**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>

int is\_identifier(char \*str) {

if (!isalpha(str[0]))

return 0;

for (int i = 1; str[i] != '\0'; i++) {

if (!isalnum(str[i]))

return 0;

}

return 1;

}

int is\_constant(char \*str) {

for (int i = 0; str[i] != '\0'; i++) {

if (!isdigit(str[i]))

return 0;

}

return 1;

}

void lexical\_analyzer(char \*input) {

int i = 0;

char current\_token[100];

int j = 0;

while (input[i] != '\0') {

if (isspace(input[i])) {

i++;

continue;

}

j = 0;

while (isalnum(input[i])) {

current\_token[j++] = input[i++];

}

current\_token[j] = '\0';

if (is\_identifier(current\_token)) {

printf("Identifier: %s\n", current\_token);

} else if (is\_constant(current\_token)) {

printf("Constant: %s\n", current\_token);

} else {

printf("Invalid token: %s\n", current\_token);

}

}

}

int main() {

char input[100];

printf("Enter a string to analyze: ");

fgets(input, sizeof(input), stdin);

input[strcspn(input, "\n")] = '\0';

lexical\_analyzer(input);

return 0;

}

**OUTPUT:**

Identifier: age

Constant: 123

Identifier: height

Invalid token: 45x4

Invalid token: \_id

Constant: 789

---------------------------------------------------------------------------------------------------------------------------------------------

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

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define SIZE 10

typedef struct Symbol {

char name[50];

int value;

int isOccupied;

} Symbol;

Symbol hashTable[SIZE];

int hash(char \*str) {

int hashValue = 0;

int length = strlen(str);

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

hashValue = (hashValue + str[i]) % SIZE;

}

return hashValue;

}

void initTable() {

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

hashTable[i].isOccupied = 0;

}

}

void insert(char \*name, int value) {

int index = hash(name);

while (hashTable[index].isOccupied) {

if (strcmp(hashTable[index].name, name) == 0) {

printf("Error: Symbol '%s' already exists!\n", name);

return;

}

index = (index + 1) % SIZE;

}

strcpy(hashTable[index].name, name);

hashTable[index].value = value;

hashTable[index].isOccupied = 1;

printf("Inserted symbol: %s with value %d\n", name, value);

}

void delete(char \*name) {

int index = hash(name);

while (hashTable[index].isOccupied) {

if (strcmp(hashTable[index].name, name) == 0) {

hashTable[index].isOccupied = 0;

printf("Deleted symbol: %s\n", name);

return;

}

index = (index + 1) % SIZE;

}

printf("Error: Symbol '%s' not found!\n", name);

}

void search(char \*name) {

int index = hash(name);

while (hashTable[index].isOccupied) {

if (strcmp(hashTable[index].name, name) == 0) {

printf("Symbol '%s' found with value %d\n", name, hashTable[index].value);

return;

}

index = (index + 1) % SIZE;

}

printf("Error: Symbol '%s' not found!\n", name);

}

void modify(char \*name, int newValue) {

int index = hash(name);

while (hashTable[index].isOccupied) {

if (strcmp(hashTable[index].name, name) == 0) {

hashTable[index].value = newValue;

printf("Modified symbol '%s' to new value %d\n", name, newValue);

return;

}

index = (index + 1) % SIZE;

}

printf("Error: Symbol '%s' not found!\n", name);

}

void displayTable() {

printf("\nSymbol Table:\n");

printf("Name\tValue\n");

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

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

if (hashTable[i].isOccupied) {

printf("%s\t%d\n", hashTable[i].name, hashTable[i].value);

}

}

}

int main() {

initTable();

insert("x", 10);

insert("y", 20);

insert("z", 30);

insert("alpha", 50);

search("x");

search("y");

search("alpha");

search("beta");

modify("y", 25);

modify("z", 35);

delete("alpha");

delete("beta");

displayTable();

return 0;

}

**OUTPUT**:

Inserted symbol: x with value 10

Inserted symbol: y with value 20

Inserted symbol: z with value 30

Inserted symbol: alpha with value 50

Symbol 'x' found with value 10

Symbol 'y' found with value 20

Symbol 'alpha' found with value 50

Error: Symbol 'beta' not found!

