flex lex.l gcc [lex.yy](http://lex.yy).c a.exe

flex lex.l bison -d y.y gcc -o parser y.tab.c [lex.yy](http://lex.yy).c parser.exe **1.C language Lexical Analyzer**

%{

#include <stdio.h>

#include <stdlib.h>

int line\_num = 1;

%}

/\* Regular definitions \*/

DIGIT [0-9]

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

NUMBER {DIGIT}+

FLOAT {DIGIT}+"."{DIGIT}+

WHITESPACE [ \t]+

NEWLINE \n

%%

"int" { printf("KEYWORD: %s\n", yytext); }

"float" { printf("KEYWORD: %s\n", yytext); }

"char" { printf("KEYWORD: %s\n", yytext); }

"void" { printf("KEYWORD: %s\n", yytext); }

"if" { printf("KEYWORD: %s\n", yytext); }

"else" { printf("KEYWORD: %s\n", yytext); }

"while" { printf("KEYWORD: %s\n", yytext); }

"for" { printf("KEYWORD: %s\n", yytext); }

"return" { printf("KEYWORD: %s\n", yytext); }

"+" { printf("OPERATOR: %s\n", yytext); }

"-" { printf("OPERATOR: %s\n", yytext); }

"\*" { printf("OPERATOR: %s\n", yytext); }

"/" { printf("OPERATOR: %s\n", yytext); }

"=" { printf("OPERATOR: %s\n", yytext); }

"==" { printf("OPERATOR: %s\n", yytext); }

"!=" { printf("OPERATOR: %s\n", yytext); }

"<" { printf("OPERATOR: %s\n", yytext); }

">" { printf("OPERATOR: %s\n", yytext); }

"(" { printf("PUNCTUATION: %s\n", yytext); }

")" { printf("PUNCTUATION: %s\n", yytext); }

"{" { printf("PUNCTUATION: %s\n", yytext); }

"}" { printf("PUNCTUATION: %s\n", yytext); }

";" { printf("PUNCTUATION: %s\n", yytext); }

"," { printf("PUNCTUATION: %s\n", yytext); }

{ID} { printf("IDENTIFIER: %s\n", yytext); }

{NUMBER} { printf("INTEGER: %s\n", yytext); }

{FLOAT} { printf("FLOAT: %s\n", yytext); }

\"([^\"\n])\*\" { printf("STRING: %s\n", yytext); }

{WHITESPACE} { /\* ignore whitespace \*/ }

{NEWLINE} { line\_num++; }

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

%%

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

if (argc > 1) {

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

if (!file) {

fprintf(stderr, "Could not open file %s\n", argv[1]);

return 1;

}

yyin = file;

}

printf("C Lexical Analyzer\n");

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

yylex();

if (argc > 1) {

fclose(yyin);

}

return 0;

}

int yywrap() {

return 1;

}

**2.SQL lex analyser:**

%{

#include <stdio.h>

#include <stdlib.h>

int line\_num = 1;

%}

%option noyywrap

%option case-insensitive

DIGIT [0-9]

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

NUMBER {DIGIT}+

FLOAT {DIGIT}+"."{DIGIT}+

WHITESPACE [ \t]+

NEWLINE \n

%%

SELECT { printf("KEYWORD: %s\n", yytext); }

FROM { printf("KEYWORD: %s\n", yytext); }

WHERE { printf("KEYWORD: %s\n", yytext); }

INSERT { printf("KEYWORD: %s\n", yytext); }

INTO { printf("KEYWORD: %s\n", yytext); }

VALUES { printf("KEYWORD: %s\n", yytext); }

UPDATE { printf("KEYWORD: %s\n", yytext); }

SET { printf("KEYWORD: %s\n", yytext); }

DELETE { printf("KEYWORD: %s\n", yytext); }

CREATE { printf("KEYWORD: %s\n", yytext); }

TABLE { printf("KEYWORD: %s\n", yytext); }

DROP { printf("KEYWORD: %s\n", yytext); }

ALTER { printf("KEYWORD: %s\n", yytext); }

AND { printf("OPERATOR: %s\n", yytext); }

OR { printf("OPERATOR: %s\n", yytext); }

NOT { printf("OPERATOR: %s\n", yytext); }

NULL { printf("KEYWORD: %s\n", yytext); }

INT { printf("DATA\_TYPE: %s\n", yytext); }

VARCHAR { printf("DATA\_TYPE: %s\n", yytext); }

CHAR { printf("DATA\_TYPE: %s\n", yytext); }

DATE { printf("DATA\_TYPE: %s\n", yytext); }

"=" { printf("OPERATOR: %s\n", yytext); }

"<=" { printf("OPERATOR: %s\n", yytext); }

">=" { printf("OPERATOR: %s\n", yytext); }

"<>" { printf("OPERATOR: %s\n", yytext); }

"!=" { printf("OPERATOR: %s\n", yytext); }

"<" { printf("OPERATOR: %s\n", yytext); }

">" { printf("OPERATOR: %s\n", yytext); }

"+" { printf("OPERATOR: %s\n", yytext); }

"-" { printf("OPERATOR: %s\n", yytext); }

"\*" { printf("OPERATOR: %s\n", yytext); }

"/" { printf("OPERATOR: %s\n", yytext); }

"(" { printf("PUNCTUATION: %s\n", yytext); }

")" { printf("PUNCTUATION: %s\n", yytext); }

";" { printf("PUNCTUATION: %s\n", yytext); }

"," { printf("PUNCTUATION: %s\n", yytext); }

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

{FLOAT} { printf("FLOAT: %s\n", yytext); }

{NUMBER} { printf("INTEGER: %s\n", yytext); }

{ID} { printf("IDENTIFIER: %s\n", yytext); }

\'[^\'\n]\*\' { printf("STRING: %s\n", yytext); }

{WHITESPACE} { /\* Ignore whitespace \*/ }

{NEWLINE} { line\_num++; }

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

%%

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

if (argc > 1) {

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

if (!file) {

fprintf(stderr, "Could not open file %s\n", argv[1]);

return 1;

}

yyin = file;

}

printf("SQL Lexical Analyzer\n");

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

yylex();

if (argc > 1) {

fclose(yyin);

}

return 0;

}

**3. C++ Lex analyzer:**

%{

#include <stdio.h>

#include <stdlib.h>

int line\_num = 1;

%}

/\* Regular definitions \*/

DIGIT [0-9]

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

NUMBER {DIGIT}+

FLOAT {DIGIT}+"."{DIGIT}+

WHITESPACE [ \t]+

NEWLINE \n

%%

"int" { printf("KEYWORD: %s\n", yytext); }

"float" { printf("KEYWORD: %s\n", yytext); }

"char" { printf("KEYWORD: %s\n", yytext); }

"double" { printf("KEYWORD: %s\n", yytext); }

"bool" { printf("KEYWORD: %s\n", yytext); }

"void" { printf("KEYWORD: %s\n", yytext); }

"if" { printf("KEYWORD: %s\n", yytext); }

"else" { printf("KEYWORD: %s\n", yytext); }

"while" { printf("KEYWORD: %s\n", yytext); }

"for" { printf("KEYWORD: %s\n", yytext); }

"return" { printf("KEYWORD: %s\n", yytext); }

"class" { printf("KEYWORD: %s\n", yytext); }

"public" { printf("KEYWORD: %s\n", yytext); }

"private" { printf("KEYWORD: %s\n", yytext); }

"protected" { printf("KEYWORD: %s\n", yytext); }

"template" { printf("KEYWORD: %s\n", yytext); }

"new" { printf("KEYWORD: %s\n", yytext); }

"delete" { printf("KEYWORD: %s\n", yytext); }

"true" { printf("KEYWORD: %s\n", yytext); }

"false" { printf("KEYWORD: %s\n", yytext); }

"namespace" { printf("KEYWORD: %s\n", yytext); }

"using" { printf("KEYWORD: %s\n", yytext); }

"std" { printf("IDENTIFIER: %s\n", yytext); }

"cout" { printf("IDENTIFIER: %s\n", yytext); }

"cin" { printf("IDENTIFIER: %s\n", yytext); }

"+" { printf("OPERATOR: %s\n", yytext); }

"-" { printf("OPERATOR: %s\n", yytext); }

"\*" { printf("OPERATOR: %s\n", yytext); }

"/" { printf("OPERATOR: %s\n", yytext); }

"=" { printf("OPERATOR: %s\n", yytext); }

"==" { printf("OPERATOR: %s\n", yytext); }

"!=" { printf("OPERATOR: %s\n", yytext); }

"<" { printf("OPERATOR: %s\n", yytext); }

">" { printf("OPERATOR: %s\n", yytext); }

">>" { printf("OPERATOR: %s\n", yytext); }

"<<" { printf("OPERATOR: %s\n", yytext); }

"++" { printf("OPERATOR: %s\n", yytext); }

"--" { printf("OPERATOR: %s\n", yytext); }

"&&" { printf("OPERATOR: %s\n", yytext); }

"||" { printf("OPERATOR: %s\n", yytext); }

"::" { printf("OPERATOR: %s\n", yytext); }

"(" { printf("PUNCTUATION: %s\n", yytext); }

")" { printf("PUNCTUATION: %s\n", yytext); }

"{" { printf("PUNCTUATION: %s\n", yytext); }

"}" { printf("PUNCTUATION: %s\n", yytext); }

";" { printf("PUNCTUATION: %s\n", yytext); }

"," { printf("PUNCTUATION: %s\n", yytext); }

"<"[a-zA-Z0-9\_]+">"{1} { printf("TEMPLATE: %s\n", yytext); }

"//".\* { printf("COMMENT: %s\n", yytext); }

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

{ID} { printf("IDENTIFIER: %s\n", yytext); }

{NUMBER} { printf("INTEGER: %s\n", yytext); }

{FLOAT} { printf("FLOAT: %s\n", yytext); }

\"([^\"\n])\*\" { printf("STRING: %s\n", yytext); }

{WHITESPACE} { /\* ignore whitespace \*/ }

{NEWLINE} { line\_num++; }

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

%%

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

if (argc > 1) {

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

if (!file) {

fprintf(stderr, "Could not open file %s\n", argv[1]);

return 1;

