# Cd lab

Saturday, November 30, 2024 6:11 PM

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

```
 #include <stdio.h>
#include <ctype.h>
#include <string.h>
 // Function prototypes
void lexicalAnalyzer(const char *input);

int main() {
   char input[100];

   // Getting input from the user
   printf("Enter the input string: ");
   fgets(input, sizeof(input), stdin);

   // Analyzing the input
   lexicalAnalyzer(input);

   return 0;
}

// Function to analyze the input string
void lexicalAnalyzer(const char *input) {
   int i = 0;
   while (input[i] != '\0') {
      // Skip white spaces
      if (isspace(input[i])) {
         i++;
         continue;
      }

      // Recognizing identifiers (variables that start with a letter)
      if (isalpha(input[i])) {
         printf("Identifier: ");
         while (isalpha(input[i]) || isdigit(input[i]) || input[i] == '_') {
            printf("%c", input[i]);
            i++;
         }
         printf("\n");
      }
      // Recognizing constants (numeric values)
      else if (isdigit(input[i])) {
         printf("Constant: ");
         while (isdigit(input[i])) {
            printf("%c", input[i]);
            i++;
         }
         printf("\n");
      }
      // For any other characters
      else {
         printf("Unknown token: %c\n", input[i]);
         i++;
```

```
        }
    }
}
```

**Output:**

**Identifier: int**

**Identifier: a**

**Constant: 123**

**Identifier: float**

**Identifier: b**

**Constant: 45**

**Unknown token: .**

**Constant: 67**

**Identifier: char**

**Identifier: c**

**Unknown token: '**

**Unknown token: x**

**Unknown token: '**

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

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

// Structure for a symbol table entry
typedef struct SymbolTableEntry {
    char name[50];
    char type[50];
    int value;
    struct SymbolTableEntry *next;
} SymbolTableEntry;

SymbolTableEntry *head = NULL;

// Function to insert a new entry into the symbol table
void insert(char *name, char *type, int value) {
    SymbolTableEntry *newEntry = (SymbolTableEntry *)malloc(sizeof(SymbolTableEntry));
    strcpy(newEntry->name, name);
    strcpy(newEntry->type, type);
    newEntry->value = value;
    newEntry->next = head;
    head = newEntry;
    printf("Inserted %s\n", name);
}

// Function to delete an entry from the symbol table
void delete(char *name) {
    SymbolTableEntry *temp = head, *prev = NULL;
    if (temp != NULL && strcmp(temp->name, name) == 0) {
        head = temp->next;
        free(temp);
        printf("Deleted %s\n", name);
        return;
    }
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
```

```
    }
    if (temp == NULL) {
        printf("Symbol %s not found\n", name);
        return;
    }
    prev->next = temp->next;
    free(temp);
    printf("Deleted %s\n", name);
}

// Function to search for an entry in the symbol table
SymbolTableEntry* search(char *name) {
    SymbolTableEntry *temp = head;
    while (temp != NULL) {
        if (strcmp(temp->name, name) == 0)
            return temp;
        temp = temp->next;
    }
    return NULL;
}

// Function to modify an entry in the symbol table
void modify(char *name, char *newType, int newValue) {
    SymbolTableEntry *entry = search(name);
    if (entry != NULL) {
        strcpy(entry->type, newType);
        entry->value = newValue;
        printf("Modified %s\n", name);
    } else {
        printf("Symbol %s not found\n", name);
    }
}

// Function to display the symbol table
void display() {
    SymbolTableEntry *temp = head;
    printf("Symbol Table:\n");
    printf("Name\tType\tValue\n");
    while (temp != NULL) {
        printf("%s\t%s\t%d\n", temp->name, temp->type, temp->value);
        temp = temp->next;
    }
}

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

    modify("x", "double", 100);
    display();

    delete("y");
    display();

    SymbolTableEntry *entry = search("x");
```

```
    if (entry != NULL)
        printf("Found %s: %s, %d\n", entry->name, entry->type, entry->value);
    else
        printf("Symbol not found\n");

    return 0;
}
```

**Output**

**Inserted x**

**Inserted y**

**Symbol Table:**

**Name Type Value**

**y float 20**

**x int 10**

**Modified x**

**Symbol Table:**

**Name Type Value**

**y float 20**

**x double 100**

**Deleted y**

**Symbol Table:**

**Name Type Value**

**x double 100**

**Found x: x, double, 100**

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

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

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

%%

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

%%

int yywrap() {
    return 1;
}

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

int main() {
    yylex();
```

```
   return 0;
}
```