Modified symbol 'y' to new value 25

Modified symbol 'z' to new value 35

Deleted symbol: alpha

Error: Symbol 'beta' not found!

Symbol Table:

Name Value

---------------

x 10

y 25

z 35

----------------------------------------------------------------------------------------------------------------------------------------------

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

%{

#include <stdio.h>

#include <stdlib.h>

%}

%%

[0-9]+ { printf("Constant: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

"+" { printf("Operator: +\n"); }

"-" { printf("Operator: -\n"); }

"\*" { printf("Operator: \*\n"); }

"/" { printf("Operator: /\n"); }

[ \t\n]+ { /\* ignore whitespace \*/ }

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

%%

int main() {

yylex();

return 0;

}

**OUTPUT**:

Identifier: x

Constant: 10

Operator: +

Identifier: y

----------------------------------------------------------------------------------------------------------------------------------------------

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

**Create Lex Specification (lexer.l)**

%{

#include "y.tab.h"

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MULT; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

[ \t\n] { /\* ignore whitespace \*/ }

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

**Create YACC Grammar (parser.y)**

%{

#include <stdio.h>

#include <stdlib.h>

extern int yylex();

extern int yylval;

void yyerror(const char \*s);

%}

%token NUMBER PLUS MINUS MULT DIV LPAREN RPAREN

%%

expression:

expression PLUS expression { printf("%d + %d\n", $1, $3); }

| expression MINUS expression { printf("%d - %d\n", $1, $3); }

| expression MULT expression { printf("%d \* %d\n", $1, $3); }

| expression DIV expression { printf("%d / %d\n", $1, $3); }

| LPAREN expression RPAREN { $$ = $2; }

| NUMBER { $$ = $1; }

;

%%

int main() {

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

yyparse();

return 0;

}

void yyerror(const char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

}

INPUT:

(3 + 2) \* 5

**OUTPUT**:

Enter an arithmetic expression:

15

-----------------------------------------------------------------------------------------------------------------------------------------

**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**

**Create Lex Specification**

%{

#include "y.tab.h"

%}

%%

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

[ \t\n] { /\* Ignore whitespaces \*/ }

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

**Create YACC Grammar**

%{

#include <stdio.h>

#include <stdlib.h>

extern int yylex();

extern int yylval;

void yyerror(const char \*s);

%}

%token VARIABLE

%%

valid\_variable:

VARIABLE { printf("Valid variable: %s\n", yytext); }

;

%%

int main() {

printf("Enter a string: ");

yyparse();

return 0;

}

void yyerror(const char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

}

**INPUT**:

myVariable123

**OUTPUT**:

Enter a string: myVariable123

Valid variable: myVariable123

-----------------------------------------------------------------------------------------------------------------------------------------------

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

**Lex Specification File**

%{

#include "y.tab.h"

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MULT; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

[ \t\n] { /\* Ignore whitespace \*/ }

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

**YACC Grammar File**

%{

#include <stdio.h>

#include <stdlib.h>

extern int yylex();

extern int yylval;

void yyerror(const char \*s);

int result = 0;

%}

%token NUMBER PLUS MINUS MULT DIV LPAREN RPAREN

%%

calculation:

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

;

expression:

expression PLUS term { $$ = $1 + $3; }

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

| term

;

term:

term MULT factor { $$ = $1 \* $3; }

| term DIV factor { $$ = $1 / $3; }

| factor

;

factor:

NUMBER { $$ = $1; }

| LPAREN expression RPAREN { $$ = $2; }

;

%%

int main() {

printf("Enter an arithmetic expression: ");

yyparse(); // Start parsing the input expression

return 0;

}

void yyerror(const char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

}

**OUTPUT**:

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

Result: 7

-------------------------------------------------------------------------------------------------------------------------------------------

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

**Lex Specification**

%{

#include "y.tab.h"