}

yyin = file;

}

printf("C++ Lexical Analyzer\n");

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

yylex();

if (argc > 1) {

fclose(yyin);

}

return 0;

}

int yywrap() {

return 1;

}

**4. HTML lex analyzer:**

%{

#include <stdio.h>

%}

%%

"<!DOCTYPE"[^>]\*">" { printf("DOCTYPE declaration\n"); }

"<html>" { printf("HTML start tag\n"); }

"</html>" { printf("HTML end tag\n"); }

"<head>" { printf("HEAD start tag\n"); }

"</head>" { printf("HEAD end tag\n"); }

"<body>" { printf("BODY start tag\n"); }

"</body>" { printf("BODY end tag\n"); }

"<title>" { printf("TITLE start tag\n"); }

"</title>" { printf("TITLE end tag\n"); }

"<h[1-6]>" { printf("Header start tag\n"); }

"</h[1-6]>" { printf("Header end tag\n"); }

"<p>" { printf("Paragraph start tag\n"); }

"</p>" { printf("Paragraph end tag\n"); }

"<[a-zA-Z]+[^>]\*>" { printf("Generic start tag\n"); }

"</[a-zA-Z]+>" { printf("Generic end tag\n"); }

\"[^\"]\*\" { printf("Attribute value: %s\n", yytext); }

[a-zA-Z0-9 ,.?!]+ { printf("Text content: %s\n", yytext); }

\n { /\* ignore newlines \*/ }

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

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

%%

int yywrap() {

return 1;

}

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

yylex();

return 0;

}

flex switchlex.l

gcc lex.yy.c

a.exe < sample.html

Sample.html file  
<!DOCTYPE html>

<html>

<head>

<title>Test Page 123</title>

</head>

<body>

<h1>Hello, World!</h1>

<p>This is a simple paragraph with numbers 456 and symbols like @, #, $, %.</p>

<p>2025 is the year of AI evolution!</p>

</body>

</html>

**5. Java lex analyzer**

%{

#include <stdio.h>

#include <stdlib.h>

int line\_num = 1;

%}

/\* Regular definitions \*/

DIGIT [0-9]

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

NUMBER {DIGIT}+

FLOAT {DIGIT}+"."{DIGIT}+([eE][+-]?{DIGIT}+)?

WHITESPACE [ \t\r]+

NEWLINE \n

%%

"abstract" { printf("KEYWORD: %s\n", yytext); }

"assert" { printf("KEYWORD: %s\n", yytext); }

"boolean" { printf("KEYWORD: %s\n", yytext); }

"break" { printf("KEYWORD: %s\n", yytext); }

"byte" { printf("KEYWORD: %s\n", yytext); }

"case" { printf("KEYWORD: %s\n", yytext); }

"catch" { printf("KEYWORD: %s\n", yytext); }

"char" { printf("KEYWORD: %s\n", yytext); }

"class" { printf("KEYWORD: %s\n", yytext); }

"const" { printf("KEYWORD: %s\n", yytext); }

"continue" { printf("KEYWORD: %s\n", yytext); }

"default" { printf("KEYWORD: %s\n", yytext); }

"do" { printf("KEYWORD: %s\n", yytext); }

"double" { printf("KEYWORD: %s\n", yytext); }

"else" { printf("KEYWORD: %s\n", yytext); }

"enum" { printf("KEYWORD: %s\n", yytext); }

"extends" { printf("KEYWORD: %s\n", yytext); }

"final" { printf("KEYWORD: %s\n", yytext); }

"finally" { printf("KEYWORD: %s\n", yytext); }

"float" { printf("KEYWORD: %s\n", yytext); }

"for" { printf("KEYWORD: %s\n", yytext); }

"if" { printf("KEYWORD: %s\n", yytext); }

"implements" { printf("KEYWORD: %s\n", yytext); }

"import" { printf("KEYWORD: %s\n", yytext); }

"instanceof" { printf("KEYWORD: %s\n", yytext); }

"int" { printf("KEYWORD: %s\n", yytext); }

"interface" { printf("KEYWORD: %s\n", yytext); }

"long" { printf("KEYWORD: %s\n", yytext); }

"native" { printf("KEYWORD: %s\n", yytext); }

"new" { printf("KEYWORD: %s\n", yytext); }

"package" { printf("KEYWORD: %s\n", yytext); }

"private" { printf("KEYWORD: %s\n", yytext); }

"protected" { printf("KEYWORD: %s\n", yytext); }

"public" { printf("KEYWORD: %s\n", yytext); }

"return" { printf("KEYWORD: %s\n", yytext); }

"short" { printf("KEYWORD: %s\n", yytext); }

"static" { printf("KEYWORD: %s\n", yytext); }

"strictfp" { printf("KEYWORD: %s\n", yytext); }

"super" { printf("KEYWORD: %s\n", yytext); }

"switch" { printf("KEYWORD: %s\n", yytext); }

"synchronized" { printf("KEYWORD: %s\n", yytext); }

"this" { printf("KEYWORD: %s\n", yytext); }

"throw" { printf("KEYWORD: %s\n", yytext); }

"throws" { printf("KEYWORD: %s\n", yytext); }

"transient" { printf("KEYWORD: %s\n", yytext); }

"try" { printf("KEYWORD: %s\n", yytext); }

"void" { printf("KEYWORD: %s\n", yytext); }

"volatile" { printf("KEYWORD: %s\n", yytext); }

"while" { printf("KEYWORD: %s\n", yytext); }

"true" { printf("BOOLEAN: %s\n", yytext); }

"false" { printf("BOOLEAN: %s\n", yytext); }

"null" { printf("LITERAL: %s\n", yytext); }

"+" { printf("OPERATOR: %s\n", yytext); }

"-" { printf("OPERATOR: %s\n", yytext); }

"\*" { printf("OPERATOR: %s\n", yytext); }

"/" { printf("OPERATOR: %s\n", yytext); }

"%" { printf("OPERATOR: %s\n", yytext); }

"=" { printf("OPERATOR: %s\n", yytext); }

"+=" { printf("OPERATOR: %s\n", yytext); }

"-=" { printf("OPERATOR: %s\n", yytext); }

"\*=" { printf("OPERATOR: %s\n", yytext); }

"/=" { printf("OPERATOR: %s\n", yytext); }

"%=" { printf("OPERATOR: %s\n", yytext); }

"==" { printf("OPERATOR: %s\n", yytext); }

"!=" { printf("OPERATOR: %s\n", yytext); }

">" { printf("OPERATOR: %s\n", yytext); }

"<" { printf("OPERATOR: %s\n", yytext); }

">=" { printf("OPERATOR: %s\n", yytext); }

"<=" { printf("OPERATOR: %s\n", yytext); }

"&&" { printf("OPERATOR: %s\n", yytext); }

"||" { printf("OPERATOR: %s\n", yytext); }

"!" { printf("OPERATOR: %s\n", yytext); }

"++" { printf("OPERATOR: %s\n", yytext); }

"--" { printf("OPERATOR: %s\n", yytext); }

"&" { printf("OPERATOR: %s\n", yytext); }

"|" { printf("OPERATOR: %s\n", yytext); }

"^" { printf("OPERATOR: %s\n", yytext); }

"~" { printf("OPERATOR: %s\n", yytext); }

"<<" { printf("OPERATOR: %s\n", yytext); }

">>" { printf("OPERATOR: %s\n", yytext); }

">>>" { printf("OPERATOR: %s\n", yytext); }

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

"(" { printf("PUNCTUATION: %s\n", yytext); }

")" { printf("PUNCTUATION: %s\n", yytext); }

"{" { printf("PUNCTUATION: %s\n", yytext); }

"}" { printf("PUNCTUATION: %s\n", yytext); }

"[" { printf("PUNCTUATION: %s\n", yytext); }

"]" { printf("PUNCTUATION: %s\n", yytext); }

";" { printf("PUNCTUATION: %s\n", yytext); }

"," { printf("PUNCTUATION: %s\n", yytext); }

":" { printf("PUNCTUATION: %s\n", yytext); }

"?" { printf("PUNCTUATION: %s\n", yytext); }

"//".\* { printf("COMMENT: %s\n", yytext); }

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

{ID} { printf("IDENTIFIER: %s\n", yytext); }

{NUMBER} { printf("INTEGER: %s\n", yytext); }

{FLOAT} { printf("FLOAT: %s\n", yytext); }

\"([^\"\n]|\\\")\*\" { printf("STRING: %s\n", yytext); }

\'([^\'\\]|\\.)\' { printf("CHAR: %s\n", yytext); }

{WHITESPACE} { /\* ignore whitespace \*/ }

{NEWLINE} { line\_num++; }

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

%%

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

if (argc > 1) {

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

if (!file) {

fprintf(stderr, "Could not open file %s\n", argv[1]);

return 1;

