```

Output:

```
Symbol Table:
x       int     10
y       float   20

Search 'x': 0
Symbol Table:
x       int     100
y       float   20

Symbol Table:
x       int     100
```

3. Lexical Analyzer using LEX

lexer.l:

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

%%
[0-9]+          { printf("Number: %s\n", yytext); }
[a-zA-Z][a-zA-Z0-9]* { printf("Identifier: %s\n", yytext); }
[ \t\n]         ;
.               { printf("Operator: %s\n", yytext); }
%%

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

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

Compile & Run:

```
lex lexer.l
gcc lex.yy.c -o lexer -ll
echo "a = b + 123" | ./lexer
```

Output:

```
Identifier: a
Operator: =
Identifier: b
Operator: +
Number: 123
```

4. Arithmetic Expression Parser using YACC

parser.y:

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

%token NUMBER
%left '+' '-'
%left '*' '/'

%%
expr: expr '+' expr { printf("Valid Expression: Addition\n"); }
    | expr '-' expr { printf("Valid Expression: Subtraction\n"); }
    | expr '*' expr { printf("Valid Expression: Multiplication\n"); }
    | expr '/' expr { printf("Valid Expression: Division\n"); }
    | NUMBER        { printf("Valid Number\n"); }
    | '(' expr ')'  { printf("Valid Parentheses\n"); }
    ;
%%

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

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

lex.l:

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

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

```
%%

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

Compile & Run:

```
yacc -d parser.y
lex lex.l
gcc y.tab.c lex.yy.c -o parser
echo "2 + 3 * 4" | ./parser
```

Output:

```
Enter arithmetic expression: Valid Number
Valid Number
Valid Expression: Multiplication
Valid Expression: Addition
```

5. Variable Recognition using YACC

var_parser.y:

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

%token VALID INVALID

%%
input: VALID   { printf("Valid Variable\n"); }
     | INVALID { printf("Invalid Variable\n"); }
     ;
%%

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

```
int yylex() {
    char input[100];
    if (scanf("%s", input) != 1) return 0;

    if (isalpha(input[0])) {
        int valid = 1;
        for (int i = 1; input[i]; i++)
            if (!isalnum(input[i])) valid = 0;
        return valid ? VALID : INVALID;
    }
    return INVALID;
}

int main() {
    printf("Enter variable name: ");
    yyparse();
    return 0;
}
```

Compile & Run:

```
yacc -d var_parser.y
gcc y.tab.c -o var_parser
echo "var123" | ./var_parser
```

Output:

```
Enter variable name: Valid Variable
```

6. Calculator using LEX & YACC

calc.y:

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

%token NUMBER
%left '+' '-'
```

```
%left '*' '/'

%%
expr: expr '+' expr { printf(" = %d\n", $1 + $3); }
    | expr '-' expr { printf(" = %d\n", $1 - $3); }
    | expr '*' expr { printf(" = %d\n", $1 * $3); }
    | expr '/' expr { if($3!=0) printf(" = %d\n", $1 / $3);
                      else printf("Error: Division by zero\n"); }
    | NUMBER        { printf("%d", $1); }
    ;
%%

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

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

calc.l:

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

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

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

Compile & Run:

```
yacc -d calc.y
lex calc.l
```

```
gcc y.tab.c lex.yy.c -o calc
echo "2 + 3 * 4" | ./calc
```

Output:

```
2+3*4 = 14
```

7. Three Address Code Generation

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

int tempCount = 1;

void generateTAC(char exp[]) {
    char stack[50][10];
    int top = -1;

    for (int i = 0; i < strlen(exp); i++) {
        char symbol = exp[i];
        if (isalnum(symbol)) {
            char operand[10];
            sprintf(operand, "%c", symbol);
            strcpy(stack[++top], operand);
        }
        else if (symbol == '+' || symbol == '-' || symbol == '*' || symbol == '/') {
            if (top < 1) {
                printf("Invalid Expression\n");
                return;
            }
            char op2[10], op1[10], result[10];
            strcpy(op2, stack[top--]);
            strcpy(op1, stack[top--]);
            sprintf(result, "t%d", tempCount++);
            printf("%s = %s %c %s\n", result, op1, symbol, op2);
            strcpy(stack[++top], result);
        }
    }
    if (top == 0)
        printf("Final Result: %s\n", stack[top]);
    else
```

```
        printf("Invalid Expression\n");
}

int main() {
    char postfix[50];
    printf("Enter the postfix expression: ");
    scanf("%s", postfix);
    printf("\nThree Address Code:\n");
    generateTAC(postfix);
    return 0;
}
```

Output:

```
text
Enter the postfix expression: ab+c*

Three Address Code:
t1 = a + b
t2 = t1 * c
Final Result: t2
```

8. Type Checker with Scope

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

#define MAX 100

struct SymbolTable {
    char var[20];
    int declared;
} table[MAX];

int count = 0;

int isDeclared(char *var) {
    for (int i = 0; i < count; i++) {
        if (strcmp(table[i].var, var) == 0)
            return table[i].declared;
    }
```

```
    return 0;
}

void declareVar(char *var) {
    strcpy(table[count].var, var);
    table[count].declared = 1;
    count++;
}

int main() {
    char code[MAX][MAX];
    int n;

    printf("Enter number of lines of code: ");
    scanf("%d", &n);
    getchar();

    printf("Enter the code lines:\n");
    for (int i = 0; i < n; i++) {
        fgets(code[i], MAX, stdin);
        code[i][strcspn(code[i], "\n")] = 0;
    }

    for (int i = 0; i < n; i++) {
        char word1[20], word2[20], word3[20];
        if (sscanf(code[i], "int %s", word1) == 1) {
            if (word1[strlen(word1) - 1] == ';')
                word1[strlen(word1) - 1] = '\0';
            declareVar(word1);
            printf("Line %d: Variable '%s' declared\n", i + 1, word1);
        }
        else if (sscanf(code[i], "%s = %s", word1, word2) == 2) {
            if (!isDeclared(word1))
                printf("Line %d: Error - Variable '%s' used before declaration\n", i + 1, word
1);
            else if (isalpha(word2[0]) && !isDeclared(word2))
                printf("Line %d: Error - Variable '%s' used before declaration\n", i + 1, word
2);
            else
                printf("Line %d: Assignment is semantically correct\n", i + 1);
        }
    }
```

```
    return 0;
}
```

Output:

```
Enter number of lines of code: 5
Enter the code lines:
int a;
a = 10;
b = 5;
int b;
a = b;

Line 1: Variable 'a' declared
Line 2: Assignment is semantically correct
Line 3: Error - Variable 'b' used before declaration
Line 4: Variable 'b' declared
Line 5: Assignment is semantically correct
```

9. Code Optimization

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

struct Instruction {
    char result[10];
    char arg1[10];
    char op[5];
    char arg2[10];
};

int isConstant(char *str) {
    for (int i = 0; str[i] != '\0'; i++) {
        if (!isdigit(str[i]))
            return 0;
    }
    return 1;
}

int main() {
```

```c
int n;
printf("Enter number of intermediate code statements: ");
scanf("%d", &n);

struct Instruction ic[20];
printf("\nEnter the intermediate code in format (result = arg1 op arg2):\n");
for (int i = 0; i < n; i++) {
    printf("Instruction %d: ", i + 1);
    scanf("%s = %s %s %s", ic[i].result, ic[i].arg1, ic[i].op, ic[i].arg2);
}

printf("\n=== Before Optimization ===\n");
for (int i = 0; i < n; i++)
    printf("%s = %s %s %s\n", ic[i].result, ic[i].arg1, ic[i].op, ic[i].arg2);

// Optimization Phase
for (int i = 0; i < n; i++) {
    // Constant Folding
    if (isConstant(ic[i].arg1) && isConstant(ic[i].arg2)) {
        int val1 = atoi(ic[i].arg1);
        int val2 = atoi(ic[i].arg2);
        int res = 0;

        if (strcmp(ic[i].op, "+") == 0) res = val1 + val2;
        else if (strcmp(ic[i].op, "-") == 0) res = val1 - val2;
        else if (strcmp(ic[i].op, "*") == 0) res = val1 * val2;
        else if (strcmp(ic[i].op, "/") == 0 && val2 != 0) res = val1 / val2;

        sprintf(ic[i].arg1, "%d", res);
        strcpy(ic[i].op, "");
        strcpy(ic[i].arg2, "");
    }

    // Constant Propagation
    for (int j = i + 1; j < n; j++) {
        if (strcmp(ic[i].op, "") == 0) {
            if (strcmp(ic[j].arg1, ic[i].result) == 0)
                strcpy(ic[j].arg1, ic[i].arg1);
            if (strcmp(ic[j].arg2, ic[i].result) == 0)
                strcpy(ic[j].arg2, ic[i].arg1);
        }
    }
```

```
    }

    // Dead Code Elimination
    int used[20] = {0};
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            if (strcmp(ic[i].result, ic[j].arg1) == 0 ||
                strcmp(ic[i].result, ic[j].arg2) == 0)
                used[i] = 1;

    printf("\n=== After Optimization ===\n");
    for (int i = 0; i < n; i++) {
        if (used[i] || i == n - 1) {
            if (strcmp(ic[i].op, "") == 0)
                printf("%s = %s\n", ic[i].result, ic[i].arg1);
            else
                printf("%s = %s %s %s\n", ic[i].result, ic[i].arg1, ic[i].op, ic[i].arg2);
        }
    }
    return 0;
}
```

Output:

```
Enter number of intermediate code statements: 5
Instruction 1: T1 = 2 + 3
Instruction 2: T2 = T1 * 5
Instruction 3: T3 = T2 + 0
Instruction 4: T4 = T3 + 4
Instruction 5: Z = T4 * 1

=== Before Optimization ===
T1 = 2 + 3
T2 = T1 * 5
T3 = T2 + 0
T4 = T3 + 4
Z = T4 * 1

=== After Optimization ===
T1 = 5
T2 = 5 * 5
T3 = 25
```

```
T4 = 25 + 4
Z = 29
```

10. Machine Code from Intermediate Code

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

struct Instruction {
    char result[10];
    char arg1[10];
    char op[5];
    char arg2[10];
};

int main() {
    int n;
    printf("Enter number of intermediate code instructions: ");
    scanf("%d", &n);

    struct Instruction ic[20];
    printf("\nEnter intermediate code in the format (result = arg1 op arg2):\n");
    for (int i = 0; i < n; i++) {
        printf("Instruction %d: ", i + 1);
        scanf("%s = %s %s %s", ic[i].result, ic[i].arg1, ic[i].op, ic[i].arg2);
    }

    printf("\n=== Generated Machine Code ===\n");
    for (int i = 0; i < n; i++) {
        printf("LOAD %s\n", ic[i].arg1);
        if (strcmp(ic[i].op, "+") == 0)
            printf("ADD %s\n", ic[i].arg2);
        else if (strcmp(ic[i].op, "-") == 0)
            printf("SUB %s\n", ic[i].arg2);
        else if (strcmp(ic[i].op, "*") == 0)
            printf("MUL %s\n", ic[i].arg2);
        else if (strcmp(ic[i].op, "/") == 0)
            printf("DIV %s\n", ic[i].arg2);
        else
            printf("NOP // Unknown operation\n");
        printf("STORE %s\n\n", ic[i].result);
```

```
    }
    return 0;
}
```

Output:

```
Enter number of intermediate code instructions: 3
Instruction 1: T1 = A + B
Instruction 2: T2 = T1 * C
Instruction 3: Z = T2 - D

=== Generated Machine Code ===
LOAD A
ADD B
STORE T1

LOAD T1
MUL C
STORE T2

LOAD T2
SUB D
STORE Z
```

11. Pattern Recognition with YACC

pattern.l:

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

%%
[A-Z][A-Za-z0-9]{5}     { return VALID; }
.|\n                    { return INVALID; }
%%

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

pattern.y:

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

```
#include <stdlib.h>

int yylex(void);
void yyerror(const char *s);
%}

%token VALID INVALID

%%
input:
      VALID      { printf("Valid pattern\n"); }
    | INVALID    { printf("Invalid pattern\n"); }
    ;
%%

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

int main() {
    printf("Enter a string: ");
    yyparse();
    return 0;
}
```

Compile & Run:

```
flex pattern.l
bison -d pattern.y
gcc lex.yy.c pattern.tab.c -o pattern
echo "ABC123" | ./pattern
```

Output:

```
Enter a string: Valid pattern
```

12. Lexical Analyzer for Comments, Operators, Keywords

lexer.l:

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

```
int keywords_count = 0, operators_count = 0, comments_count = 0;

char *keywords[] = {
    "int", "float", "char", "if", "else", "for", "while", "do",
    "return", "break", "continue", "void", "switch", "case", "default"
};

int is_keyword(const char *word) {
    for (int i = 0; i < sizeof(keywords)/sizeof(keywords[0]); i++)
        if (strcmp(word, keywords[i]) == 0)
            return 1;
    return 0;
}
%}

%%
"//".*                          { printf("Comment: %s\n", yytext); comments_count++; }
"/*"([^*]|\*+[^*/])*"*/"            { printf("Comment: %s\n", yytext); comments_count+
+; }

"=="|"!="|"<="|">="|"&&"|"||"|"="|"+"|"-"|"*"|"/"|"<"|">" {
                                    printf("Operator: %s\n", yytext);
                                    operators_count++;
                                }

[a-zA-Z_][a-zA-Z0-9_]* {
                                    if (is_keyword(yytext)) {
                                        printf("Keyword: %s\n", yytext);
                                        keywords_count++;
                                    }
                                }

[ \t\n]+                          ;   /* ignore whitespace */
.                                 ;   /* ignore everything else */
%%

int yywrap() { return 1; }

int main() {
    printf("Enter code (Ctrl+D to end):\n");
    yylex();
```

```
    printf("\nSummary:\n");
    printf("Keywords: %d\n", keywords_count);
    printf("Operators: %d\n", operators_count);
    printf("Comments: %d\n", comments_count);
    return 0;
}
```

test.c:

```
int main() {
    int a = 10; // variable declaration
    if (a > 5) {
        a = a + 1;
    }
    /* This is a block comment */
    return a;
}
```

