1.CALCULATOR USING LEX FLEX

**nano calculator.y-**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int flag = 0;

int yylex(void);

void yyerror(const char \*s);

typedef struct Node {

char label[50];

struct Node \*left, \*right;

int val;

} Node;

// Corrected function declaration

Node\* createNode(const char \*label, Node\* left, Node\* right);

void printParseTree(Node\* root, int depth);

void freeTree(Node\* root);

Node\* root = NULL;

%}

%union {

int val;

struct Node\* node;

}

%token <val> NUMBER

%type <node> E

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

%%

ArithmeticExpression:

E {

root = $1;

}

;

E:

E '+' E {

$$ = createNode("+", $1, $3);

$$->val = $1->val + $3->val;

}

| E '-' E {

$$ = createNode("-", $1, $3);

$$->val = $1->val - $3->val;

}

| E '\*' E {

$$ = createNode("\*", $1, $3);

$$->val = $1->val \* $3->val;

}

| E '/' E {

$$ = createNode("/", $1, $3);

$$->val = $1->val / $3->val;

}

| E '%' E {

$$ = createNode("%", $1, $3);

$$->val = $1->val % $3->val;

}

| '(' E ')' {

$$ = createNode("()", $2, NULL);

$$->val = $2->val;

}

| NUMBER {

char buffer[20];

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

$$ = createNode(buffer, NULL, NULL);

$$->val = $1;

}

;

%%

Node\* createNode(const char \*label, Node\* left, Node\* right) {

Node\* newNode = (Node\*) malloc(sizeof(Node));

strcpy(newNode->label, label);

newNode->left = left;

newNode->right = right;

newNode->val = 0;

return newNode;

}

void printParseTree(Node\* root, int depth) {

if (!root) return;

for (int i = 0; i < depth; i++)

printf(" ");

printf("|-- %s\n", root->label);

printParseTree(root->left, depth + 1);

printParseTree(root->right, depth + 1);

}

void freeTree(Node\* root) {

if (!root) return;

freeTree(root->left);

freeTree(root->right);

free(root);

}

void yyerror(const char \*s) {

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

flag = 1;

}

int main() {

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

yyparse();

if (flag == 0 && root != NULL) {

printf("\nResult = %d\n", root->val);

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

printf("\nParse Tree\n===========================================\n");

printParseTree(root, 0);

}

freeTree(root);

return 0;

}

**nano calculator.l-**

%{

#include <stdio.h>

#include <stdlib.h>

#include "calculator.tab.h" // Correctly declares yylval as YYSTYPE

%}

%%

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

[\t ]+ ; // Ignore whitespace

\n return 0; // End input on newline

. return yytext[0]; // Return single characters

%%

int yywrap() {

return 1;

}

**bison -d calculator.y**

**flex calculator.l**

**gcc calculator.tab.c lex.yy.c -o calculator -lfl**

**./calculator**

|  |
| --- |
|  |

2.LEXICAL ANALYZER FOR C-LIKE SYNTAX

**nano c\_lex.l-**

**C\_lex.l-**

%{

// Implement lexical analyser for subset of 'C' language using LEX.

#include <stdio.h>

extern FILE \*yyin;

int lno = 1;

%}

%%

"int"|"return"|"if"|"else"|"for" {

printf("Line %d: | Lexeme '%s' | Token: KEYWORD | Token-Value: %s\n", lno, yytext, yytext);

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

printf("Line %d: | Lexeme '%s' | Token: IDENTIFIER | Token-Value: %s\n", lno, yytext, yytext);

}

[0-9]+\.[0-9]+ {

printf("Line %d: | Lexeme '%s' | Token: FLOAT | Token-Value: %s\n", lno, yytext, yytext);

}

[0-9]+ {

printf("Line %d: | Lexeme '%s' | Token: NUMBER | Token-Value: %s\n", lno, yytext, yytext);

}

"=="|"!="|"<"|">"|"="|"+"|"-"|"\*"|"/" {

printf("Line %d: | Lexeme '%s' | Token: OPERATOR | Token-Value: %s\n", lno, yytext, yytext);

}

"{"|"}"|"("|")"|";" {

printf("Line %d: | Lexeme '%s' | Token: SYMBOL | Token-Value: %s\n", lno, yytext, yytext);

}

[ \t]+ { /\* Skip whitespace \*/ }

\n { lno++; }

. {

printf("Line %d: | Lexeme '%s' | Error: Unrecognized symbol\n", lno, yytext);

}

%%

int yywrap() {

return 1;

}

int main() {

FILE \*f = fopen("input.txt", "r");

if (!f) {

perror("Failed to open c.txt");

return 1;

}

yyin = f;

printf("Reading C code from c.txt...\n\n");

yylex();

fclose(f);

return 0;

}

Also make an input file-

Nano input.txt

**Input.txt:**

#include <stdio.h>

int main() {

printf("Hello World");

return 0;

}

**flex c\_lex.l**

**gcc lex.yy.c -o c\_lex -lfl**

**./c\_lex**

3.SYNTAX ANALYZER FOR ENGLISH

**nano english\_lex.l-**

%{

// Implement lexical analyzer for subset of English language using LEX

#include <stdio.h>

extern FILE \*yyin;

int lno = 1;

%}

%%

[0-9]+ {

printf("Line %d: Lexeme '%s' Token: NUMBER Token-Value: %s\n", lno, yytext, yytext);

}

"I"|"you"|"he"|"she"|"it"|"we"|"they"|"me"|"him"|"her"|"us"|"them"|"my"|"your"|"his"|"its"|"our"|"their"|"mine"|"yours"|"hers"|"ours"|"theirs"|"myself"|"yourself"|"himself"|"herself"|"itself"|"ourselves"|"yourselves"|"themselves" {

printf("Line %d: Lexeme '%s' Token: PRONOUN Token-Value: %s\n", lno, yytext, yytext);

}

[A-Z][a-z]+ {

printf("Line %d: Lexeme '%s' Token: NOUN Token-Value: %s\n", lno, yytext, yytext);

}

[a-z]+("able"|"ible"|"al"|"ant"|"ent"|"ary"|"ory"|"ed"|"en"|"ful"|"ic"|"ical"|"ish"|"ive"|"less"|"like"|"ly"|"ous"|"ious"|"eous"|"y"|"some"|"ing"|"ese"|"esque"|"ian"|"ar"|"ine"|"istic"|"worthy")$ {

printf("Line %d: Lexeme '%s' Token: ADJECTIVE Token-Value: %s\n", lno, yytext, yytext);

}

[a-z]+("ly") {

printf("Line %d: Lexeme '%s' Token: ADVERB Token-Value: %s\n", lno, yytext, yytext);

}

[a-z]+("ed"|"ing") {

printf("Line %d: Lexeme '%s' Token: VERB Token-Value: %s\n", lno, yytext, yytext);

}

[a-z]+ {

printf("Line %d: Lexeme '%s' Token: WORD Token-Value: %s\n", lno, yytext, yytext);

}

[ \t]+ {

/\* skip whitespace \*/

}

\n {

lno++;

}

['"“”] {

printf("Line %d: Lexeme '%s' Token: QUOTATION Token-Value: %s\n", lno, yytext, yytext);

}

[.,!?;:] {

printf("Line %d: Lexeme '%s' Token: PUNCTUATION Token-Value: %s\n", lno, yytext, yytext);

}

. {

printf("Line %d: Lexeme '%s' Error: UNRECOGNIZED SYMBOL\n", lno, yytext);

}

%%

int yywrap() {

return 1;

}

int main() {

FILE \*f = fopen("english.txt", "r");

if (!f) {

perror("Failed to open english.txt");

return 1;

}

yyin = f;

printf("Reading English text from english.txt...\n\n");

yylex();

fclose(f);

return 0;

}

**nano engsyntax.l-**

%{

#include "engsyntax.tab.h"