}

yyin = file;

}

printf("Java Lexical Analyzer\n");

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

yylex();

if (argc > 1) {

fclose(yyin);

}

return 0;

}

int yywrap() {

return 1;

}

**6. Lex analyzer for structured input formats**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int line\_num = 1;

int valid\_structure = 1;

int error\_count = 0;

int current\_state = 0;

char current\_section[100] = "";

void report\_error(const char\* message) {

fprintf(stderr, "Error at line %d: %s\n", line\_num, message);

error\_count++;

valid\_structure = 0;

}

%}

%x SECTION\_HEADER

%x KEY\_VALUE

%x VALUE

%x ARRAY

%x ARRAY\_VALUE

%x COMMENT

/\* Regular definitions \*/

WHITESPACE [ \t\r]+

NEWLINE \n

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

NUMBER [0-9]+

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

%%

"#".\* { printf("COMMENT: %s\n", yytext); }

"[" {

BEGIN(SECTION\_HEADER);

printf("SECTION\_START: %s\n", yytext);

}

<SECTION\_HEADER>{ID} {

strcpy(current\_section, yytext);

printf("SECTION\_NAME: %s\n", yytext);

}

<SECTION\_HEADER>"]" {

BEGIN(INITIAL);

printf("SECTION\_END: %s\n", yytext);

}

<SECTION\_HEADER>{WHITESPACE} { /\* ignore whitespace \*/ }

<SECTION\_HEADER>{NEWLINE} {

report\_error("Unexpected newline in section header");

BEGIN(INITIAL);

line\_num++;

}

<SECTION\_HEADER>. {

report\_error("Invalid character in section header");

}

{ID} {

BEGIN(KEY\_VALUE);

printf("KEY: %s\n", yytext);

}

<KEY\_VALUE>"=" {

BEGIN(VALUE);

printf("EQUALS: %s\n", yytext);

}

<KEY\_VALUE>{WHITESPACE} { /\* ignore whitespace \*/ }

<KEY\_VALUE>{NEWLINE} {

report\_error("Expected '=' after key");

BEGIN(INITIAL);

line\_num++;

}

<KEY\_VALUE>. {

report\_error("Invalid character in key");

}

<VALUE>\"([^\"\n]|\\\")\*\" {

printf("STRING\_VALUE: %s\n", yytext);

BEGIN(INITIAL);

}

<VALUE>{NUMBER} {

printf("NUMBER\_VALUE: %s\n", yytext);

BEGIN(INITIAL);

}

<VALUE>{FLOAT} {

printf("FLOAT\_VALUE: %s\n", yytext);

BEGIN(INITIAL);

}

<VALUE>"true"|"false" {

printf("BOOLEAN\_VALUE: %s\n", yytext);

BEGIN(INITIAL);

}

<VALUE>"[" {

BEGIN(ARRAY);

printf("ARRAY\_START: %s\n", yytext);

}

<VALUE>{WHITESPACE} { /\* ignore whitespace \*/ }

<VALUE>{NEWLINE} {

report\_error("Expected value after '='");

BEGIN(INITIAL);

line\_num++;

}

<VALUE>. {

report\_error("Invalid character in value");

}

<ARRAY>{WHITESPACE} { /\* ignore whitespace \*/ }

<ARRAY>{NEWLINE} { line\_num++; }

<ARRAY>"]" {

BEGIN(INITIAL);

printf("ARRAY\_END: %s\n", yytext);

}

<ARRAY>"," {

printf("COMMA: %s\n", yytext);

}

<ARRAY>\"([^\"\n]|\\\")\*\" {

printf("ARRAY\_STRING: %s\n", yytext);

}

<ARRAY>{NUMBER} {

printf("ARRAY\_NUMBER: %s\n", yytext);

}

<ARRAY>{FLOAT} {

printf("ARRAY\_FLOAT: %s\n", yytext);

}

<ARRAY>"true"|"false" {

printf("ARRAY\_BOOLEAN: %s\n", yytext);

}

<ARRAY>. {

report\_error("Invalid character in array");

}

{WHITESPACE} { /\* ignore whitespace \*/ }

{NEWLINE} { line\_num++; }

. {

report\_error("Invalid character in input");

}

%%

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

if (argc > 1) {

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

if (!file) {

fprintf(stderr, "Could not open file %s\n", argv[1]);

return 1;

}

yyin = file;

}

printf("Structured Input Format Lexical Analyzer\n");

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

yylex();

if (argc > 1) {

fclose(yyin);

}

if (valid\_structure) {

printf("\nInput validation successful! The input conforms to the structured format.\n");

} else {

printf("\nInput validation failed with %d errors.\n", error\_count);

}

return 0;

}

int yywrap() {

return 1;

}

**7. Lex program to count vowels, constants,digits and whitespaces**

%{

#include <stdio.h>

#include <stdlib.h>

int vowels = 0;

int consonants = 0;

int digits = 0;

int whitespaces = 0;

int other\_chars = 0;

%}

%%

[aeiouAEIOU] { vowels++; printf("Vowel: %s\n", yytext); }

[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ] { consonants++; printf("Consonant: %s\n", yytext); }

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

[ \t\n\r] { whitespaces++; printf("Whitespace: %s\n", yytext); }

. { other\_chars++; printf("Other character: %s\n", yytext); }

%%

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

if (argc > 1) {

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

if (!file) {

fprintf(stderr, "Could not open file %s\n", argv[1]);

return 1;

}

yyin = file;

} else {

printf("Enter text (Ctrl+D to end on Unix/Linux/macOS, Ctrl+Z on Windows):\n");

}

yylex();

if (argc > 1) {

fclose(yyin);

}

printf("\nCounting Results:\n");

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

printf("Vowels: %d\n", vowels);

printf("Consonants: %d\n", consonants);

printf("Digits: %d\n", digits);

printf("Whitespaces: %d\n", whitespaces);

printf("Other characters: %d\n", other\_chars);

printf("Total characters: %d\n", vowels + consonants + digits + whitespaces + other\_chars);

return 0;

}

int yywrap() {

return 1;

}

**8. Lex to check for valid palindrome**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

char normalized\_input[10000];

int char\_count = 0;

int is\_palindrome() {

int i, j;

int len = strlen(normalized\_input);

// Check if the normalized string is a palindrome

for(i = 0, j = len - 1; i < j; i++, j--) {

if(normalized\_input[i] != normalized\_input[j]) {

return 0; // Not a palindrome

}

}

return 1; // Is a palindrome

}

%}

%%

[a-zA-Z0-9] {

if(char\_count < 9999) {

normalized\_input[char\_count++] = tolower(yytext[0]);

normalized\_input[char\_count] = '\0';

}

}

. { /\* Ignore non-alphanumeric characters \*/ }

%%

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

if (argc > 1) {

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

if (!file) {

fprintf(stderr, "Could not open file %s\n", argv[1]);

return 1;

}

yyin = file;

} else {

printf("Enter text (Ctrl+D to end on Unix/Linux/macOS, Ctrl+Z on Windows):\n");

}

// Initialize the normalized input buffer

normalized\_input[0] = '\0';

yylex();

if (argc > 1) {

fclose(yyin);

}

printf("\nPalindrome Check Result:\n");

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

if(char\_count == 0) {

printf("No valid characters entered. Cannot check for palindrome.\n");

} else {

printf("Normalized input (ignoring case and non-alphanumeric): %s\n", normalized\_input);

if(is\_palindrome()) {

printf("Result: YES, the input IS a valid palindrome.\n");

} else {

printf("Result: NO, the input is NOT a palindrome.\n");

}

}

return 0;

}

int yywrap() {

return 1;

}

**9.Lex program to count the number of words in a given input:**

%{

#include <stdio.h>

int word\_count = 0;

%}

%%

[a-zA-Z]+ { word\_count++; } /\* Count words (sequences of letters) \*/

.|\n { /\* Ignore other characters \*/ }

%%

int main() {

printf("Enter text (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):\n");

yylex(); // Lexical analysis

printf("Number of words: %d\n", word\_count); // Output word count

return 0;

}

int yywrap() {

return 1; // Return 1 to signal end of input

**}**

**10.lex analyser for subset of English language**

%{

#include <stdio.h>

int noun\_count = 0;

int verb\_count = 0;

int adj\_count = 0;

int adv\_count = 0;

int prep\_count = 0;

int pron\_count = 0;

int conj\_count = 0;

int misc\_count = 0;

%}

%%

cat|dog|house|car|book|tree|man|woman|time|day { printf("NOUN: %s\n", yytext); noun\_count++; }

run|walk|jump|eat|sleep|read|write|talk|think|see { printf("VERB: %s\n", yytext); verb\_count++; }

big|small|good|bad|hot|cold|new|old|happy|sad { printf("ADJECTIVE: %s\n", yytext); adj\_count++; }

quickly|slowly|carefully|easily|loudly|quietly { printf("ADVERB: %s\n", yytext); adv\_count++; }

in|on|at|by|with|from|to|for|of|about { printf("PREPOSITION: %s\n", yytext); prep\_count++; }

I|you|he|she|it|we|they|me|him|her { printf("PRONOUN: %s\n", yytext); pron\_count++; }

and|but|or|nor|yet|so|because|if|when|while { printf("CONJUNCTION: %s\n", yytext); conj\_count++; }