#include <stdio.h>

#include <stdlib.h>

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MULT; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

[ \t\n] { /\* Ignore whitespace \*/ }

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

**YACC Specification**

%{

#include <stdio.h>

#include <stdlib.h>

int temp\_count = 0;

int result\_count = 0;

extern int yylex();

extern int yylval;

void yyerror(const char \*s);

void generate\_code(char\* op, char\* operand1, char\* operand2);

%}

%token NUMBER PLUS MINUS MULT DIV LPAREN RPAREN

%%

program:

expression { printf("\n"); }

;

expression:

expression PLUS term {

char temp[10];

sprintf(temp, "t%d", temp\_count++);

generate\_code("+", $1, $3);

$$ = temp;

}

| expression MINUS term {

char temp[10];

sprintf(temp, "t%d", temp\_count++);

generate\_code("-", $1, $3);

$$ = temp;

}

| term

;

term:

term MULT factor {

char temp[10];

sprintf(temp, "t%d", temp\_count++);

generate\_code("\*", $1, $3);

$$ = temp;

}

| term DIV factor {

char temp[10];

sprintf(temp, "t%d", temp\_count++);

generate\_code("/", $1, $3);

$$ = temp;

}

| factor

;

factor:

NUMBER {

char temp[10];

sprintf(temp, "%d", $1);

$$ = temp;

}

| LPAREN expression RPAREN { $$ = $2; }

;

%%

void yyerror(const char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

}

void generate\_code(char\* op, char\* operand1, char\* operand2) {

char temp[10];

sprintf(temp, "t%d", temp\_count++);

printf("%s = %s %s %s\n", temp, operand1, op, operand2);

}

int main() {

printf("Enter an arithmetic expression: ");

yyparse(); // Start parsing the input expression

return 0;

}

**OUTPUT:**

Enter an arithmetic expression: 10 / (2 + 3)

t0 = 2 + 3

t1 = 10 / t0

Result: t1

------------------------------------------------------------------------------------------------------------------------------

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

**Lex Specification**

%{

#include "y.tab.h"

#include <stdio.h>

#include <stdlib.h>

%}

%%

"int" { return INT; }

"float" { return FLOAT; }

[a-zA-Z][a-zA-Z0-9]\* { yylval.str = strdup(yytext); return IDENTIFIER; }

[0-9]+ { yylval.num = atoi(yytext); return NUMBER; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MULT; }

"/" { return DIV; }

"=" { return ASSIGN; }

";" { return SEMICOLON; }

"(" { return LPAREN; }

")" { return RPAREN; }

[ \t\n] { /\* Ignore whitespaces \*/ }

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

**YACC Specification**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_VARS 100

typedef struct {

char \*name;

char \*type;

} Variable;

Variable symbol\_table[MAX\_VARS];

int var\_count = 0;

void yyerror(const char \*s);

int is\_declared(char \*name);

int check\_type(char \*var\_name, char \*expected\_type);

%}

%token INT FLOAT IDENTIFIER NUMBER ASSIGN PLUS MINUS MULT DIV SEMICOLON LPAREN RPAREN

%%

program:

declarations statements

;

declarations:

declarations declaration

| /\* empty \*/

;

declaration:

INT IDENTIFIER SEMICOLON {

if (is\_declared($2)) {

printf("Error: Variable '%s' already declared.\n", $2);

} else {

symbol\_table[var\_count].name = strdup($2);

symbol\_table[var\_count].type = "int";

var\_count++;

}

}

| FLOAT IDENTIFIER SEMICOLON {

if (is\_declared($2)) {

printf("Error: Variable '%s' already declared.\n", $2);

} else {

symbol\_table[var\_count].name = strdup($2);

symbol\_table[var\_count].type = "float";

var\_count++;

}

}

;

statements:

statements statement

| /\* empty \*/

;

statement:

IDENTIFIER ASSIGN expression SEMICOLON {

if (!is\_declared($1)) {

printf("Error: Variable '%s' is not declared.\n", $1);