**Output**

**Identifier: int**

**Identifier: x**

**Constant: 123**

**Unknown token: ;**

**Identifier: float**

**Identifier: y**

**Constant: 4**

**Unknown token: .**

**Constant: 56**

**Unknown token: ;**

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

**lex**

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

%%

[0-9]+       { yylval = atoi(yytext); return NUMBER; }
[ \t]        ;
\n           { return 0; }
"+"          { return PLUS; }
"-"          { return MINUS; }
"*"          { return MULTIPLY; }
"/"          { return DIVIDE; }
.            { return yytext[0]; }

%%
```

**Yacc:**

```
int yywrap() {
   return 1;
}
%{
#include <stdio.h>
#include <stdlib.h>

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

%token NUMBER
%token PLUS MINUS MULTIPLY DIVIDE

%%

expr    : expr PLUS term      { printf("%d + %d = %d\n", $1, $3, $1 + $3); $$ = $1 + $3; }
      | expr MINUS term     { printf("%d - %d = %d\n", $1, $3, $1 - $3); $$ = $1 - $3; }
      | term                { $$ = $1; }
      ;

term    : term MULTIPLY factor { printf("%d * %d = %d\n", $1, $3, $1 * $3); $$ = $1 * $3; }
      | term DIVIDE factor   { if ($3 == 0) {
```

```
                    yyerror("division by zero");
                    exit(1);
                  } else {
                    printf("%d / %d = %d\n", $1, $3, $1 / $3); $$ = $1 / $3; } }
    | factor          { $$ = $1; }
    ;

factor  : NUMBER          { $$ = $1; }
    ;

%%

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

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

**Output**

**Enter an arithmetic expression: 3 + 4 * 2 - 1**

**3 + 8 = 11**

**11 - 1 = 10**

**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.**

**lex**

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

%%

[a-zA-Z][a-zA-Z0-9]*  { return IDENTIFIER; }
[ \t\n]+          ;  // Ignore whitespace
.                 { return yytext[0]; } // Unknown character

%%

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

**yacc**

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

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

%token IDENTIFIER

%%
```

```
input   : /* empty */
        | input line
        ;

line    : IDENTIFIER  { printf("Valid identifier: %s\n", yytext); }
        | error       { yyerror("Invalid identifier"); yyclearin; }
        ;

%%

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

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

**Output:**

**Enter a variable name: myVar123**

**Valid identifier: myVar123**

**Enter a variable name: 123Var**

**Invalid identifier**

**6)Use LEX and YACC tools to implement a native calculator**

**Lex**

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

%%

[0-9]+          { yylval = atoi(yytext); return NUMBER; }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return MULTIPLY; }
"/"             { return DIVIDE; }
[ \t\n]         ;  // Ignore whitespace
.               { return yytext[0]; }  // For any other character

%%

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

**Yacc**

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

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

```
%token NUMBER
%token PLUS MINUS MULTIPLY DIVIDE

%%

calculation:
   expression '\n'   { printf("Result: %d\n", $1); }
   ;

expression:
   expression PLUS term     { $$ = $1 + $3; }
   | expression MINUS term   { $$ = $1 - $3; }
   | term                { $$ = $1; }
   ;

term:
   term MULTIPLY factor     { $$ = $1 * $3; }
   | term DIVIDE factor     { if ($3 == 0) {
                            yyerror("division by zero");
                            exit(1);
                           } else {
                            $$ = $1 / $3; } }
   | factor              { $$ = $1; }
   ;

factor:
   NUMBER                { $$ = $1; }
   ;

%%

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

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

**Output**

**Enter an arithmetic expression: 3 + 4 * 2 - 1**

**Result: 10**

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

**Lex**

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

%%

[0-9]+          { yylval = atoi(yytext); return NUMBER; }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return MULTIPLY; }
"/"             { return DIVIDE; }
```

```
[ \t\n]          ; // Ignore whitespace
.                { return yytext[0]; }

%%

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

**yacc**

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

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

int tempCount = 0;
char tacCode[100][100];
int codeIndex = 0;

void addTacCode(char *code) {
   strcpy(tacCode[codeIndex++], code);
}

char* newTemp() {
   char *temp = (char *)malloc(10 * sizeof(char));
   sprintf(temp, "t%d", tempCount++);
   return temp;
}
%}

%token NUMBER
%token PLUS MINUS MULTIPLY DIVIDE

%%

input:
   expr '\n'   { printf("Three-address code:\n");
                 for (int i = 0; i < codeIndex; i++) {
                    printf("%s\n", tacCode[i]);
                 }
               }
   ;

expr:
   expr PLUS term  {
                    char code[100];
                    char *temp = newTemp();
                    sprintf(code, "%s = %s + %s", temp, $1, $3);
                    addTacCode(code);
                    $$ = temp;
                   }
   | expr MINUS term {
                    char code[100];
                    char *temp = newTemp();
```

```
                sprintf(code, "%s = %s - %s", temp, $1, $3);
                addTacCode(code);
                $$ = temp;
              }
  | term        { $$ = $1; }
  ;

term:
  term MULTIPLY factor {
                char code[100];
                char *temp = newTemp();
                sprintf(code, "%s = %s * %s", temp, $1, $3);
                addTacCode(code);
                $$ = temp;
               }
  | term DIVIDE factor {
                char code[100];
                char *temp = newTemp();
                sprintf(code, "%s = %s / %s", temp, $1, $3);
                addTacCode(code);
                $$ = temp;
               }
  | factor      { $$ = $1; }
  ;

factor:
  NUMBER       { char *temp = newTemp();
              sprintf(temp, "%d", $1);
              $$ = temp; }
  ;

%%

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

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

**output**

**Enter an arithmetic expression: 3 + 4 * 2 - 1**

**Three-address code:**

**t0 = 3**

**t1 = 4**

**t2 = 2**

**t3 = t1 * t2**

**t4 = t0 + t3**

**t5 = t4 – t1**

**8)--------->manual -5**

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

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

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

typedef struct {
    char code[100];
    bool is_dead;
} Instruction;

void optimize(Instruction* instructions, int count) {
    // Constant Folding
    for (int i = 0; i < count; i++) {
        if (strstr(instructions[i].code, "2 + 2")) {
            strcpy(instructions[i].code, "4");
        }
    }

    // Dead Code Elimination
    for (int i = 0; i < count; i++) {
        if (strstr(instructions[i].code, "unused")) {
            instructions[i].is_dead = true;
        }
    }
}

int main() {
    Instruction instructions[] = {
        {"int x = 2 + 2;", false},
        {"int y = x;", false},
        {"int unused = 10;", false},
        {"y = y + 1;", false}
    };
    int count = sizeof(instructions) / sizeof(instructions[0]);

    optimize(instructions, count);

    printf("Optimized Code:\n");
    for (int i = 0; i < count; i++) {
        if (!instructions[i].is_dead) {
            printf("%s\n", instructions[i].code);
        }
    }

    return 0;
}
```