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include <stdio.h>

#ifndef strdup

char \*strdup(const char \*s) {

char \*d = malloc(strlen(s) + 1);

if (d == NULL) return NULL;

strcpy(d, s);

return d;

}

#endif

// Convert to lowercase for case-insensitive match

char \*to\_lower(const char \*s) {

char \*lower = strdup(s);

if (lower == NULL) return NULL; // <-- Avoid NULL dereference

for (int i = 0; lower[i]; i++) {

lower[i] = tolower((unsigned char)lower[i]);

}

return lower;

}

%}

%%

[aA]|[tT][hH][eE] {

char \*word = to\_lower(yytext);

yylval.str = word;

return ARTICLE;

}

[bB][lL][aA][cC][kK]|[wW][hH][iI][tT][eE]|[hH][aA][pP][pP][yY]|[aA][nN][gG][rR][yY] {

yylval.str = to\_lower(yytext);

return ADJECTIVE;

}

[cC][aA][tT][sS]?|[dD][oO][gG][sS]?|[fF][iI][sS][hH]|[fF][oO][oO][dD][sS]? {

yylval.str = to\_lower(yytext);

return NOUN;

}

[eE][aA][tT][sS]?|[lL][oO][vV][eE][sS]?|[hH][aA][tT][eE][sS]? {

yylval.str = to\_lower(yytext);

return VERB;

}

[ \t\r\n]+ { /\* skip whitespace \*/ }

[.?!] { /\* ignore punctuation \*/ }

[a-zA-Z]+ {

char \*word = to\_lower(yytext);

// printf("Unknown word: %s\n", word);

free(word);

return 0;

}

. { return yytext[0]; }

.[!@#$%^&\*|] {

printf("unidentified token\n");

exit(0);

}

%%

int yywrap() { return 1; }

**nano engsyntax.y-**

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

int yylex();

extern FILE \*yyin; // Declare yyin for file input

%}

%union {

char\* str;

}

%token <str> ARTICLE NOUN VERB ADJECTIVE

%type <str> sentence opt\_adj

%%

sentence:

ARTICLE opt\_adj NOUN VERB NOUN {

printf("Valid sentence structure:\n");

printf("Article: %s\n", $1);

if ($2) printf("Adjective: %s\n", $2);

printf("Subject: %s\n", $3);

printf("Verb: %s\n", $4);

printf("Object: %s\n", $5);

// Free allocated memory

free($1); free($2); free($3); free($4); free($5);

}

;

opt\_adj:

/\* empty \*/ { $$ = NULL; }

| ADJECTIVE { $$ = $1; }

;

%%

void yyerror(const char \*s) {

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

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <input\_file>\n", argv[0]);

return 1;

}

yyin = fopen(argv[1], "r");

if (!yyin) {

perror("Error opening file");

return 1;

}

yyparse();

fclose(yyin);

return 0;

}

**nano english.txt-**

The cat eats fish.

**bison -d engsyntax.y**

**flex english\_lex.l**

**flex engsyntax.l**

**gcc engsyntax.tab.c lex.yy.c -o engparser -lfl**

**./engparser**

4. Parser for if/else constructs

**Nano ifelse\_syntax.l-**

%{

#include "ifelse\_syntax.tab.h"

#include <string.h>

#include <stdlib.h>

#ifndef strdup

char \*strdup(const char \*s) {

char \*d = malloc(strlen(s) + 1); // allocate memory

if (d == NULL) return NULL; // check for malloc failure

strcpy(d, s); // copy string

return d;

}

#endif

%}

%%

"if" { return IF; }

"else" { return ELSE; }

"(" { return LPAREN; }

")" { return RPAREN; }

"{" { return LBRACE; }

"}" { return RBRACE; }

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

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

"=" { return ASSIGN; }

"==" { return EQ; }

"<" { return LT; }

">" { return GT; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MUL; }

"/" { return DIV; }

";" { return SEMICOLON; }

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

. { return yytext[0]; }

%%

int yywrap() { return 1; }

**Nano ifelse\_syntax.y-**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void yyerror(const char \*s);

int yylex();

void print\_indent(int level);

extern FILE \*yyin;

int indent = 0; // Global indent level

%}

%union {

char\* str; // for ID and others

}

%token <str> ID NUMBER

/\* Declare operator precedence \*/

%left PLUS MINUS

%left MUL DIV

%left LT GT EQ

%token IF ELSE LPAREN RPAREN LBRACE RBRACE EQ LT GT PLUS MINUS MUL DIV SEMICOLON ASSIGN

%%

program:

if\_else\_stmt {

indent++;

/\* if\_else\_stmt printing is handled by its own action \*/

indent--;

}

;

if\_else\_stmt:

IF LPAREN condition RPAREN LBRACE stmt\_list RBRACE ELSE LBRACE stmt\_list RBRACE {

print\_indent(indent);

printf("|-- if\_else\_stmt\n");

indent++;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* condition rule prints its structure \*/

indent--;

print\_indent(indent);

printf("|-- stmt\_list (if-block)\n");

indent++;

/\* if stmt\_list rule prints its structure \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

printf("|-- stmt\_list (else-block)\n");

indent++;

/\* else stmt\_list rule prints its structure \*/

indent--;

indent--;

}

;

condition:

expr LT expr {

print\_indent(indent);

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

indent++;

print\_indent(indent);

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

indent++;

/\* Left expr printed by expr rule \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* Right expr printed by expr rule \*/

indent--;

indent--;

}

| expr GT expr {

print\_indent(indent);

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

indent++;

print\_indent(indent);

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

indent++;

/\* Left expr printed by expr rule \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* Right expr printed by expr rule \*/

indent--;

indent--;

}

| expr EQ expr {

print\_indent(indent);

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

indent++;

print\_indent(indent);

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

indent++;

/\* Left expr printed by expr rule \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* Right expr printed by expr rule \*/

indent--;

indent--;

}

;

stmt\_list:

stmt {

print\_indent(indent);

printf("|-- stmt\_list\n");

indent++;

print\_indent(indent);

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

indent--;

}

| stmt\_list stmt {

print\_indent(indent);

printf("|-- stmt\_list\n");

indent++;

print\_indent(indent);

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

indent--;

}

;

stmt:

ID ASSIGN expr SEMICOLON {

print\_indent(indent);

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

indent++;

print\_indent(indent);

printf("|-- ID: %s\n", $1);

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* expr already prints its contents \*/

indent--;

print\_indent(indent);

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

indent--;

}

;

expr:

expr PLUS expr {

print\_indent(indent);

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

indent++;

print\_indent(indent);

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

indent++;

/\* Left expr printed by expr rule \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* Right expr printed by expr rule \*/

indent--;

indent--;

}

| expr MINUS expr {

print\_indent(indent);

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

indent++;

print\_indent(indent);

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

indent++;

/\* Left expr printed by expr rule \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* Right expr printed by expr rule \*/

indent--;

indent--;

}

| expr MUL expr {

print\_indent(indent);

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

indent++;

print\_indent(indent);

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

indent++;

/\* Left expr printed by expr rule \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* Right expr printed by expr rule \*/

indent--;

indent--;

}

| expr DIV expr {

print\_indent(indent);

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

indent++;

print\_indent(indent);

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

indent++;

/\* Left expr printed by expr rule \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* Right expr printed by expr rule \*/

indent--;

indent--;

}

| ID {

print\_indent(indent);

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

indent++;

print\_indent(indent);

printf("|-- ID: %s\n", $1);

indent--;

}

| NUMBER {

print\_indent(indent);

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

indent++;

print\_indent(indent);

printf("|-- NUMBER: %s\n", $1);

indent--;

}

;

%%

void yyerror(const char \*s) {

printf("Error: %s\n", s);

}

void print\_indent(int level) {

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

printf(" ");

}

}

int main(int argc, char \*argv[]) {

if (argc < 2) {

printf("Usage: %s <input\_file>\n", argv[0]);

return 1;