[a-zA-Z]+ { printf("UNKNOWN: %s\n", yytext); misc\_count++; }

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

. { /\* Ignore other characters \*/ }

%%

int main() {

printf("Enter text (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):\n");

yylex();

printf("\n===== ANALYSIS RESULTS =====\n");

printf("Nouns: %d\n", noun\_count);

printf("Verbs: %d\n", verb\_count);

printf("Adjectives: %d\n", adj\_count);

printf("Adverbs: %d\n", adv\_count);

printf("Prepositions: %d\n", prep\_count);

printf("Pronouns: %d\n", pron\_count);

printf("Conjunctions: %d\n", conj\_count);

printf("Unknown words: %d\n", misc\_count);

return 0;

}

int yywrap() {

return 1;

}

**11.parser for an expression grammer using yacc and lex**

Lex file:

%{

#include <stdio.h>

#include "y.tab.h"

%}

%%

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

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return TIMES; }

"/" { return DIVIDE; }

"(" { return LPAREN; }

")" { return RPAREN; }

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

. { /\* Ignore other characters \*/ }

%%

int yywrap() {

return 1;

}

Yacc file:

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(char \*s);

int yylex();

%}

%token NUMBER PLUS MINUS TIMES DIVIDE LPAREN RPAREN

%%

program:

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

;

expr:

term { $$ = $1; }

| expr PLUS term { $$ = $1 + $3; }

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

;

term:

factor { $$ = $1; }

| term TIMES factor { $$ = $1 \* $3; }

| term DIVIDE factor {

if ($3 == 0) {

yyerror("Division by zero");

$$ = 0;

} else {

$$ = $1 / $3;

}

}

;

factor:

NUMBER { $$ = $1; }

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

| MINUS factor { $$ = -$2; }

;

%%

void yyerror(char \*s) {

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

}

int main() {

printf("Enter expression (e.g., 3+4\*2):\n");

yyparse();

return 0;

}

12. Validate and parse if else statements

Lex file:

%{

#include <stdio.h>

#include "y.tab.h"

%}

%%

"if" { return IF; }

"else" { return ELSE; }

"(" { return LPAREN; }

")" { return RPAREN; }

"{" { return LBRACE; }

"}" { return RBRACE; }

"==" { return EQ; }

"!=" { return NEQ; }

"<" { return LT; }

">" { return GT; }

"<=" { return LE; }

">=" { return GE; }

"&&" { return AND; }

"||" { return OR; }

"!" { return NOT; }

"=" { return ASSIGN; }

";" { return SEMICOLON; }

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

[0-9]+ { return NUMBER; }

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

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

Yacc file:

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(char \*s);

int yylex();

int indent = 0;

void print\_indent() {

for (int i = 0; i < indent; i++) printf(" ");

}

%}

%token IF ELSE LPAREN RPAREN LBRACE RBRACE

%token EQ NEQ LT GT LE GE AND OR NOT

%token ASSIGN SEMICOLON IDENTIFIER NUMBER

%nonassoc IFX

%nonassoc ELSE

%%

program:

statement\_list

;

statement\_list:

statement

| statement\_list statement

;

statement:

if\_statement

| assignment\_statement

| block\_statement

;

if\_statement:

IF LPAREN expression RPAREN statement %prec IFX {

printf("IF statement with condition\n");

}

| IF LPAREN expression RPAREN statement ELSE statement {

printf("IF-ELSE statement with condition\n");

}

;

block\_statement:

LBRACE {

indent++;

print\_indent();

printf("BEGIN BLOCK\n");

} statement\_list RBRACE {

print\_indent();

printf("END BLOCK\n");

indent--;

}

| LBRACE RBRACE {

print\_indent();

printf("EMPTY BLOCK\n");

}

;

assignment\_statement:

IDENTIFIER ASSIGN expression SEMICOLON {

print\_indent();

printf("ASSIGNMENT statement\n");

}

;

expression:

logical\_expr

;

logical\_expr:

relational\_expr

| logical\_expr AND relational\_expr {

printf("AND operation\n");

}

| logical\_expr OR relational\_expr {

printf("OR operation\n");

}

;

relational\_expr:

primary

| relational\_expr EQ primary {

print\_indent();

printf("EQUALS comparison\n");

}

| relational\_expr NEQ primary {

print\_indent();

printf("NOT EQUALS comparison\n");

}

| relational\_expr LT primary {

print\_indent();

printf("LESS THAN comparison\n");

}

| relational\_expr GT primary {

print\_indent();

printf("GREATER THAN comparison\n");

}

| relational\_expr LE primary {

print\_indent();

printf("LESS THAN OR EQUAL comparison\n");

}

| relational\_expr GE primary {

print\_indent();

printf("GREATER THAN OR EQUAL comparison\n");

}

;

primary:

IDENTIFIER {

print\_indent();

printf("IDENTIFIER\n");

}

| NUMBER {

print\_indent();

printf("NUMBER\n");

}

| LPAREN expression RPAREN

| NOT primary {

print\_indent();

printf("NOT operation\n");

}

;

%%

void yyerror(char \*s) {

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

}

int main() {

printf("Enter statements (example: if (x == 5) { x = 10; } else { x = 20; }):\n");

yyparse();

return 0;

}

**13. Calculate using lex and yacc that support arithmetic operations and variable operations:**

Lex file:

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "y.tab.h"

extern void yyerror(const char \*s);

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

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

if (!p) {

yyerror("Out of memory");

exit(1);

}

strcpy(p, s);

return p;

}

%}

%option noyywrap

%%

[0-9]+(\.[0-9]+)? { yylval.dval = atof(yytext); return NUMBER; }

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

yylval.sval = makestr(yytext);

if (strcasecmp(yytext, "print") == 0) return PRINT;

if (strcasecmp(yytext, "clear") == 0) return CLEAR;

if (strcasecmp(yytext, "quit") == 0 || strcasecmp(yytext, "exit") == 0) return QUIT;

return VARIABLE;

}

"=" { return ASSIGN; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return TIMES; }

"/" { return DIVIDE; }

"^" { return POWER; }

"(" { return LPAREN; }

")" { return RPAREN; }

";" { return SEMICOLON; }

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

\n { return NEWLINE; }

. { yyerror("Invalid character"); }

%%

Yacc file:  
%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

void yyerror(const char \*s);

int yylex();

/\* Symbol Table \*/

#define MAX\_VARS 100

struct symbol {

char \*name;

double value;

} symbol\_table[MAX\_VARS];

int symbol\_count = 0;

int get\_symbol\_index(const char \*name) {

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

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

}

return -1;

}

double get\_symbol\_value(const char \*name) {

int i = get\_symbol\_index(name);

if (i >= 0) return symbol\_table[i].value;

if (symbol\_count >= MAX\_VARS) {

yyerror("Symbol table overflow");

return 0.0;

}

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

symbol\_table[symbol\_count].value = 0.0;

return symbol\_table[symbol\_count++].value;

}

void set\_symbol\_value(const char \*name, double val) {

int i = get\_symbol\_index(name);

if (i >= 0) {

symbol\_table[i].value = val;

} else {

if (symbol\_count >= MAX\_VARS) {

yyerror("Symbol table overflow");

return;

}

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

symbol\_table[symbol\_count].value = val;

symbol\_count++;

}

}

void print\_symbols() {

printf("Variables:\n");

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

printf(" %s = %g\n", symbol\_table[i].name, symbol\_table[i].value);

}

}

void clear\_symbols() {

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

free(symbol\_table[i].name);

}

symbol\_count = 0;

printf("All variables cleared.\n");

}

void cleanup() {

clear\_symbols();

}

%}

%union {

double dval;

char \*sval;

}

%token <dval> NUMBER

%token <sval> VARIABLE

%token PLUS MINUS TIMES DIVIDE POWER

%token LPAREN RPAREN

%token ASSIGN SEMICOLON PRINT CLEAR QUIT NEWLINE

%type <dval> expr term factor

%right POWER

%left PLUS MINUS

%left TIMES DIVIDE

%left UMINUS

%%

input:

/\* empty \*/

| input line

;

line:

NEWLINE

| expr NEWLINE { printf("Result: %g\n", $1); }

| VARIABLE ASSIGN expr NEWLINE {

set\_symbol\_value($1, $3);

printf("%s = %g\n", $1, $3);

free($1);

}

| PRINT NEWLINE { print\_symbols(); }

| CLEAR NEWLINE { clear\_symbols(); }

| QUIT NEWLINE { printf("Goodbye!\n"); cleanup(); exit(0); }

| error NEWLINE { yyerror("Syntax error"); yyerrok; }

;

expr:

term { $$ = $1; }

| expr PLUS term { $$ = $1 + $3; }

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

;

term:

factor { $$ = $1; }

| term TIMES factor { $$ = $1 \* $3; }

| term DIVIDE factor {

if ($3 == 0) {

yyerror("Division by zero");

$$ = 0;

} else {

$$ = $1 / $3;

}

}

;

factor:

NUMBER { $$ = $1; }

| VARIABLE { $$ = get\_symbol\_value($1); free($1); }

| MINUS factor %prec UMINUS { $$ = -$2; }

| factor POWER factor { $$ = pow($1, $3); }

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

;

%%

void yyerror(const char \*s) {

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

}

int main() {

atexit(cleanup);

printf("Calculator with Variables\n");

printf("Type expressions. Use 'print' to show variables, 'clear' to reset, 'quit' to exit.\n");

return yyparse();