Compile & Run:

```
flex lexer.l
gcc lex.yy.c -o lexer -ll
./lexer < test.c
```

Output:

```
Keyword: int
Keyword: int
Operator: =
Comment: // variable declaration
Keyword: if
Operator: >
Operator: =
Operator: +
Comment: /* This is a block comment */
Keyword: return

Summary:
Keywords: 4
Operators: 4
```

Comments: 2

13. Control Structure Recognition

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

int isValidFor(char *s) {
    return strstr(s, "for(") == s || strstr(s, "for (") == s;
}

int isValidWhile(char *s) {
    return strstr(s, "while(") == s || strstr(s, "while (") == s;
}

int isValidIf(char *s) {
    return strstr(s, "if(") == s || strstr(s, "if (") == s;
}

int isValidElseIf(char *s) {
    return strstr(s, "else if(") == s || strstr(s, "else if (") == s;
}

int isValidElse(char *s) {
    return strstr(s, "else") == s;
}

int isValidSwitch(char *s) {
    return strstr(s, "switch(") == s || strstr(s, "switch (") == s;
}

int main() {
    char input[200];
    printf("Enter a C control structure statement:\n");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = '\0';

    if (isValidFor(input))
        printf("Valid 'for' loop syntax.\n");
    else if (isValidWhile(input))
        printf("Valid 'while' loop syntax.\n");
```

```
    else if (isValidElseIf(input))
        printf("Valid 'else if' structure.\n");
    else if (isValidIf(input))
        printf("Valid 'if' structure.\n");
    else if (isValidElse(input))
        printf("Valid 'else' structure.\n");
    else if (isValidSwitch(input))
        printf("Valid 'switch' case structure.\n");
    else
        printf("Invalid or unrecognized control structure syntax.\n");

    return 0;
}
```

Output:

```
Enter a C control structure statement:
for(i=0;i<10;i++)
Valid 'for' loop syntax.
```

14. Vowel and Consonant Counter

vowels.l:

```
%{
#include <stdio.h>
int vowels = 0, consonants = 0;
%}

%%
[AEIOUaeiou]        { vowels++; }
[A-Za-z]            { consonants++; }
[ \t\n]+            ;    /* ignore spaces, tabs, and newlines */
.                   ;    /* ignore any other characters */
%%

int yywrap() { return 1; }

int main() {
    printf("Enter a string: ");
    yylex();
    printf("\nTotal vowels: %d\n", vowels);
```

```
    printf("Total consonants: %d\n", consonants);
    return 0;
}
```

Compile & Run:

```
flex vowels.l
gcc lex.yy.c -o vowels -ll
echo "Hello World" | ./vowels
```

Output:

```
Enter a string:
Total vowels: 3
Total consonants: 7
```

15. Machine Code from Postfix

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

#define MAX 100

char stack[MAX][10];
int top = -1;
int tempVar = 1;

void push(char *str) {
    strcpy(stack[++top], str);
}

char* pop() {
    return stack[top--];
}

int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

int main() {
    char postfix[100];
```

```
    printf("Enter postfix expression: ");
    fgets(postfix, sizeof(postfix), stdin);

    for (int i = 0; postfix[i] != '\0'; i++) {
        if (isspace(postfix[i]))
            continue;

        if (isalnum(postfix[i])) {
            char operand[2];
            operand[0] = postfix[i];
            operand[1] = '\0';
            push(operand);
        }
        else if (isOperator(postfix[i])) {
            char operand2[10], operand1[10], temp[10];
            strcpy(operand2, pop());
            strcpy(operand1, pop());
            sprintf(temp, "T%d", tempVar++);

            printf("LOAD %s\n", operand1);
            switch (postfix[i]) {
                case '+': printf("ADD %s\n", operand2); break;
                case '-': printf("SUB %s\n", operand2); break;
                case '*': printf("MUL %s\n", operand2); break;
                case '/': printf("DIV %s\n", operand2); break;
            }
            printf("STORE %s\n", temp);
            push(temp);
        }
    }
    printf("\nFinal Result stored in: %s\n", pop());
    return 0;
}
```

Output:

```
Enter postfix expression: A B C * + D /
LOAD B
MUL C
STORE T1
LOAD A
ADD T1
```

STORE T2
LOAD T2
DIV D
STORE T3

Final Result stored in: T3