}

FILE \*input\_file = fopen(argv[1], "r");

if (!input\_file) {

perror("Error opening input file");

return 1;

}

yyin = input\_file;

yyparse();

fclose(input\_file);

return 0;

}

**Nano input\_ifelse.txt-**

if (x < 10)

x = x + 1;

else

x = 0;

**bison -d ifelse\_syntax.y**

**flex ifelse\_syntax.l**

**gcc ifelse\_syntax.tab.c lex.yy.c -o ifelse -lfl**

**./ifelse input\_ifelse.txt**

5. Lexical analyzer for Java tokens

**nano java\_lex.l-**

%{

// Lexical Analyzer for a subset of Java using Lex

#include <stdio.h>

extern FILE \*yyin;

int lno = 1;

%}

%%

"//".\* {

printf("Line %d: Lexeme '%s' Token: SINGLE\_LINE\_COMMENT Token-Value: %s\n", lno, yytext, yytext);

}

\/\\*([^\*]|\\*+[^\*/])\*\\*+\/ {

printf("Line %d: Lexeme '%s' Token: MULTI\_LINE\_COMMENT Token-Value: %s\n", lno, yytext, yytext);

}

"abstract"|"assert"|"boolean"|"break"|"byte"|"case"|"catch"|"char"|"class"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extends"|"final"|"finally"|"float"|"for"|"goto"|"if"|"implements"|"import"|"instanceof"|"int"|"interface"|"long"|"native"|"new"|"package"|"private"|"protected"|"public"|"return"|"short"|"static"|"strictfp"|"super"|"switch"|"synchronized"|"this"|"throw"|"throws"|"transient"|"try"|"void"|"volatile"|"while" {

printf("Line %d: Lexeme '%s' Token: KEYWORD Token-Value: %s\n", lno, yytext, yytext);

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

printf("Line %d: Lexeme '%s' Token: IDENTIFIER Token-Value: %s\n", lno, yytext, yytext);

}

[0-9]+\.[0-9]\*([eE][-+]?[0-9]+)?[fFdD]? {

printf("Line %d: Lexeme '%s' Token: FLOAT\_LITERAL Token-Value: %s\n", lno, yytext, yytext);

}

[0-9]+ {

printf("Line %d: Lexeme '%s' Token: INT\_LITERAL Token-Value: %s\n", lno, yytext, yytext);

}

\'([^\\\']|\\.)\' {

printf("Line %d: Lexeme '%s' Token: CHAR\_LITERAL Token-Value: %s\n", lno, yytext, yytext);

}

\"([^\\\"]|\\.)\*\" {

printf("Line %d: Lexeme '%s' Token: STRING\_LITERAL Token-Value: %s\n", lno, yytext, yytext);

}

"="|"+"|"-"|"\*"|"/"|"%"|"++"|"--"|"=="|"!="|"<"|"<="|">"|">="|"&&"|"||"|"!"|"&"|"|"|"^"|"~"|"<<"|">>"|">>>" {

printf("Line %d: Lexeme '%s' Token: OPERATOR Token-Value: %s\n", lno, yytext, yytext);

}

"("|")"|"{"|"}"|"["|"]"|";"|","|"." {

printf("Line %d: Lexeme '%s' Token: SEPARATOR Token-Value: %s\n", lno, yytext, yytext);

}

[ \t]+ { /\* skip whitespace \*/ }

\n { lno++; }

. {

printf("Line %d: Lexeme '%s' Error: Unrecognized symbol\n", lno, yytext);

}

%%

int yywrap() {

return 1;

}

int main() {

FILE \*f = fopen("java.txt", "r");

if (!f) {

perror("Failed to open java\_input.txt");

return 1;

}

yyin = f;

printf("Reading Java code from java\_input.txt...\n\n");

yylex();

fclose(f);

return 0;

}

**nano java.txt-**

public class Main {

public static void main(String[] args) {

int number = 10;

float rate = 12.5;

char letter = 'A';

String message = "Hello, World!";

if (number < 20) {

number++;

} else {

number = 0;

}

System.out.println(message);

}

}

**flex java\_lex.l**

**gcc lex.yy.c -o java\_lex -lfl**

**./java\_lex**

6. Full SQL scanner + parser

**sql\_lex.l-**

%{

// Lexical Analyzer for a subset of SQL using Lex

#include <stdio.h>

extern FILE \*yyin;

int lno = 1;

%}

%%

"--"[^\n]\* {

printf("Line %d: Lexeme '%s' Token: SINGLE\_LINE\_COMMENT\n", lno, yytext);

}

\/\\*([^\*]|\\*+[^\*/])\*\\*+\/ {

printf("Line %d: Lexeme '%s' Token: MULTI\_LINE\_COMMENT\n", lno, yytext);

}

"SELECT"|"FROM"|"WHERE"|"INSERT"|"INTO"|"VALUES"|"UPDATE"|"SET"|"DELETE"|"CREATE"|"TABLE"|"DROP"|"ALTER"|"ADD"|"JOIN"|"ON"|"AS"|"GROUP"|"BY"|"ORDER"|"HAVING"|"DISTINCT"|"AND"|"OR"|"NOT"|"IN"|"IS"|"NULL"|"LIKE"|"BETWEEN"|"EXISTS"|"UNION"|"ALL"|"CASE"|"WHEN"|"THEN"|"ELSE"|"END"|"LIMIT"|"OFFSET" {

printf("Line %d: Lexeme '%s' Token: KEYWORD Token-Value: %s\n", lno, yytext, yytext);

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

printf("Line %d: Lexeme '%s' Token: IDENTIFIER Token-Value: %s\n", lno, yytext, yytext);

}

[0-9]+\.[0-9]+ {

printf("Line %d: Lexeme '%s' Token: FLOAT\_LITERAL Token-Value: %s\n", lno, yytext, yytext);

}

[0-9]+ {

printf("Line %d: Lexeme '%s' Token: INT\_LITERAL Token-Value: %s\n", lno, yytext, yytext);

}

\'([^\\\']|\\.)\*\' {

printf("Line %d: Lexeme '%s' Token: STRING\_LITERAL Token-Value: %s\n", lno, yytext, yytext);

}

"="|"!="|"<>"|"<="|">="|"<"|">"|"+"|"-"|"\*"|"/"|"%" {

printf("Line %d: Lexeme '%s' Token: OPERATOR Token-Value: %s\n", lno, yytext, yytext);

}

"("|")"|","|";"|"\." {

printf("Line %d: Lexeme '%s' Token: SYMBOL Token-Value: %s\n", lno, yytext, yytext);

}

[ \t]+ { /\* skip whitespace \*/ }

\n { lno++; }

. {

printf("Line %d: Lexeme '%s' Error: Unrecognized symbol\n", lno, yytext);

}

%%

int yywrap() {

return 1;

}

int main() {

FILE \*f = fopen("sql.txt", "r");

if (!f) {

perror("Failed to open input.txt");

return 1;

}

yyin = f;

printf("Reading SQL code from input.txt...\n\n");

yylex();

fclose(f);

return 0;

}

**sql\_syntax.l-**

%{

#include "sql\_syntax.tab.h"

#include <string.h>

#include <stdlib.h>

#ifndef strdup

char \*strdup(const char \*s) {

char \*d = malloc(strlen(s) + 1);

if (d == NULL) return NULL;

strcpy(d, s);

return d;

}

#endif

%}

%%

"SELECT" { return SELECT; }

"FROM" { return FROM; }

"," { return COMMA; }

";" { return SEMICOLON; }

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

[ \t\r\n]+ { /\* skip whitespace \*/ }

. { return yytext[0]; }

%%

int yywrap() { return 1; }

**sql\_syntax.y-**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void yyerror(const char \*s);

int yylex();

void print\_indent(int level);

extern FILE \*yyin;

int indent = 0;

%}

%union {

char\* str;