}

**14. YACC specification to check the syntax of while statements in C with error detection:**

Lex file:

%{

#include <stdio.h>

#include "y.tab.h"

%}

%%

"while" { return WHILE; }

"if" { return IF; }

"else" { return ELSE; }

"int" { return INT; }

"float" { return FLOAT; }

"char" { return CHAR; }

"return" { return RETURN; }

"(" { return LPAREN; }

")" { return RPAREN; }

"{" { return LBRACE; }

"}" { return RBRACE; }

";" { return SEMICOLON; }

"," { return COMMA; }

"=" { return ASSIGN; }

"==" { return EQ; }

"!=" { return NEQ; }

"<" { return LT; }

">" { return GT; }

"<=" { return LE; }

">=" { return GE; }

"&&" { return AND; }

"||" { return OR; }

"!" { return NOT; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return TIMES; }

"/" { return DIVIDE; }

"break" { return BREAK; }

"continue" { return CONTINUE; }

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

[0-9]+(\.[0-9]+)? { return NUMBER; }

\"[^\"]\*\" { return STRING; }

\'[^\']\*\' { return CHAR\_LITERAL; }

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

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

Yacc file:

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

int yylex();

int line\_num = 1;

%}

%token WHILE IF ELSE INT FLOAT CHAR RETURN

%token LPAREN RPAREN LBRACE RBRACE SEMICOLON COMMA ASSIGN

%token EQ NEQ LT GT LE GE AND OR NOT

%token PLUS MINUS TIMES DIVIDE

%token BREAK CONTINUE

%token IDENTIFIER NUMBER STRING CHAR\_LITERAL

%nonassoc LOWER\_THAN\_ELSE

%nonassoc ELSE

%%

program:

statement\_list

;

statement\_list:

statement

| statement\_list statement

;

statement:

simple\_statement SEMICOLON { printf("Valid statement\n"); }

| while\_statement { printf("Valid while statement\n"); }

| if\_statement { printf("Valid if statement\n"); }

| block\_statement { printf("Valid block statement\n"); }

| error SEMICOLON { yyerrok; printf("Error recovered at ';'\n"); }

;

while\_statement:

WHILE LPAREN expression RPAREN statement {

printf("WHILE statement with condition and body\n");

}

| WHILE error statement {

yyerrok;

printf("Error in WHILE condition\n");

}

| WHILE LPAREN expression RPAREN error {

yyerrok;

printf("Error in WHILE body\n");

}

;

if\_statement:

IF LPAREN expression RPAREN statement %prec LOWER\_THAN\_ELSE {

printf("IF statement with condition and body\n");

}

| IF LPAREN expression RPAREN statement ELSE statement {

printf("IF-ELSE statement with condition and bodies\n");

}

;

block\_statement:

LBRACE statement\_list RBRACE {

printf("Block statement with statements\n");

}

| LBRACE RBRACE {

printf("Empty block statement\n");

}

;

simple\_statement:

assignment

| variable\_declaration

| BREAK { printf("BREAK statement\n"); }

| CONTINUE { printf("CONTINUE statement\n"); }

| RETURN expression { printf("RETURN statement with expression\n"); }

| RETURN { printf("RETURN statement\n"); }

;

variable\_declaration:

type IDENTIFIER {

printf("Variable declaration\n");

}

| type IDENTIFIER ASSIGN expression {

printf("Variable declaration with initialization\n");

}

| type IDENTIFIER LBRACE expression RBRACE {

printf("Array declaration\n");

}

;

assignment:

IDENTIFIER ASSIGN expression {

printf("Assignment\n");

}

;

type:

INT { printf("Integer type\n"); }

| FLOAT { printf("Float type\n"); }

| CHAR { printf("Character type\n"); }

;

expression:

logical\_expr

;

logical\_expr:

relational\_expr

| logical\_expr AND relational\_expr {

printf("AND operation\n");

}

| logical\_expr OR relational\_expr {

printf("OR operation\n");

}

;

relational\_expr:

arithmetic\_expr

| relational\_expr EQ arithmetic\_expr {

printf("EQUALS comparison\n");

}

| relational\_expr NEQ arithmetic\_expr {

printf("NOT EQUALS comparison\n");

}

| relational\_expr LT arithmetic\_expr {

printf("LESS THAN comparison\n");

}

| relational\_expr GT arithmetic\_expr {

printf("GREATER THAN comparison\n");

}

| relational\_expr LE arithmetic\_expr {

printf("LESS THAN OR EQUAL comparison\n");

}

| relational\_expr GE arithmetic\_expr {

printf("GREATER THAN OR EQUAL comparison\n");

}

;

arithmetic\_expr:

term

| arithmetic\_expr PLUS term {

printf("Addition\n");

}

| arithmetic\_expr MINUS term {

printf("Subtraction\n");

}

;

term:

factor

| term TIMES factor {

printf("Multiplication\n");

}

| term DIVIDE factor {

printf("Division\n");

}

;

factor:

IDENTIFIER {

printf("Identifier reference\n");

}

| NUMBER {

printf("Number literal\n");

}

| STRING {

printf("String literal\n");

}

| CHAR\_LITERAL {

printf("Character literal\n");

}

| LPAREN expression RPAREN

| NOT factor {

printf("NOT operation\n");

}

| MINUS factor {

printf("Negation\n");

}

;

%%

void yyerror(const char \*s) {

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

}

int main() {

printf("Enter C code (e.g., if (x < 5) { x = x + 1; } else { x = 0; })\n");

return yyparse();

}

**15. create a LEX and YACC program to parse basic SQL statements:**

Lex file:

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "y.tab.h"

extern void yyerror(const char \*s);

/\* For handling strings \*/

char \*makestr(char \*s) {

char \*p;

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

if (p == NULL) {

yyerror("out of memory");

exit(1);

}

strcpy(p, s);

return p;

}

%}

%option case-insensitive

%%

SELECT { return SELECT; }

FROM { return FROM; }

WHERE { return WHERE; }

AND { return AND; }

OR { return OR; }

NOT { return NOT; }

IN { return IN; }

BETWEEN { return BETWEEN; }

LIKE { return LIKE; }

IS { return IS; }

NULL { return NULLVAL; }

ORDER { return ORDER; }

BY { return BY; }

ASC { return ASC; }

DESC { return DESC; }

INSERT { return INSERT; }

INTO { return INTO; }

VALUES { return VALUES; }

UPDATE { return UPDATE; }

SET { return SET; }

DELETE { return DELETE; }

CREATE { return CREATE; }

TABLE { return TABLE; }

DROP { return DROP; }

ALTER { return ALTER; }

ADD { return ADD; }

INT|INTEGER { return INT; }

VARCHAR { return VARCHAR; }

TEXT { return TEXT; }

DATE { return DATE; }

"(" { return LPAREN; }

")" { return RPAREN; }

";" { return SEMICOLON; }

"," { return COMMA; }

"=" { return EQ; }

"!=" { return NEQ; }

"<" { return LT; }

">" { return GT; }

"<=" { return LE; }

">=" { return GE; }

"\*" { return STAR; }

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

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

\'[^\']\*\' {

yytext[yyleng-1] = '\0'; /\* Remove trailing quote \*/

yylval.sval = makestr(yytext+1); /\* Skip leading quote \*/

return STRING;

}

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

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

Yacc file:

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void yyerror(const char \*s);

int yylex();

/\* For printing indentation \*/

int indent = 0;

void print\_indent() {

for (int i = 0; i < indent; i++) printf(" ");

}

%}

%union {

int ival;

char \*sval;

}

/\* Tokens \*/

%token SELECT FROM WHERE AND OR NOT IN BETWEEN LIKE IS NULLVAL

%token ORDER BY ASC DESC INSERT INTO VALUES UPDATE SET DELETE

%token CREATE TABLE DROP ALTER ADD INT VARCHAR TEXT DATE

%token LPAREN RPAREN SEMICOLON COMMA EQ NEQ LT GT LE GE STAR

%token <ival> NUMBER

%token <sval> IDENTIFIER STRING

/\* Operator precedence and associativity \*/

%left OR

%left AND

%right NOT

%nonassoc EQ NEQ LT GT LE GE IN LIKE BETWEEN IS

%%

statements:

statement

| statements statement

;

statement:

select\_statement SEMICOLON { printf("SELECT statement executed\n"); }

| insert\_statement SEMICOLON { printf("INSERT statement executed\n"); }

| update\_statement SEMICOLON { printf("UPDATE statement executed\n"); }

| delete\_statement SEMICOLON { printf("DELETE statement executed\n"); }

| create\_statement SEMICOLON { printf("CREATE statement executed\n"); }

| drop\_statement SEMICOLON { printf("DROP statement executed\n"); }

| alter\_statement SEMICOLON { printf("ALTER statement executed\n"); }

| error SEMICOLON { yyerrok; printf("Error in SQL statement, recovered at ';'\n"); }

;

select\_statement:

SELECT select\_expr FROM from\_clause opt\_where\_clause opt\_orderby\_clause {

print\_indent(); printf("Complete SELECT statement\n");

}

;

select\_expr:

STAR { print\_indent(); printf("Selected all columns\n"); }

| column\_list { print\_indent(); printf("Selected specific columns\n"); }

;

column\_list:

IDENTIFIER { print\_indent(); printf("Column: %s\n", $1); }

| column\_list COMMA IDENTIFIER { print\_indent(); printf("Column: %s\n", $3); }

;

from\_clause:

IDENTIFIER { print\_indent(); printf("From table: %s\n", $1); }

| from\_clause COMMA IDENTIFIER { print\_indent(); printf("Additional table: %s\n", $3); }

;

opt\_where\_clause:

/\* empty \*/

| WHERE condition { print\_indent(); printf("WHERE clause present\n"); }

;

condition:

IDENTIFIER EQ value { print\_indent(); printf("Condition: %s = [value]\n", $1); }

| IDENTIFIER NEQ value { print\_indent(); printf("Condition: %s != [value]\n", $1); }

| IDENTIFIER LT value { print\_indent(); printf("Condition: %s < [value]\n", $1); }

| IDENTIFIER GT value { print\_indent(); printf("Condition: %s > [value]\n", $1); }

| IDENTIFIER LE value { print\_indent(); printf("Condition: %s <= [value]\n", $1); }

| IDENTIFIER GE value { print\_indent(); printf("Condition: %s >= [value]\n", $1); }

| IDENTIFIER LIKE STRING { print\_indent(); printf("Condition: %s LIKE %s\n", $1, $3); }

| IDENTIFIER IS NULLVAL { print\_indent(); printf("Condition: %s IS NULL\n", $1); }

| IDENTIFIER IS NOT NULLVAL { print\_indent(); printf("Condition: %s IS NOT NULL\n", $1); }

| IDENTIFIER IN LPAREN value\_list RPAREN { print\_indent(); printf("Condition: %s IN (...)\n", $1); }

| IDENTIFIER BETWEEN value AND value { print\_indent(); printf("Condition: %s BETWEEN [...] AND [...]\n", $1); }

| condition AND condition { print\_indent(); printf("AND condition\n"); }

| condition OR condition { print\_indent(); printf("OR condition\n"); }

| NOT condition { print\_indent(); printf("NOT condition\n"); }

| LPAREN condition RPAREN { print\_indent(); printf("Parenthesized condition\n"); }

;

value:

NUMBER { print\_indent(); printf("Numeric value: %d\n", $1); }

| STRING { print\_indent(); printf("String value: %s\n", $1); }

| IDENTIFIER { print\_indent(); printf("Identifier value: %s\n", $1); }

| NULLVAL { print\_indent(); printf("NULL value\n"); }

;

value\_list:

value

| value\_list COMMA value

;

opt\_orderby\_clause:

/\* empty \*/

| ORDER BY order\_list { print\_indent(); printf("ORDER BY clause present\n"); }

;

order\_list:

order\_item

| order\_list COMMA order\_item

;

order\_item:

IDENTIFIER { print\_indent(); printf("Order by: %s (default)\n", $1); }

| IDENTIFIER ASC { print\_indent(); printf("Order by: %s ASC\n", $1); }

| IDENTIFIER DESC { print\_indent(); printf("Order by: %s DESC\n", $1); }

;

insert\_statement:

INSERT INTO IDENTIFIER LPAREN column\_list RPAREN VALUES LPAREN value\_list RPAREN {

print\_indent(); printf("INSERT INTO %s with specified columns\n", $3);

}

| INSERT INTO IDENTIFIER VALUES LPAREN value\_list RPAREN {

print\_indent(); printf("INSERT INTO %s with values\n", $3);

}

;

update\_statement:

UPDATE IDENTIFIER SET update\_list opt\_where\_clause {

print\_indent(); printf("UPDATE on table: %s\n", $2);

}

;

update\_list:

update\_item

| update\_list COMMA update\_item

;

update\_item:

IDENTIFIER EQ value {

print\_indent(); printf("Update column: %s\n", $1);

}

;

delete\_statement:

DELETE FROM IDENTIFIER opt\_where\_clause {

print\_indent(); printf("DELETE from table: %s\n", $3);

}

;

create\_statement:

CREATE TABLE IDENTIFIER LPAREN create\_column\_list RPAREN {

print\_indent(); printf("CREATE TABLE: %s\n", $3);

}

;

create\_column\_list:

create\_column\_def

| create\_column\_list COMMA create\_column\_def

;

create\_column\_def:

IDENTIFIER column\_type {

print\_indent(); printf("Column definition: %s\n", $1);

}

;

column\_type:

INT { print\_indent(); printf("Type: INT\n"); }

| VARCHAR LPAREN NUMBER RPAREN { print\_indent(); printf("Type: VARCHAR(%d)\n", $3); }

| TEXT { print\_indent(); printf("Type: TEXT\n"); }

| DATE { print\_indent(); printf("Type: DATE\n"); }

;

drop\_statement:

DROP TABLE IDENTIFIER {

print\_indent(); printf("DROP TABLE: %s\n", $3);

}

;

alter\_statement:

ALTER TABLE IDENTIFIER ADD IDENTIFIER column\_type {

print\_indent(); printf("ALTER TABLE %s ADD COLUMN %s\n", $3, $5);

}

;

%%

void yyerror(const char \*s) {

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

}

int main() {

printf("SQL Parser Ready.\n");

printf("Enter SQL statements (end with semicolon):\n");

return yyparse();

}

**16. Implement Common Sub-expression Elimination optimization on input quadruples in C/C++/ Java**

#include <iostream>

#include <vector>

#include <map>

#include <tuple>

#include <string>

using namespace std;

// Struct to represent a quadruple

struct Quadruple {

string op, arg1, arg2, result;

Quadruple(string o, string a1, string a2, string r)

: op(o), arg1(a1), arg2(a2), result(r) {}

string str() const {

return "(" + op + ", " + arg1 + ", " + arg2 + ", " + result + ")";

}

};

// Apply Common Subexpression Elimination on input quadruples

vector<Quadruple> eliminate\_common\_subexpressions(const vector<Quadruple>& input) {

map<tuple<string, string, string>, string> expr\_map;

vector<Quadruple> optimized;

vector<tuple<string, string, string>> cse\_applied;

cout << "\n===== Optimization Method Used =====\n" << endl;

for (const auto& q : input) {

auto key = make\_tuple(q.op, q.arg1, q.arg2);

if (q.op == "+" || q.op == "-" || q.op == "\*" || q.op == "/") {

if (expr\_map.find(key) != expr\_map.end()) {

string existing = expr\_map[key];

cout << "Reusing common subexpression " << q.arg1 << " " << q.op << " " << q.arg2 << " -> " << existing << endl;

optimized.emplace\_back("=", existing, "-", q.result);

cse\_applied.push\_back(key);

} else {

expr\_map[key] = q.result;

optimized.push\_back(q);

}

} else {

optimized.push\_back(q);

}

}

if (!cse\_applied.empty()) {

cout << "\nCommon Subexpression Elimination (CSE) applied to:\n" << endl;

for (const auto& expr : cse\_applied) {

cout << " " << get<1>(expr) << " " << get<0>(expr) << " " << get<2>(expr) << endl;

}

} else {

cout << "No common subexpressions found. No CSE applied." << endl;

}

return optimized;

}

int main() {

int n;

cout << "Enter number of quadruples: ";

cin >> n;

vector<Quadruple> quads;

cout << "Enter each quadruple in format: op arg1 arg2 result" << endl;

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

string op, arg1, arg2, result;

cin >> op >> arg1 >> arg2 >> result;

quads.emplace\_back(op, arg1, arg2, result);

}

vector<Quadruple> optimized = eliminate\_common\_subexpressions(quads);

cout << "\n--- Optimized Quadruples with CSE ---\n" << endl;

for (size\_t i = 0; i < optimized.size(); ++i) {

cout << i << ": " << optimized[i].str() << endl;

}

cout << "\n--------------------------------------\n";

return 0;

}

Enter number of quadruples: 4

\* b c t1

\* b c t2

+ t1 t2 t3

= t3 - a

**17. Constant folding optimization**

/\*\*

\* Constant Folding Optimization on Input Quadruples

\* This optimization identifies arithmetic operations with constant operands

\* and pre-computes their results at compile time.

\*/  
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <math.h>

#define MAX\_QUADRUPLES 100

#define MAX\_IDENTIFIER 20

#define MAX\_CONSTANTS 50

typedef struct {

char op[5];

char arg1[MAX\_IDENTIFIER];

char arg2[MAX\_IDENTIFIER];

char result[MAX\_IDENTIFIER];

} Quadruple;

typedef struct {

char name[MAX\_IDENTIFIER];

double value;

} Constant;

Quadruple quadruples[MAX\_QUADRUPLES];

int quadCount = 0;

Constant constants[MAX\_CONSTANTS];

int constCount = 0;

void addQuadruple(const char\* op, const char\* arg1, const char\* arg2, const char\* result);

int isConstant(const char\* value);

double getValue(const char\* value);

void applyConstantFolding();

void displayQuadruples();

void setConstant(const char\* name, double value);

int main() {

int n;

printf("Enter number of quadruples: ");

scanf("%d", &n);

getchar(); // to consume newline after number input

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

char op[5], arg1[MAX\_IDENTIFIER], arg2[MAX\_IDENTIFIER], result[MAX\_IDENTIFIER];

printf("\nQuadruple %d\n", i + 1);

printf("Operator (+, -, \*, /, =): ");

scanf("%s", op);

printf("Argument 1: ");

scanf("%s", arg1);

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

arg2[0] = '\0'; // no second argument

} else {

printf("Argument 2: ");

scanf("%s", arg2);

}

printf("Result variable: ");

scanf("%s", result);

addQuadruple(op, arg1, (strcmp(op, "=") == 0 ? NULL : arg2), result);