} else {

char \*var\_type = get\_var\_type($1);

if (check\_type($1, var\_type) == 0) {

printf("Error: Type mismatch in assignment to '%s'.\n", $1);

}

}

}

| expression SEMICOLON

;

expression:

NUMBER { $$ = $1; }

| IDENTIFIER { $$ = 0; }

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

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

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

| expression DIV expression { $$ = $1 / $3; }

;

%%

int is\_declared(char \*name) {

for (int i = 0; i < var\_count; i++) {

if (strcmp(symbol\_table[i].name, name) == 0) {

return 1;

}

}

return 0;

}

char\* get\_var\_type(char \*name) {

for (int i = 0; i < var\_count; i++) {

if (strcmp(symbol\_table[i].name, name) == 0) {

return symbol\_table[i].type;

}

}

return NULL;

}

int check\_type(char \*var\_name, char \*expected\_type) {

if (strcmp(expected\_type, "int") == 0) {

return 1;

}

return 0;

}

void yyerror(const char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

}

int main() {

yyparse();

return 0;

}

**OUTPUT**:

int x;

x = 10;

(no errors)

---------------------------------------------------------------------------------------------------------------------------------

**9. develop a program that optimizes the given input block using Code Optimization techniques**

**Lex Specification**

%{

#include "y.tab.h"

#include <stdio.h>

#include <stdlib.h>

%}

%%

"int" { return INT; }

[a-zA-Z][a-zA-Z0-9]\* { yylval.str = strdup(yytext); return IDENTIFIER; }

[0-9]+ { yylval.num = atoi(yytext); return NUMBER; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MULT; }

"/" { return DIV; }

"=" { return ASSIGN; }

";" { return SEMICOLON; }

[ \t\n] { /\* Ignore whitespaces \*/ }

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

**YACC Specification**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_VARS 100

typedef struct {

char \*name;

int value;

int is\_used;

int is\_constant;

} Variable;

Variable symbol\_table[MAX\_VARS];

int var\_count = 0;

int is\_declared(char \*name);

int find\_var(char \*name);

int optimize\_constant(int val1, int val2, char op);

void yyerror(const char \*s);

%}

%token INT IDENTIFIER NUMBER ASSIGN PLUS MINUS MULT DIV SEMICOLON

%%

program:

declarations statements

;

declarations:

declarations declaration

| /\* empty \*/

;

declaration:

INT IDENTIFIER SEMICOLON {

if (is\_declared($2)) {

printf("Error: Variable '%s' already declared.\n", $2);

} else {

symbol\_table[var\_count].name = strdup($2);

symbol\_table[var\_count].value = 0;

symbol\_table[var\_count].is\_used = 0;

symbol\_table[var\_count].is\_constant = 0;

var\_count++;

}

}

;

statements:

statements statement

| /\* empty \*/

;

statement:

IDENTIFIER ASSIGN expression SEMICOLON {

if (!is\_declared($1)) {

printf("Error: Variable '%s' is not declared.\n", $1);

} else {

int var\_index = find\_var($1);

int val = $3;

if (symbol\_table[var\_index].is\_constant) {

printf("%s = %d\n", symbol\_table[var\_index].name, val);

} else {

symbol\_table[var\_index].value = val;

symbol\_table[var\_index].is\_used = 1;

printf("%s = %d\n", symbol\_table[var\_index].name, val);

}

}

}

| expression SEMICOLON

;

expression:

NUMBER { $$ = $1; }

| IDENTIFIER { $$ = find\_var($1); }

| expression PLUS expression { $$ = optimize\_constant($1, $3, '+'); }

| expression MINUS expression { $$ = optimize\_constant($1, $3, '-'); }

| expression MULT expression { $$ = optimize\_constant($1, $3, '\*'); }

| expression DIV expression { $$ = optimize\_constant($1, $3, '/'); }

;

%%

int is\_declared(char \*name) {

for (int i = 0; i < var\_count; i++) {

if (strcmp(symbol\_table[i].name, name) == 0) {

return 1;

}

}

return 0;