}

%token <str> IDENTIFIER

%token SELECT FROM COMMA SEMICOLON

%%

query:

SELECT select\_list FROM IDENTIFIER SEMICOLON {

print\_indent(indent);

printf("|-- SELECT statement\n");

indent++;

print\_indent(indent);

printf("|-- Columns:\n");

indent++;

// Columns printed in select\_list rule

indent--;

print\_indent(indent);

printf("|-- Table: %s\n", $4);

indent--;

}

;

select\_list:

IDENTIFIER {

print\_indent(indent);

printf("|-- Column: %s\n", $1);

}

| select\_list COMMA IDENTIFIER {

print\_indent(indent);

printf("|-- Column: %s\n", $3);

}

;

%%

void yyerror(const char \*s) {

printf("Syntax Error: %s\n", s);

}

void print\_indent(int level) {

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

printf(" ");

}

}

int main(int argc, char \*argv[]) {

if (argc < 2) {

printf("Usage: %s <input\_file>\n", argv[0]);

return 1;

}

FILE \*input\_file = fopen(argv[1], "r");

if (!input\_file) {

perror("Error opening input file");

return 1;

}

yyin = input\_file;

yyparse();

fclose(input\_file);

return 0;

}

**sql.txt-**

SELECT name, age FROM users WHERE age >= 18;

**bison -d sql\_syntax.y**

**flex sql\_lex.l**

**flex sql\_syntax.l**

**gcc sql\_syntax.tab.c lex.yy.c -o sqlparser -lfl**

**./sqlparser sql.txt**

7. Lexical analyzer for Python‐style tokens

**python\_lex.l-**

%{

// Lexical Analyzer for a subset of Python using Lex

#include <stdio.h>

extern FILE \*yyin;

int lno = 1;

%}

%%

"def"|"return"|"if"|"elif"|"else"|"for"|"while"|"in"|"and"|"or"|"not"|"import"|"from"|"as"|"class"|"pass"|"break"|"continue"|"True"|"False"|"None" {

printf("Line %d: Lexeme '%s' Token: KEYWORD Token-Value: %s\n", lno, yytext, yytext);

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

printf("Line %d: Lexeme '%s' Token: IDENTIFIER Token-Value: %s\n", lno, yytext, yytext);

}

[0-9]+\.[0-9]+ {

printf("Line %d: Lexeme '%s' Token: FLOAT Token-Value: %s\n", lno, yytext, yytext);

}

[0-9]+ {

printf("Line %d: Lexeme '%s' Token: INTEGER Token-Value: %s\n", lno, yytext, yytext);

}

\"([^\\\"]|\\.)\*\" {

printf("Line %d: Lexeme '%s' Token: STRING Token-Value: %s\n", lno, yytext, yytext);

}

\'([^\\\']|\\.)\*\' {

printf("Line %d: Lexeme '%s' Token: STRING Token-Value: %s\n", lno, yytext, yytext);

}

"=="|"!="|"<="|">="|"<"|">"|"="|"+"|"-"|"\*"|"/"|"%"|"//"|"\*\*" {

printf("Line %d: Lexeme '%s' Token: OPERATOR Token-Value: %s\n", lno, yytext, yytext);

}

"("|")"|"["|"]"|"{"|"}"|":"|","|"."|";"|"@"|"->"|"//="|"%="|"\*\*="|"+="|"-="|"\*="|"/="|"=" {

printf("Line %d: Lexeme '%s' Token: SYMBOL Token-Value: %s\n", lno, yytext, yytext);

}

[ \t]+ { /\* skip spaces and tabs \*/ }

\n { lno++; }

"#".\* { /\* skip comment \*/ }

. {

printf("Line %d: Lexeme '%s' Error: Unrecognized symbol\n", lno, yytext);

}

%%

int yywrap() {

return 1;

}

int main() {

FILE \*f = fopen("py.txt", "r");

if (!f) {

perror("Failed to open input.txt");

return 1;

}

yyin = f;

printf("Reading Python code from input.txt...\n\n");

yylex();

fclose(f);

return 0;

}

**flex python\_lex.l**

**gcc lex.yy.c -o python\_lex -lfl**

**./python\_lex**

8. Parser for while‐loops.

**nano whilesyntax.l –**

%{

#include "whilesyntax.tab.h"

#include <string.h>

#include <stdlib.h>

extern FILE \*yyin; // Declare external file pointer

#ifndef strdup

char \*strdup(const char \*s) {

char \*d = malloc(strlen(s) + 1);

if (d == NULL) return NULL;

strcpy(d, s);

return d;

}

#endif

%}

%%

"while" { return WHILE; }

"(" { return LPAREN; }

")" { return RPAREN; }

"{" { return LBRACE; }

"}" { return RBRACE; }

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

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

"=" { return ASSIGN; }

"==" { return EQ; }

"<" { return LT; }

">" { return GT; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MUL; }

"/" { return DIV; }

";" { return SEMICOLON; }

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

. { return yytext[0]; }

%%

int yywrap() { return 1; }

**nano whilesyntax.y –**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void yyerror(const char \*s);

int yylex();

void print\_indent(int level);

int indent = 0; // Global indent level

extern FILE \*yyin; // Declare external file pointer

%}

%union {

char\* str; // for ID and others

}

%token <str> ID NUMBER

/\* Declare operator precedence \*/

%left PLUS MINUS

%left MUL DIV

%left LT GT EQ

%token WHILE LPAREN RPAREN LBRACE RBRACE EQ LT GT PLUS MINUS MUL DIV SEMICOLON ASSIGN

%%

program:

while\_stmt {

//printf("\nParse Tree:\n");

//printf("|-- program\n");

indent++;

/\* while\_stmt printing is handled by its own action \*/

indent--;

}

;

while\_stmt:

WHILE LPAREN condition RPAREN LBRACE stmt\_list RBRACE {

print\_indent(indent);

printf("|-- while\_stmt\n");

indent++;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* condition rule prints its structure \*/

indent--;

print\_indent(indent);

printf("|-- stmt\_list\n");

indent++;

/\* stmt\_list rule prints its structure \*/

indent--;

print\_indent(indent);

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

indent--;

}

;

condition:

expr LT expr {

print\_indent(indent);

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

indent++;

/\* Left expr printed by expr rule \*/

indent--;

print\_indent(indent);

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

print\_indent(indent);

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

indent++;

/\* Right expr printed by expr rule \*/

indent--;

}

;

/\* Use a standard left-recursive definition (without an empty alternative)

to avoid reduce/reduce conflicts.

In the base case, we simulate an extra branch showing an empty additional statement.

\*/

stmt\_list:

stmt {

print\_indent(indent);

printf("|-- stmt\_list\n");

indent++;

/\* Print the stmt details (already printed in stmt rule) \*/

print\_indent(indent);

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

print\_indent(indent+1);

printf("|-- (empty, if any additional statement)\n");

indent--;

}

| stmt\_list stmt {

print\_indent(indent);

printf("|-- stmt\_list\n");

indent++;

print\_indent(indent);

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

indent--;

}

;

expr:

expr PLUS expr {

print\_indent(indent);

printf("|-- expr + expr\n");

}

| ID {

print\_indent(indent);

printf("|-- ID: %s\n", $1);

}

| NUMBER {

print\_indent(indent);

printf("|-- NUMBER: %s\n", $1);

}

;

stmt:

ID ASSIGN expr SEMICOLON {

print\_indent(indent);

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

indent++;

print\_indent(indent);

printf("|-- ID: %s\n", $1);

print\_indent(indent);

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

/\* expr already prints its contents \*/

print\_indent(indent);

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

indent--;

}

%%

void yyerror(const char \*s) {

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

}

void print\_indent(int level) {

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

printf(" ");

}

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <input\_file>\n", argv[0]);

return 1;

}

yyin = fopen(argv[1], "r");

if (!yyin) {

perror("Error opening file");

return 1;