**Output**

**Optimized Code:**

**int x = 4;**

**int y = x;**

**y = y + 1;**

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

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

typedef struct {
```

```
    char result[10];
    char op1[10];
    char op2[10];
    char operator;
} Instruction;

void generate_machine_code(Instruction* instructions, int count) {
    for (int i = 0; i < count; i++) {
        if (instructions[i].operator == '+') {
            printf("MOV EAX, %s\n", instructions[i].op1);
            printf("ADD EAX, %s\n", instructions[i].op2);
            printf("MOV %s, EAX\n", instructions[i].result);
        } else if (instructions[i].operator == '*') {
            printf("MOV EAX, %s\n", instructions[i].op1);
            printf("IMUL EAX, %s\n", instructions[i].op2);
            printf("MOV %s, EAX\n", instructions[i].result);
        }
    }
}

int main() {
    Instruction instructions[] = {
        {"t1", "a", "b", '+'},
        {"t2", "t1", "c", '*'},
        {"d", "t2", "", '\0'}
    };
    int count = sizeof(instructions) / sizeof(instructions[0]);

    generate_machine_code(instructions, count);

    return 0;
}
```

**Output**

```
MOV EAX, a
ADD EAX, b
MOV t1, EAX
MOV EAX, t1
IMUL EAX, c
MOV t2, EAX
MOV d, t2
```

**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.**

**Lex**

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

%%
[A-Z][A-Za-z0-9]{5}    { return IDENTIFIER; }
\n                     { return '\n'; }
.                      { return yytext[0]; }
%%

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

**Yacc**

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

%token IDENTIFIER

%%
statements:
  statement '\n' { printf("Valid statement: %s\n", yytext); }
  |
  statement '\n' statements
  ;

statement:
  IDENTIFIER
  ;

%%

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

int main() {
  printf("Enter statements:\n");
  return yyparse();
}
```

**Output**

**Enter statements:**

**A12345**

**A123ab**

**Valid statement: A12345**

**Valid statement: A123ab**

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

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

void handle_comment(char* yytext) {
  printf("Comment: %s\n", yytext);
}

void handle_operator(char* yytext) {
  printf("Operator: %s\n", yytext);
}

void handle_keyword(char* yytext) {
  printf("Keyword: %s\n", yytext);
}
%}
```

```
/* Definitions */
%%
"/*".*"*/"     { handle_comment(yytext); }
"//".*         { handle_comment(yytext); }

"+"            { handle_operator(yytext); }
"-"            { handle_operator(yytext); }
"*"            { handle_operator(yytext); }
"/"            { handle_operator(yytext); }
"%"            { handle_operator(yytext); }
"="            { handle_operator(yytext); }
"=="           { handle_operator(yytext); }
"!="           { handle_operator(yytext); }
"<"            { handle_operator(yytext); }
"<="           { handle_operator(yytext); }
">"            { handle_operator(yytext); }
">="           { handle_operator(yytext); }