}

int find\_var(char \*name) {

for (int i = 0; i < var\_count; i++) {

if (strcmp(symbol\_table[i].name, name) == 0) {

return i;

}

}

return -1;

}

int optimize\_constant(int val1, int val2, char op) {

if (op == '+') return val1 + val2;

if (op == '-') return val1 - val2;

if (op == '\*') return val1 \* val2;

if (op == '/') return val1 / val2;

return 0;

}

void yyerror(const char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

}

int main() {

printf("Enter a block of code to optimize:\n");

yyparse();

return 0;

}

**OUTPUT**:

Enter a block of code to optimize:

int x;

x = 5 + 3;

x = 8

----------------------------------------------------------------------------------------------------------------------------------

**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>

#define MOV 0x01

#define ADD 0x02

#define SUB 0x03

#define MUL 0x04

#define DIV 0x05

typedef struct {

char var[10];

char op[10];

char arg1[10];

char arg2[10];

} IntermediateCode;

void generate\_assembly(IntermediateCode ic) {

char assembly[50];

unsigned char machine\_code[5];

if (strcmp(ic.op, "+") == 0) {

// Addition: var = arg1 + arg2

printf("Assembly: MOV %s, %s\n", ic.arg1, ic.var);

printf("Assembly: ADD %s, %s\n", ic.var, ic.arg2);

machine\_code[0] = MOV;

machine\_code[1] = atoi(ic.arg1);

machine\_code[2] = atoi(ic.arg2);

printf("Machine Code: %02X %02X %02X\n", machine\_code[0], machine\_code[1], machine\_code[2]);

} else if (strcmp(ic.op, "\*") == 0) {

// Multiplication: var = arg1 \* arg2

printf("Assembly: MOV %s, %s\n", ic.arg1, ic.var);

printf("Assembly: MUL %s, %s\n", ic.var, ic.arg2);

machine\_code[0] = MUL;

machine\_code[1] = atoi(ic.arg1);

machine\_code[2] = atoi(ic.arg2);

printf("Machine Code: %02X %02X %02X\n", machine\_code[0], machine\_code[1], machine\_code[2]);

} else if (strcmp(ic.op, "-") == 0) {

// Subtraction: var = arg1 - arg2

printf("Assembly: MOV %s, %s\n", ic.arg1, ic.var);

printf("Assembly: SUB %s, %s\n", ic.var, ic.arg2);

machine\_code[0] = SUB;

machine\_code[1] = atoi(ic.arg1);

machine\_code[2] = atoi(ic.arg2);

printf("Machine Code: %02X %02X %02X\n", machine\_code[0], machine\_code[1], machine\_code[2]);

} else if (strcmp(ic.op, "/") == 0) {

// Division: var = arg1 / arg2

printf("Assembly: MOV %s, %s\n", ic.arg1, ic.var);

printf("Assembly: DIV %s, %s\n", ic.var, ic.arg2);

machine\_code[0] = DIV;

machine\_code[1] = atoi(ic.arg1);

machine\_code[2] = atoi(ic.arg2);

printf("Machine Code: %02X %02X %02X\n", machine\_code[0], machine\_code[1], machine\_code[2]);

}

}

int main() {

IntermediateCode code[] = {

{"t1", "+", "a", "b"},

{"t2", "\*", "t1", "c"},

{"t3", "-", "t2", "d"},

{"result", "/", "t3", "e"}

};

int num\_instructions = sizeof(code) / sizeof(code[0]);

for (int i = 0; i < num\_instructions; i++) {

generate\_assembly(code[i]);

}

return 0;

}

**OUTPUT**:

Assembly: MOV a, t1

Assembly: ADD t1, b

Machine Code: 01 97 98

Assembly: MOV t1, t2

Assembly: MUL t2, c

Machine Code: 04 98 99

Assembly: MOV t2, t3

Assembly: SUB t3, d

Machine Code: 03 99 100

Assembly: MOV t3, result

Assembly: DIV result, e

Machine Code: 05 100 101

---------------------------------------------------------------------------------------------------------------------------------------------