}

printf("Parsing while statement from file:\n");

yyparse();

fclose(yyin);

return 0;

}

**Nano while.txt-**

while (x < 10) {

x = x + 1;

}

**bison -d whilesyntax.y**

**flex whilesyntax.l**

**gcc whilesyntax.tab.c lex.yy.c -o whileparser -lfl**

**./whileparser while.txt**

9. Word‐counting lexer.

**nano word\_count.l –**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_WORDS 1000

#define MAX\_LINES 100

typedef struct {

char word[100];

int count;

int lines[MAX\_LINES];

int lineCount;

} WordEntry;

WordEntry words[MAX\_WORDS];

int word\_count = 0;

int line\_no = 1;

void toLower(char \*s) {

for (int i = 0; s[i]; i++) {

s[i] = tolower(s[i]);

}

}

void addWord(char \*text) {

toLower(text);

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

if (strcmp(words[i].word, text) == 0) {

words[i].count++;

if (words[i].lines[words[i].lineCount - 1] != line\_no) {

words[i].lines[words[i].lineCount++] = line\_no;

}

return;

}

}

strcpy(words[word\_count].word, text);

words[word\_count].count = 1;

words[word\_count].lines[0] = line\_no;

words[word\_count].lineCount = 1;

word\_count++;

}

%}

%%

[a-zA-Z]+ { addWord(yytext); }

\n { line\_no++; }

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

. { /\* ignore punctuation \*/ }

%%

int yywrap() {

return 1;

}

int main(int argc, char \*\*argv) {

FILE \*file;

if (argc < 2) {

printf("Usage: %s <input\_file>\n", argv[0]);

return 1;

}

file = fopen(argv[1], "r");

if (!file) {

perror("Could not open the file");

return 1;

}

yyin = file;

yylex();

fclose(file);

printf("\nWord\t\tCount\tLine Numbers\n");

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

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

printf("%-15s\t%-5d\t", words[i].word, words[i].count);

for (int j = 0; j < words[i].lineCount; j++) {

printf("%d ", words[i].lines[j]);

}

printf("\n");

}

return 0;

}

**nano some\_text.txt-**

Hello world

This is a test.

Hello again, world.

Testing word count, word count again.

**flex word\_count.l**

**gcc lex.yy.c -o word\_count -lfl**

**./word\_count some\_text.txt**

10. C program for constant folding optimization.

**nano constant\_folding.c –**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

typedef struct {

char op[4], arg1[10], arg2[10], result[10];

} Quad;

Quad quads[100];

int quad\_count = 0;

// Store values for temporary variables

typedef struct {

char name[10];

int value;

int is\_const;

} TempVal;

TempVal temp\_vals[100];

int temp\_count = 0;

int isNumber(char \*s) {

for (int i = 0; s[i]; i++)

if (!isdigit(s[i])) return 0;

return 1;

}

int getValue(char \*s, int \*val) {

if (isNumber(s)) {

\*val = atoi(s);

return 1;

}

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

if (strcmp(temp\_vals[i].name, s) == 0 && temp\_vals[i].is\_const) {

\*val = temp\_vals[i].value;

return 1;

}

}

return 0;

}

void setTempVal(char \*name, int value, int is\_const) {

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

if (strcmp(temp\_vals[i].name, name) == 0) {

temp\_vals[i].value = value;

temp\_vals[i].is\_const = is\_const;

return;

}

}

strcpy(temp\_vals[temp\_count].name, name);

temp\_vals[temp\_count].value = value;

temp\_vals[temp\_count].is\_const = is\_const;

temp\_count++;

}

int evaluate(char \*op, int v1, int v2) {

if (strcmp(op, "+") == 0) return v1 + v2;

if (strcmp(op, "-") == 0) return v1 - v2;

if (strcmp(op, "\*") == 0) return v1 \* v2;

if (strcmp(op, "/") == 0) return v1 / v2;

return 0;

}

void foldConstants() {

printf("Performing Constant Folding...\n");

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

int v1, v2;

if (getValue(quads[i].arg1, &v1) && getValue(quads[i].arg2, &v2)) {

int folded = evaluate(quads[i].op, v1, v2);

printf("Folding: %s %s %s => %d\n", quads[i].op, quads[i].arg1, quads[i].arg2, folded);

sprintf(quads[i].op, "=");

sprintf(quads[i].arg1, "%d", folded);

quads[i].arg2[0] = '\0';

setTempVal(quads[i].result, folded, 1);

} else {

setTempVal(quads[i].result, 0, 0);

}

}

}

void printQuads() {

printf("Nr | Op | Arg1 | Arg2 | Result\n");

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

printf("%03d| %-3s | %-4s | %-4s | %-6s\n", i + 1, quads[i].op, quads[i].arg1, quads[i].arg2, quads[i].result);

}

}

int main(int argc, char \*\*argv) {

if (argc < 2) {

printf("Usage: %s <quad\_file>\n", argv[0]);

return 1;

}

FILE \*fp = fopen(argv[1], "r");

if (!fp) {

perror("File open error");

return 1;

}

while (fscanf(fp, "%s %s %s %s", quads[quad\_count].op, quads[quad\_count].arg1,

quads[quad\_count].arg2, quads[quad\_count].result) == 4) {

quad\_count++;

}

fclose(fp);

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

printQuads();

foldConstants();

printf("\n--- After Constant Folding ---\n");

printQuads();

return 0;

}

**nano quad.txt -**

+ 2 3 t1

\* 4 5 t2

+ t1 t2 t3

/ 10 2 t4

**gcc constant\_folding.c -o const\_fold**

**./const\_fold quad.txt**

11. C program for constant propagation.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <ctype.h>

#define MAX\_QUADS 500

#define MAX\_STR 100

typedef struct {

char op[20];

char arg1[MAX\_STR];

char arg2[MAX\_STR];

char result[MAX\_STR];

bool eliminated;

} Quadruple;

Quadruple quad\_list[MAX\_QUADS];

int quad\_count = 0;

bool isConstant(const char \*s) {

if(s == NULL || \*s == '\0' || strcmp(s, "") == 0) return false;

int i = 0;

if(s[0]=='-' || s[0]=='+') { if (s[1] == '\0') return false; i++; }

for(; s[i] != '\0'; i++) if(!isdigit(s[i])) return false;

return i > 0 && s[i-1] != '-' && s[i-1] != '+';

}

int evaluateExpression(const char \*op, int c1, int c2) {

if(strcmp(op, "+") == 0) return c1 + c2;

if(strcmp(op, "-") == 0) return c1 - c2;

if(strcmp(op, "\*") == 0) return c1 \* c2;

if(strcmp(op, "/") == 0) {

if (c2 == 0) {

fprintf(stderr, "Warning: Division by zero\n");

return 0;

}

return c1 / c2;

}

return 0;

}

bool readQuadsFromFile(const char\* filename) {

FILE\* fp = fopen(filename, "r");

if (!fp) {

perror("Error opening Quad input file");

return false;

}

printf("Reading Quads from %s...\n", filename);

quad\_count = 0;

char line[100];

while (fgets(line, sizeof(line), fp) && quad\_count < MAX\_QUADS) {

line[strcspn(line, "\r\n")] = 0;

if (strlen(line) == 0) continue;

// Initialize all fields

quad\_list[quad\_count].op[0] = '\0';

quad\_list[quad\_count].arg1[0] = '\0';

quad\_list[quad\_count].arg2[0] = '\0';

quad\_list[quad\_count].result[0] = '\0';

char op[20], arg1[MAX\_STR], arg2[MAX\_STR], result[MAX\_STR];

int num\_items = sscanf(line, "%19s %99s %99s %99s", op, arg1, arg2, result);

strcpy(quad\_list[quad\_count].op, op);