}

printf("\nOriginal quadruples:\n");

displayQuadruples();

applyConstantFolding();

printf("\nAfter constant folding:\n");

displayQuadruples();

return 0;

}

void addQuadruple(const char\* op, const char\* arg1, const char\* arg2, const char\* result) {

if (quadCount >= MAX\_QUADRUPLES) {

printf("Error: Maximum number of quadruples reached.\n");

return;

}

strcpy(quadruples[quadCount].op, op);

strcpy(quadruples[quadCount].arg1, arg1);

if (arg2 != NULL) {

strcpy(quadruples[quadCount].arg2, arg2);

} else {

quadruples[quadCount].arg2[0] = '\0';

}

strcpy(quadruples[quadCount].result, result);

quadCount++;

}

int isConstant(const char\* value) {

if (value == NULL || value[0] == '\0') return 0;

char\* endptr;

strtod(value, &endptr);

if (\*endptr == '\0') return 1;

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

if (strcmp(constants[i].name, value) == 0) return 1;

}

return 0;

}

double getValue(const char\* value) {

if (value == NULL || value[0] == '\0') return 0;

char\* endptr;

double numValue = strtod(value, &endptr);

if (\*endptr == '\0') return numValue;

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

if (strcmp(constants[i].name, value) == 0) {

return constants[i].value;

}

}

return 0;

}

void setConstant(const char\* name, double value) {

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

if (strcmp(constants[i].name, name) == 0) {

constants[i].value = value;

return;

}

}

if (constCount < MAX\_CONSTANTS) {

strcpy(constants[constCount].name, name);

constants[constCount].value = value;

constCount++;

} else {

printf("Error: Maximum number of constants reached.\n");

}

}

void applyConstantFolding() {

int i, j = 0;

Quadruple foldedQuadruples[MAX\_QUADRUPLES];

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

Quadruple q = quadruples[i];

if (strcmp(q.op, "=") == 0 && q.arg2[0] == '\0' && isConstant(q.arg1)) {

double value = getValue(q.arg1);

setConstant(q.result, value);

}

if ((strcmp(q.op, "+") == 0 || strcmp(q.op, "-") == 0 ||

strcmp(q.op, "\*") == 0 || strcmp(q.op, "/") == 0) &&

isConstant(q.arg1) && isConstant(q.arg2)) {

double val1 = getValue(q.arg1);

double val2 = getValue(q.arg2);

double computedValue = 0;

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

computedValue = val1 + val2;

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

computedValue = val1 - val2;

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

computedValue = val1 \* val2;

} else if (strcmp(q.op, "/") == 0 && val2 != 0) {

computedValue = val1 / val2;

} else {

foldedQuadruples[j++] = q;

continue;

}

strcpy(foldedQuadruples[j].op, "=");

char valueStr[MAX\_IDENTIFIER];

sprintf(valueStr, "%g", computedValue);

strcpy(foldedQuadruples[j].arg1, valueStr);

foldedQuadruples[j].arg2[0] = '\0';

strcpy(foldedQuadruples[j].result, q.result);

setConstant(q.result, computedValue);

j++;

} else {

foldedQuadruples[j++] = q;

}

}

quadCount = j;

for (i = 0; i < j; i++) {

quadruples[i] = foldedQuadruples[i];

}

}

void displayQuadruples() {

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

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

if (quadruples[i].arg2[0] == '\0') {

printf("%d: %s = %s\n", i, quadruples[i].result, quadruples[i].arg1);

} else {

printf("%d: %s = %s %s %s\n", i, quadruples[i].result,

quadruples[i].arg1, quadruples[i].op, quadruples[i].arg2);

}

}

}

Enter number of quadruples: 3

Quadruple 1

Operator (+, -, \*, /, =): =

Argument 1: 4

Result variable: a

Quadruple 2

Operator (+, -, \*, /, =): =

Argument 1: 5

Result variable: b

Quadruple 3

Operator (+, -, \*, /, =): +

Argument 1: a

Argument 2: b

Result variable: c

**18. Constant propagation optimization in c**

/\*\*

\* Constant Propagation Optimization on Input Quadruples

\* This optimization identifies variables with constant values and replaces

\* their occurrences with the actual constants.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_QUADRUPLES 100

#define MAX\_IDENTIFIER 20

#define MAX\_CONSTANTS 50

typedef struct {

char op[5];

char arg1[MAX\_IDENTIFIER];

char arg2[MAX\_IDENTIFIER];

char result[MAX\_IDENTIFIER];

} Quadruple;

typedef struct {

char name[MAX\_IDENTIFIER];

char value[MAX\_IDENTIFIER];

} Constant;

Quadruple quadruples[MAX\_QUADRUPLES];

int quadCount = 0;

Constant constants[MAX\_CONSTANTS];

int constCount = 0;

void addQuadruple(const char\* op, const char\* arg1, const char\* arg2, const char\* result);

int isNumericConstant(const char\* value);

void identifyConstants();

void applyConstantPropagation();

void displayQuadruples();

int getConstantValue(const char\* name, char\* value);

int main() {

int n;

printf("Enter number of quadruples: ");

scanf("%d", &n);

getchar(); // consume newline

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

char op[5], arg1[MAX\_IDENTIFIER], arg2[MAX\_IDENTIFIER], result[MAX\_IDENTIFIER];

printf("Quadruple %d:\n", i + 1);

printf(" Operator (+, -, \*, /, =): ");

scanf("%s", op);

printf(" Argument 1: ");

scanf("%s", arg1);

printf(" Argument 2 (or - for none): ");

scanf("%s", arg2);

printf(" Result variable: ");

scanf("%s", result);

// Handle '-' as NULL

if (strcmp(arg2, "-") == 0) {

addQuadruple(op, arg1, NULL, result);

} else {

addQuadruple(op, arg1, arg2, result);

}

}

printf("\nOriginal quadruples:\n");

displayQuadruples();

applyConstantPropagation();

printf("\nAfter constant propagation:\n");

displayQuadruples();

return 0;

}

void addQuadruple(const char\* op, const char\* arg1, const char\* arg2, const char\* result) {

if (quadCount >= MAX\_QUADRUPLES) {

printf("Error: Maximum number of quadruples reached.\n");

return;

}

strcpy(quadruples[quadCount].op, op);

strcpy(quadruples[quadCount].arg1, arg1);

if (arg2 != NULL) {

strcpy(quadruples[quadCount].arg2, arg2);

} else {

quadruples[quadCount].arg2[0] = '\0';

}

strcpy(quadruples[quadCount].result, result);

quadCount++;

}

int isNumericConstant(const char\* value) {

if (value == NULL || value[0] == '\0') {

return 0;

}

char\* endptr;

strtod(value, &endptr);

return (\*endptr == '\0');

}

int getConstantValue(const char\* name, char\* value) {

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

if (strcmp(constants[i].name, name) == 0) {

strcpy(value, constants[i].value);

return 1;

}

}

return 0;

}

void identifyConstants() {

constCount = 0;

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

if (strcmp(quadruples[i].op, "=") == 0 && quadruples[i].arg2[0] == '\0') {

if (isNumericConstant(quadruples[i].arg1)) {

strcpy(constants[constCount].name, quadruples[i].result);

strcpy(constants[constCount].value, quadruples[i].arg1);

constCount++;

}

}

}

}

void applyConstantPropagation() {

identifyConstants();

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

char value[MAX\_IDENTIFIER];

if (!isNumericConstant(quadruples[i].arg1) && getConstantValue(quadruples[i].arg1, value)) {

strcpy(quadruples[i].arg1, value);

}

if (quadruples[i].arg2[0] != '\0' && !isNumericConstant(quadruples[i].arg2) &&

getConstantValue(quadruples[i].arg2, value)) {

strcpy(quadruples[i].arg2, value);

}

}

}

void displayQuadruples() {

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

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

if (quadruples[i].arg2[0] == '\0') {

printf("%d: %s = %s\n", i, quadruples[i].result, quadruples[i].arg1);

} else {

printf("%d: %s = %s %s %s\n", i, quadruples[i].result,

quadruples[i].arg1, quadruples[i].op, quadruples[i].arg2);

}

}

}

Enter number of quadruples: 3

Quadruple 1:

Operator (+, -, \*, /, =): =

Argument 1: 5

Argument 2 (or - for none): -

Result variable: a

Quadruple 2:

Operator (+, -, \*, /, =): =

Argument 1: 10

Argument 2 (or - for none): -

Result variable: b

Quadruple 3:

Operator (+, -, \*, /, =): +

Argument 1: a

Argument 2 (or - for none): b

Result variable: c

**19. Copy propagation optimization**

/\*\*

\* Copy Propagation Optimization on Input Quadruples

\* This optimization identifies copy assignments (a = b) and replaces uses of 'a' with 'b'

\* in subsequent code, as long as neither 'a' nor 'b' are redefined.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_QUADRUPLES 100

#define MAX\_IDENTIFIER 20

#define MAX\_COPIES 50

typedef struct {

char op[5];

char arg1[MAX\_IDENTIFIER];

char arg2[MAX\_IDENTIFIER];

char result[MAX\_IDENTIFIER];

} Quadruple;

typedef struct {

char target[MAX\_IDENTIFIER];

char source[MAX\_IDENTIFIER];

int valid;

} CopyRelation;

Quadruple quadruples[MAX\_QUADRUPLES];

int quadCount = 0;