"if"           { handle_keyword(yytext); }
"else"         { handle_keyword(yytext); }
"for"          { handle_keyword(yytext); }
"while"        { handle_keyword(yytext); }
"do"           { handle_keyword(yytext); }
"return"       { handle_keyword(yytext); }
"int"          { handle_keyword(yytext); }
"float"        { handle_keyword(yytext); }
"char"         { handle_keyword(yytext); }
"void"         { handle_keyword(yytext); }
"include"      { handle_keyword(yytext); }

/* Catch-all rule for anything else (whitespace, identifiers, numbers, etc.) */
.              ;

%%

int main(int argc, char** argv) {
    yylex();
    return 0;
}

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

**Sample input :**

```
#include <stdio.h>
 // This is a line comment
int main() {
    int a = 5;
    float b = 3.14;
    if (a == b) {
        a = a + 1;
    }
    return 0;
}
```

**Output:**

**Keyword: include**

**Comment: // This is a line comment**

**Keyword: int**

**Keyword: int**

**Operator: =**

**Operator: +**

**Keyword: float**

**Keyword: if**

**Operator: ==**

**Operator: =**

**Keyword: return**

**Operator: =**

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

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

int vowels = 0, consonants = 0;
%}

%%

[AEIOUaeiou]        { vowels++; }
[a-zA-Z]            { consonants++; }
[ \t\n]+            ;  // Ignore whitespace
.                   ;  // Ignore any other character

%%

int yywrap() {
   return 1;
}

int main() {
   printf("Enter a string: ");
   yylex();
   printf("Total vowels: %d\n", vowels);
   printf("Total consonants: %d\n", consonants);
   return 0;
}
```

**Output:**

**Enter a string: Hello World**

**Total vowels: 3**

**Total consonants: 7**

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

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

#define MAX 100

typedef struct {
char op;
```

```
int left;
int right;
int result;
} Instruction;

Instruction code[MAX];
int code_index = 0;
int reg = 0;

void generate(char op, int left, int right, int result) {
code[code_index].op = op;
code[code_index].left = left;
code[code_index].right = right;
code[code_index].result = result;
code_index++;
}

void print_machine_code() {
for (int i = 0; i < code_index; i++) {
printf("R%d = R%d %c R%d\n", code[i].result, code[i].left, code[i].op, code[i].right);
}
}

int evaluate_postfix(char* postfix) {
int stack[MAX];
int top = -1;

for (int i = 0; postfix[i] != '\0'; i++) {
if (postfix[i] >= '0' && postfix[i] <= '9') {
 stack[++top] = postfix[i] - '0';
} else {
 int right = stack[top--];
int left = stack[top--];
 int result = reg++;
generate(postfix[i], left, right, result);
stack[++top] = result;
}
}

return stack[top];
}

int main() {
char postfix[MAX];

printf("Enter the postfix expression: ");
scanf("%s", postfix);

int result = evaluate_postfix(postfix);

printf("Machine code:\n");
print_machine_code();

printf("Final result in register: R%d\n", result);

return 0;
```

```
}
```

**Output**

**Enter the postfix expression: 34+5***

**Machine code:**

**R0 = 3 + 4**

**R1 = R0 * 5**

**Final result in register: R1**

**16)Write a LEX program to scan reserved words, variables and operators of C language**

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

void printToken(char *token, char *type) {
   printf("%s: %s\n", type, token);
}

%}

%%

"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|
"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"return"|"short"|
"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"union"|"unsigned"|"void"|
"volatile"|"while"        { printToken(yytext, "Reserved Word"); }
[a-zA-Z_][a-zA-Z0-9_]*     { printToken(yytext, "Variable"); }
"+"|"-"|"*"|"/"|"="|"=="|"!="|"++"|"- -"|"&&"|"||"|"!"|"&"|"|"|"<"|">"|"<="|">=" {
printToken(yytext, "Operator"); }
[ \t\n]                  ;  // Ignore whitespace
.                       { printToken(yytext, "Unknown"); }

%%

int yywrap() {
   return 1;
}

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

**Output**

**Enter a string: int main() { int x = 10; if (x > 5) x++; }**

**Reserved Word: int**

**Variable: main**

**Unknown: (**

**Unknown: )**

**Unknown: {**

**Reserved Word: int**

**Variable: x**

**Operator: =**

**Variable: 10**

**Unknown: ;**

**Reserved Word: if**

**Unknown: (**

**Variable: x**

**Operator: >**

**Variable: 5**

**Unknown: )**

**Variable: x**

**Operator: ++**

**Unknown: ;**

**Unknown: }**