// Handle assignment operations (= value var)

if (strcmp(op, "=") == 0) {

if (num\_items == 3) {

strcpy(quad\_list[quad\_count].arg1, arg1); // The constant value

strcpy(quad\_list[quad\_count].result, arg2); // The variable

} else {

fprintf(stderr, "Invalid assignment format: %s\n", line);

continue;

}

}

// Handle binary operations (op arg1 arg2 result)

else if (num\_items == 4) {

strcpy(quad\_list[quad\_count].arg1, arg1);

strcpy(quad\_list[quad\_count].arg2, arg2);

strcpy(quad\_list[quad\_count].result, result);

}

else {

fprintf(stderr, "Invalid quad format: %s\n", line);

continue;

}

quad\_list[quad\_count].eliminated = false;

quad\_count++;

}

fclose(fp);

printf("Read %d quads.\n", quad\_count);

return true;

}

void printQuads(const char\* title) {

printf("\n--- %s ---\n", title);

printf("Nr | %-15s | %-25s | %-25s | %-25s\n", "Op", "Arg1", "Arg2", "Result");

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

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

if (!quad\_list[i].eliminated) {

printf("%03d| %-15s | %-25s | %-25s | %-25s\n",

i+1, quad\_list[i].op,

quad\_list[i].arg1[0] ? quad\_list[i].arg1 : "-",

quad\_list[i].arg2[0] ? quad\_list[i].arg2 : "-",

quad\_list[i].result[0] ? quad\_list[i].result : "-");

}

}

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

}

void constantPropagationOptimization() {

printf("\nPerforming Constant Propagation (with Folding)...\n");

bool changed = true;

int pass = 1;

while (changed && pass <= 5) {

changed = false;

printf("\nPass %d:\n", pass++);

// First do constant propagation

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

if (quad\_list[i].eliminated) continue;

// Look for constant assignments (= value var)

if (strcmp(quad\_list[i].op, "=") == 0 && isConstant(quad\_list[i].arg1)) {

const char\* const\_val = quad\_list[i].arg1;

const char\* const\_var = quad\_list[i].result;

// Propagate to subsequent quads

for (int j = i + 1; j < quad\_count; j++) {

if (quad\_list[j].eliminated) continue;

bool propagated = false;

// Check arg1

if (strcmp(quad\_list[j].arg1, const\_var) == 0) {

strcpy(quad\_list[j].arg1, const\_val);

propagated = true;

}

// Check arg2

if (strcmp(quad\_list[j].arg2, const\_var) == 0) {

strcpy(quad\_list[j].arg2, const\_val);

propagated = true;

}

if (propagated) {

printf(" Propagated %s=%s to Quad %d\n", const\_var, const\_val, j+1);

changed = true;

}

}

}

}

// Then do constant folding

char buffer[MAX\_STR];

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

if (quad\_list[i].eliminated) continue;

if ((strcmp(quad\_list[i].op, "+") == 0 || strcmp(quad\_list[i].op, "-") == 0 ||

strcmp(quad\_list[i].op, "\*") == 0 || strcmp(quad\_list[i].op, "/") == 0) &&

isConstant(quad\_list[i].arg1) && isConstant(quad\_list[i].arg2)) {

int c1 = atoi(quad\_list[i].arg1);

int c2 = atoi(quad\_list[i].arg2);

int result = evaluateExpression(quad\_list[i].op, c1, c2);

sprintf(buffer, "%d", result);

printf(" Folded Quad %d: %s %s %s -> %s = %s\n",

i+1, quad\_list[i].op, quad\_list[i].arg1,

quad\_list[i].arg2, quad\_list[i].result, buffer);

// Convert to assignment

strcpy(quad\_list[i].op, "=");

strcpy(quad\_list[i].arg1, buffer);

quad\_list[i].arg2[0] = '\0';

changed = true;

}

}

}

printf("\nConstant Propagation complete.\n");

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

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

printf("Usage: %s <quad\_file>\n", argv[0]);

return 1;

}

if (!readQuadsFromFile(argv[1])) {

fprintf(stderr, "Failed to read input file\n");

return 1;

}

printQuads("Initial Quads");

constantPropagationOptimization();

printQuads("After Constant Propagation");

return 0;

}

**nano quad.txt -**

+ 2 3 t1

\* 4 5 t2

+ t1 t2 t3

/ 10 2 t4

**gcc constant\_propogation.c -o const\_prop**

**./const\_prop quad.txt**

12. Common Subexpression Elimination in C.

**nano cse.c-**

#include<stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

// Structure to represent a quadruple (op, arg1, arg2, result)

typedef struct {

char op[5]; // Operator: +, -, \*, /, etc.

char arg1[10]; // First argument

char arg2[10]; // Second argument (can be empty for unary operations)

char result[10]; // Result

} Quadruple;

// Structure to represent an expression

typedef struct {

char op[5];

char arg1[10];

char arg2[10];

char temp[10]; // Temporary variable used to store this expression

bool used; // Flag to mark if this expression has been used

} Expression;

// Function to check if two expressions are the same

bool isSameExpression(Expression \*expr1, Expression \*expr2) {

// Check if operator and operands match

if (strcmp(expr1->op, expr2->op) != 0) return false;

// For commutative operations like addition and multiplication,

// check both orders of operands

if (strcmp(expr1->op, "+") == 0 || strcmp(expr1->op, "\*") == 0) {

return ((strcmp(expr1->arg1, expr2->arg1) == 0 &&

strcmp(expr1->arg2, expr2->arg2) == 0) ||

(strcmp(expr1->arg1, expr2->arg2) == 0 &&

strcmp(expr1->arg2, expr2->arg1) == 0));

} else {

// For non-commutative operations, operand order matters

return (strcmp(expr1->arg1, expr2->arg1) == 0 &&

strcmp(expr1->arg2, expr2->arg2) == 0);

}

}

// Function to apply Common Subexpression Elimination

Quadruple\* applyCSE(Quadruple \*quads, int quadCount, int \*newQuadCount) {

Expression \*expressions = (Expression\*)malloc(quadCount \* sizeof(Expression));

int exprCount = 0;

// Allocate space for optimized quadruples

Quadruple \*optimizedQuads = (Quadruple\*)malloc(quadCount \* sizeof(Quadruple));

\*newQuadCount = 0;

// Process each quadruple

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

Quadruple \*quad = &quads[i];

bool foundCommonExpr = false;

// Skip assignments (like a = b) as they're not expressions to optimize

if (strlen(quad->arg2) == 0 && strcmp(quad->op, "=") == 0) {

// Just copy assignment to optimized list

optimizedQuads[\*newQuadCount] = \*quad;

(\*newQuadCount)++;

continue;

}

// Create an expression from this quadruple

Expression currentExpr;

strcpy(currentExpr.op, quad->op);

strcpy(currentExpr.arg1, quad->arg1);

strcpy(currentExpr.arg2, quad->arg2);

strcpy(currentExpr.temp, quad->result);

currentExpr.used = false;

// Check if this expression already exists

for (int j = 0; j < exprCount; j++) {

if (isSameExpression(&currentExpr, &expressions[j])) {

// Common subexpression found

// Instead of recalculating, use the previous result

strcpy(quad->op, "=");

strcpy(quad->arg1, expressions[j].temp);

strcpy(quad->arg2, ""); // No second argument for assignments

foundCommonExpr = true;

break;

}

}

// Add to optimized quadruples

optimizedQuads[\*newQuadCount] = \*quad;

(\*newQuadCount)++;

// If not a common subexpression, add to expressions list

if (!foundCommonExpr) {

expressions[exprCount++] = currentExpr;

}

}

free(expressions);

return optimizedQuads;

}

// Function to print quadruples

void printQuadruples(Quadruple \*quads, int count, const char \*title) {

printf("%s:\n", title);

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

if (strlen(quads[i].arg2) > 0) {

printf("%d: (%s, %s, %s, %s)\n", i,

quads[i].op, quads[i].arg1, quads[i].arg2, quads[i].result);

} else {

printf("%d: (%s, %s, , %s)\n", i,

quads[i].op, quads[i].arg1, quads[i].result);

}

}

printf("\n");

}

// Function to parse input quadruples from a file

Quadruple\* parseQuadruples(const char \*filename, int \*count) {

FILE \*file = fopen(filename, "r");

if (!file) {

printf("Could not open file %s\n", filename);

exit(1);

}

// Count lines in the file

int lineCount = 0;

char ch;

while(!feof(file)) {

ch = fgetc(file);

if(ch == '\n') {

lineCount++;

}

}

// Reset file pointer

rewind(file);

// Allocate memory for quadruples

Quadruple \*quads = (Quadruple\*)malloc(lineCount \* sizeof(Quadruple));

\*count = 0;

// Parse quadruples

char line[100];

while (fgets(line, sizeof(line), file)) {

// Skip empty lines

if (strlen(line) <= 1) continue;

// Remove newline

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

// Parse the quadruple

char \*token = strtok(line, ",");

strcpy(quads[\*count].op, token);

token = strtok(NULL, ",");

strcpy(quads[\*count].arg1, token);

token = strtok(NULL, ",");

if (token) {

strcpy(quads[\*count].arg2, token);

} else {

strcpy(quads[\*count].arg2, "");

}

token = strtok(NULL, ",");

strcpy(quads[\*count].result, token);

(\*count)++;

}

fclose(file);

return quads;

}

// Example function to create sample quadruples for testing

Quadruple\* createSampleQuadruples(int \*count) {

\*count = 10;

Quadruple \*quads = (Quadruple\*)malloc(\*count \* sizeof(Quadruple));

// a = b + c

strcpy(quads[0].op, "+");

strcpy(quads[0].arg1, "b");

strcpy(quads[0].arg2, "c");

strcpy(quads[0].result, "t1");

// d = b + c

strcpy(quads[1].op, "+");

strcpy(quads[1].arg1, "b");

strcpy(quads[1].arg2, "c");

strcpy(quads[1].result, "t2");

// e = d - a

strcpy(quads[2].op, "-");

strcpy(quads[2].arg1, "t2");

strcpy(quads[2].arg2, "t1");

strcpy(quads[2].result, "t3");

// f = b + c

strcpy(quads[3].op, "+");

strcpy(quads[3].arg1, "b");

strcpy(quads[3].arg2, "c");

strcpy(quads[3].result, "t4");

// g = c + b (commutative operation)

strcpy(quads[4].op, "+");

strcpy(quads[4].arg1, "c");

strcpy(quads[4].arg2, "b");

strcpy(quads[4].result, "t5");

// h = f \* g

strcpy(quads[5].op, "\*");

strcpy(quads[5].arg1, "t4");

strcpy(quads[5].arg2, "t5");

strcpy(quads[5].result, "t6");

// i = b - c (not commutative)

strcpy(quads[6].op, "-");

strcpy(quads[6].arg1, "b");

strcpy(quads[6].arg2, "c");

strcpy(quads[6].result, "t7");

// j = c - b (different order, not equal to i)

strcpy(quads[7].op, "-");

strcpy(quads[7].arg1, "c");

strcpy(quads[7].arg2, "b");

strcpy(quads[7].result, "t8");

// k = h + 5

strcpy(quads[8].op, "+");

strcpy(quads[8].arg1, "t6");

strcpy(quads[8].arg2, "5");

strcpy(quads[8].result, "t9");

// l = b + c (again)

strcpy(quads[9].op, "+");

strcpy(quads[9].arg1, "b");

strcpy(quads[9].arg2, "c");

strcpy(quads[9].result, "t10");

return quads;

}

int main(int argc, char \*argv[]) {

int quadCount;

Quadruple \*quads;

// Check if an input file is provided

if (argc > 1) {

quads = parseQuadruples(argv[1], &quadCount);

} else {

printf("No input file provided. Using sample quadruples.\n");

quads = createSampleQuadruples(&quadCount);

}

// Print original quadruples

printQuadruples(quads, quadCount, "Original Quadruples");

// Apply CSE optimization

int optimizedCount;

Quadruple \*optimizedQuads = applyCSE(quads, quadCount, &optimizedCount);

// Print optimized quadruples

printQuadruples(optimizedQuads, optimizedCount, "Optimized Quadruples (After CSE)");

// Free memory

free(quads);

free(optimizedQuads);

return 0;

}

**nano input.txt-**

+,x,y,t1

\*,a,b,t2

+,x,y,t3

-,t3,z,t4

\*,t1,c,t5

\*,a,b,t6

+,t2,t6,t7

\*,t5,2,t8

+,x,y,t9

-,t9,z,t10

/,t10,2,t11

\*,t4,t8,t12

-,t7,t12,result

**gcc cse.c -o cse**

**./cse input.txt**

13. General C optimizations.

(not working 400 line)

14. C++ optimization code.

(too big 600 line)

15. Custom lexer driver in C.

**Handlex.c-**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// Token types

typedef enum {

TOKEN\_IDENTIFIER,

TOKEN\_NUMBER,

TOKEN\_OPERATOR,

TOKEN\_KEYWORD,

TOKEN\_STRING,

TOKEN\_SYMBOL,

TOKEN\_EOF,

TOKEN\_ERROR

} TokenType;

// Token structure

typedef struct {

TokenType type;

char\* value;

int line;

int column;

} Token;

// Keywords

const char\* keywords[] = {

"if", "else", "while", "for", "int", "float", "char", "return", "void", "main"

};

// Maximum token length

#define MAX\_TOKEN\_LENGTH 100

#define MAX\_FILE\_SIZE 1024

// Current position in input

char\* current;

int line = 1;

int column = 1;

// Function to check if a string is a keyword

int isKeyword(const char\* str) {

int numKeywords = sizeof(keywords) / sizeof(keywords[0]);

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

if (strcmp(str, keywords[i]) == 0) {

return 1;

}

}

return 0;

}

// Function to create a new token

Token\* createToken(TokenType type, const char\* value) {

Token\* token = (Token\*)malloc(sizeof(Token));

if (token == NULL) {

fprintf(stderr, "Memory allocation failed for token\n");

exit(EXIT\_FAILURE);

}

token->type = type;

token->value = strdup(value);

if (token->value == NULL) {

fprintf(stderr, "Memory allocation failed for token value\n");

free(token);

exit(EXIT\_FAILURE);

}

token->line = line;

token->column = column - strlen(value);

return token;

}

Token\* getNextToken() {

// Skip whitespace

while (isspace(\*current)) {

if (\*current == '\n') {

line++;

column = 1;

} else {

column++;

}

current++;

}

// Check for end of input

if (\*current == '\0') {

return createToken(TOKEN\_EOF, "EOF");

}

// Handle identifiers and keywords

if (isalpha(\*current) || \*current == '\_') {

char buffer[MAX\_TOKEN\_LENGTH];

int i = 0;

while ((isalnum(\*current) || \*current == '\_') && i < MAX\_TOKEN\_LENGTH - 1) {

buffer[i++] = \*current++;

column++;

}

buffer[i] = '\0';

if (isKeyword(buffer)) {

return createToken(TOKEN\_KEYWORD, buffer);

} else {

return createToken(TOKEN\_IDENTIFIER, buffer);

}

}

// Handle numbers

if (isdigit(\*current)) {

char buffer[MAX\_TOKEN\_LENGTH];

int i = 0;

while ((isdigit(\*current) || \*current == '.') && i < MAX\_TOKEN\_LENGTH - 1) {

buffer[i++] = \*current++;

column++;

}

buffer[i] = '\0';

return createToken(TOKEN\_NUMBER, buffer);

}

// Handle strings

if (\*current == '"') {

char buffer[MAX\_TOKEN\_LENGTH];

int i = 0;

current++;

column++;

while (\*current != '"' && \*current != '\0' && i < MAX\_TOKEN\_LENGTH - 1) {

buffer[i++] = \*current++;

column++;

}

if (\*current == '"') {

current++;

column++;

} else {

return createToken(TOKEN\_ERROR, "Unterminated string");

}

buffer[i] = '\0';

return createToken(TOKEN\_STRING, buffer);

}

// Handle operators and symbols

if (strchr("+-\*/=<>!&|", \*current)) {

char buffer[3] = {0};

buffer[0] = \*current++;

column++;

// Check for two-character operators

if (strchr("=<>|&", \*current)) {

buffer[1] = \*current++;

column++;

}

return createToken(TOKEN\_OPERATOR, buffer);

}

// Handle single-character symbols

if (strchr("(){}[];,:", \*current)) {

char buffer[2] = {\*current++, '\0'};

column++;

return createToken(TOKEN\_SYMBOL, buffer);

}

// Unknown character

char buffer[2] = {\*current++, '\0'};

column++;

return createToken(TOKEN\_ERROR, buffer);

}

// Function to print token information

void printToken(Token\* token) {

const char\* typeNames[] = {

"IDENTIFIER", "NUMBER", "OPERATOR", "KEYWORD",

"STRING", "SYMBOL", "EOF", "ERROR"

};

printf("Token: %-12s Value: %-10s Line: %d, Column: %d\n",

typeNames[token->type], token->value, token->line, token->column);

}

// Function to free token memory

void freeToken(Token\* token) {

free(token->value);

free(token);

}

int main(int argc, char\* argv[]) {

if (argc != 2) {

printf("Usage: %s <input\_file>\n", argv[0]);

return 1;

}

FILE\* file = fopen(argv[1], "r");

if (!file) {

printf("Error: Could not open file %s\n", argv[1]);

return 1;

}

// Read the entire file

char\* input = (char\*)malloc(MAX\_FILE\_SIZE);

if (!input) {

printf("Error: Memory allocation failed\n");

fclose(file);

return 1;

}

size\_t fileSize = fread(input, 1, MAX\_FILE\_SIZE - 1, file);

input[fileSize] = '\0';

fclose(file);

current = input;

printf("Lexical Analysis Results:\n");

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

Token\* token;

while (1) {

token = getNextToken();

printToken(token);

if (token->type == TOKEN\_EOF || token->type == TOKEN\_ERROR) {

freeToken(token);

break;

}

freeToken(token);

}

free(input);

return 0;

}

**Handlex.txt-**

int main() {

int a = 5;

float b = 3.14;

char c = 'z';

a = a + 10;

if (a > 10) {

b = b \* 2;

}

return 0;

}

**gcc handlex.c -o handlex**

**./handlex handlex.txt**

16. Java-based lexer.

//handwritten lexer using java

import java.util.Arrays;

import java.util.List;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class Lexer {

    // Token types

    public enum TokenType {

        IDENTIFIER,

        NUMBER,

        OPERATOR,

        KEYWORD,

        STRING,

        SYMBOL,

        EOF,

        ERROR

    }

    // Token class

    public static class Token {

        public final TokenType type;

        public final String value;

        public final int line;

        public final int column;

        public Token(TokenType type, String value, int line, int column) {

            this.type = type;

            this.value = value;

            this.line = line;

            this.column = column;

        }

        @Override

        public String toString() {

            return String.format("Token: %-12s Value: %-10s Line: %d, Column: %d",

                    type, value, line, column);

        }

    }

    // Keywords

    private static final List<String> KEYWORDS = Arrays.asList(

            "if", "else", "while", "for", "int", "float", "char", "return", "void", "main");

    // Input text and position tracking

    private final String input;

    private int position;

    private int line;

    private int column;

    public Lexer(String input) {

        this.input = input;

        this.position = 0;

        this.line = 1;

        this.column = 1;

    }

    // Check if a string is a keyword

    private boolean isKeyword(String str) {

        return KEYWORDS.contains(str);

    }

    // Get the next token

    public Token getNextToken() {

        // Skip whitespace

        while (position < input.length() && Character.isWhitespace(input.charAt(position))) {

            if (input.charAt(position) == '\n') {

                line++;

                column = 1;

            } else {

                column++;

            }

            position++;

        }

        // Check for end of input

        if (position >= input.length()) {

            return new Token(TokenType.EOF, "EOF", line, column);

        }

        char current = input.charAt(position);

        // Handle identifiers and keywords

        if (Character.isLetter(current) || current == '\_') {

            StringBuilder buffer = new StringBuilder();

            while (position < input.length() &&

                    (Character.isLetterOrDigit(input.charAt(position)) ||

                            input.charAt(position) == '\_')) {

                buffer.append(input.charAt(position));

                position++;

                column++;

            }

            String value = buffer.toString();

            return new Token(

                    isKeyword(value) ? TokenType.KEYWORD : TokenType.IDENTIFIER,

                    value,

                    line,

                    column - value.length());

        }

        // Handle numbers

        if (Character.isDigit(current)) {

            StringBuilder buffer = new StringBuilder();

            while (position < input.length() &&

                    (Character.isDigit(input.charAt(position)) ||

                            input.charAt(position) == '.')) {

                buffer.append(input.charAt(position));

                position++;

                column++;

            }

            return new Token(TokenType.NUMBER, buffer.toString(), line, column - buffer.length());

        }

        // Handle strings

        if (current == '"') {

            StringBuilder buffer = new StringBuilder();

            position++;

            column++;

            while (position < input.length() && input.charAt(position) != '"') {

                buffer.append(input.charAt(position));

                position++;

                column++;

            }

            if (position < input.length() && input.charAt(position) == '"') {

                position++;

                column++;

                return new Token(TokenType.STRING, buffer.toString(), line, column - buffer.length() - 2);

            } else {

                return new Token(TokenType.ERROR, "Unterminated string", line, column);

            }

        }

        // Handle operators and symbols

        if ("+-\*/=<>!&|".indexOf(current) != -1) {

            StringBuilder buffer = new StringBuilder();

            buffer.append(current);

            position++;

            column++;

            // Check for two-character operators

            if (position < input.length() && "=<>|&".indexOf(input.charAt(position)) != -1) {

                buffer.append(input.charAt(position));

                position++;

                column++;

            }

            return new Token(TokenType.OPERATOR, buffer.toString(), line, column - buffer.length());

        }

        // Handle single-character symbols

        if ("(){}[];,:. ".indexOf(current) != -1) {

            String value = String.valueOf(current);

            position++;

            column++;

            return new Token(TokenType.SYMBOL, value, line, column - 1);

        }

        // Unknown character

        String value = String.valueOf(current);

        position++;

        column++;

        return new Token(TokenType.ERROR, value, line, column - 1);

    }

    public static void main(String[] args) {

        if (args.length != 1) {

            System.out.println("Usage: java Lexer <input\_file>");

            System.exit(1);

        }

        try {

            // Read the entire file

            StringBuilder inputBuilder = new StringBuilder();

            try (BufferedReader reader = new BufferedReader(new FileReader(args[0]))) {

                String line;

                while ((line = reader.readLine()) != null) {

                    inputBuilder.append(line).append("\n");

                }

            }

            String input = inputBuilder.toString();

            Lexer lexer = new Lexer(input);

            System.out.println("Lexical Analysis Results:");

            System.out.println("------------------------");

            Token token;

            do {

                token = lexer.getNextToken();

                System.out.println(token);

            } while (token.type != TokenType.EOF && token.type != TokenType.ERROR);

        } catch (IOException e) {

            System.out.println("Error reading file: " + e.getMessage());

            System.exit(1);

        }

    }

}

// Note to run :name the code Lexer 1]javac Lexer.java then 2] java Lexer

// test\_input.txt

// int main() {

// int x = 10;

// if (x > 5) {

// printf("Hello, world!");

// }

// return 0;

// }

**javac Lexer.java**

**java Lexer**