CopyRelation copyRelations[MAX\_COPIES];

int copyCount = 0;

void addQuadruple(const char\* op, const char\* arg1, const char\* arg2, const char\* result);

int isNumericConstant(const char\* value);

void identifyCopyAssignments();

void applyCopyPropagation();

void displayQuadruples();

void getUserInput();

int main() {

getUserInput();

printf("\nOriginal quadruples:\n");

displayQuadruples();

applyCopyPropagation();

printf("\nAfter copy propagation:\n");

displayQuadruples();

return 0;

}

void addQuadruple(const char\* op, const char\* arg1, const char\* arg2, const char\* result) {

if (quadCount >= MAX\_QUADRUPLES) {

printf("Error: Maximum number of quadruples reached.\n");

return;

}

strcpy(quadruples[quadCount].op, op);

strcpy(quadruples[quadCount].arg1, arg1);

if (arg2 != NULL) {

strcpy(quadruples[quadCount].arg2, arg2);

} else {

quadruples[quadCount].arg2[0] = '\0';

}

strcpy(quadruples[quadCount].result, result);

quadCount++;

}

int isNumericConstant(const char\* value) {

if (value == NULL || value[0] == '\0') {

return 0;

}

char\* endptr;

strtod(value, &endptr);

return (\*endptr == '\0');

}

int getCopySource(const char\* target, char\* source) {

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

if (strcmp(copyRelations[i].target, target) == 0 && copyRelations[i].valid) {

strcpy(source, copyRelations[i].source);

return 1;

}

}

return 0;

}

void invalidateCopyRelations(const char\* var) {

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

if (strcmp(copyRelations[i].target, var) == 0 ||

strcmp(copyRelations[i].source, var) == 0) {

copyRelations[i].valid = 0;

}

}

}

void identifyCopyAssignments() {

copyCount = 0;

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

if (strcmp(quadruples[i].op, "=") == 0 &&

quadruples[i].arg2[0] == '\0' &&

!isNumericConstant(quadruples[i].arg1)) {

char source[MAX\_IDENTIFIER];

strcpy(source, quadruples[i].arg1);

char target[MAX\_IDENTIFIER];

strcpy(target, quadruples[i].result);

strcpy(copyRelations[copyCount].source, source);

strcpy(copyRelations[copyCount].target, target);

copyRelations[copyCount].valid = 1;

copyCount++;

invalidateCopyRelations(target);

} else if (quadruples[i].result[0] != '\0') {

invalidateCopyRelations(quadruples[i].result);

}

}

}

void applyCopyPropagation() {

identifyCopyAssignments();

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

char source[MAX\_IDENTIFIER];

if (!isNumericConstant(quadruples[i].arg1) &&

getCopySource(quadruples[i].arg1, source)) {

strcpy(quadruples[i].arg1, source);

}

if (quadruples[i].arg2[0] != '\0' &&

!isNumericConstant(quadruples[i].arg2) &&

getCopySource(quadruples[i].arg2, source)) {

strcpy(quadruples[i].arg2, source);

}

}

}

void displayQuadruples() {

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

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

if (quadruples[i].arg2[0] == '\0') {

printf("%d: %s = %s\n", i, quadruples[i].result, quadruples[i].arg1);

} else {

printf("%d: %s = %s %s %s\n", i, quadruples[i].result,

quadruples[i].arg1, quadruples[i].op, quadruples[i].arg2);

}

}

}

void getUserInput() {

int n;

char op[5], arg1[MAX\_IDENTIFIER], arg2[MAX\_IDENTIFIER], result[MAX\_IDENTIFIER];

printf("Enter number of quadruples: ");

scanf("%d", &n);

getchar(); // Clear newline

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

printf("Quadruple #%d:\n", i + 1);

printf("Operator (=, +, -, \*, /): ");

scanf("%s", op);

printf("Argument 1: ");

scanf("%s", arg1);

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

printf("Argument 2: ");

scanf("%s", arg2);

} else {

arg2[0] = '\0';

}

printf("Result variable: ");

scanf("%s", result);

addQuadruple(op, arg1, (arg2[0] == '\0' ? NULL : arg2), result);

printf("\n");

}

}

Enter number of quadruples: 5

Quadruple #1:

Operator (=, +, -, \*, /): =

Argument 1: x

Result variable: a

Quadruple #2:

Operator (=, +, -, \*, /): =

Argument 1: y

Result variable: b

Quadruple #3:

Operator (=, +, -, \*, /): +

Argument 1: a

Argument 2: b

Result variable: c

Quadruple #4:

Operator (=, +, -, \*, /): =

Argument 1: c

Result variable: d

Quadruple #5:

Operator (=, +, -, \*, /): \*

Argument 1: d

Argument 2: a

Result variable: e

**20. dead code elimination**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_QUADRUPLES 100

#define MAX\_IDENTIFIER 20

#define MAX\_VARIABLES 50

typedef struct {

char op[10];

char arg1[MAX\_IDENTIFIER];

char arg2[MAX\_IDENTIFIER];

char result[MAX\_IDENTIFIER];

int isLive;

} Quadruple;

Quadruple quadruples[MAX\_QUADRUPLES];

int quadCount = 0;

char liveVariables[MAX\_VARIABLES][MAX\_IDENTIFIER];

int liveCount = 0;

// Function declarations

void addQuadruple(const char\* op, const char\* arg1, const char\* arg2, const char\* result);

int isNumericConstant(const char\* value);

void addLiveVariable(const char\* var);

int isLiveVariable(const char\* var);

void applyDeadCodeElimination();

void displayQuadruples();

int main() {

int n;

char op[10], arg1[MAX\_IDENTIFIER], arg2[MAX\_IDENTIFIER], result[MAX\_IDENTIFIER];

printf("Enter number of quadruples: ");

scanf("%d", &n);

getchar();

printf("Enter each quadruple in the format: <op> <arg1> <arg2 or -> <result or ->>\n");

printf("Use '-' if arg2 or result is NULL. Example: + a b c or output e - -\n");

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

scanf("%s %s %s %s", op, arg1, arg2, result);

addQuadruple(op,

strcmp(arg1, "-") == 0 ? "" : arg1,

strcmp(arg2, "-") == 0 ? "" : arg2,

strcmp(result, "-") == 0 ? "" : result);

}

printf("\nOriginal quadruples:\n");

displayQuadruples();

applyDeadCodeElimination();

printf("\nAfter dead code elimination:\n");

displayQuadruples();

return 0;

}

void addQuadruple(const char\* op, const char\* arg1, const char\* arg2, const char\* result) {

if (quadCount >= MAX\_QUADRUPLES) return;

strcpy(quadruples[quadCount].op, op);

strcpy(quadruples[quadCount].arg1, arg1);

strcpy(quadruples[quadCount].arg2, arg2);

strcpy(quadruples[quadCount].result, result);

quadruples[quadCount].isLive = 0; // Initially assume dead

quadCount++;

}

int isNumericConstant(const char\* value) {

if (value == NULL || value[0] == '\0') return 0;

char\* endptr;

strtod(value, &endptr);

return (\*endptr == '\0');

}

void addLiveVariable(const char\* var) {

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

if (strcmp(liveVariables[i], var) == 0) return;

}

if (liveCount < MAX\_VARIABLES) {

strcpy(liveVariables[liveCount++], var);

}

}

int isLiveVariable(const char\* var) {

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

if (strcmp(liveVariables[i], var) == 0) return 1;

}

return 0;

}

void applyDeadCodeElimination() {

liveCount = 0;

// Step 1: Add variables used in output statements

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

if (strcmp(quadruples[i].op, "output") == 0) {

if (quadruples[i].arg1[0] && !isNumericConstant(quadruples[i].arg1)) {

addLiveVariable(quadruples[i].arg1);

quadruples[i].isLive = 1;

}

}

}

// Step 2: Backward traversal to identify all live code

int changed = 1;

while (changed) {

changed = 0;

for (int i = quadCount - 1; i >= 0; i--) {

Quadruple\* q = &quadruples[i];

if (q->result[0] && isLiveVariable(q->result) && q->isLive == 0) {

q->isLive = 1;

if (q->arg1[0] && !isNumericConstant(q->arg1)) addLiveVariable(q->arg1);

if (q->arg2[0] && !isNumericConstant(q->arg2)) addLiveVariable(q->arg2);

changed = 1;

}

}

}

// Step 3: Compact the array

int j = 0;

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

if (quadruples[i].isLive) {

if (i != j) quadruples[j] = quadruples[i];

j++;

}

}

quadCount = j;

}

void displayQuadruples() {

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

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

Quadruple\* q = &quadruples[i];

if (strcmp(q->op, "output") == 0) {

printf("%d: output %s\n", i, q->arg1);

} else if (q->arg2[0] == '\0') {

printf("%d: %s = %s\n", i, q->result, q->arg1);

} else {

printf("%d: %s = %s %s %s\n", i, q->result, q->arg1, q->op, q->arg2);

}

}

}

**21. Lex and Yacc Program for Switch-Case Validation**

/\* Lexical analyzer (Lex/Flex) for switch-case statements \*/

/\* switchlex.l \*/

%{

#include <stdio.h>

#include <stdlib.h>

#include "switchparser.tab.h" // ✅ Correct header

%}

%%

"switch" { return SWITCH; }

"case" { return CASE; }

"break" { return BREAK; }

"default" { return DEFAULT; }

"{" { return LBRACE; }

"}" { return RBRACE; }

":" { return COLON; }

";" { return SEMICOLON; }

"(" { return LPAREN; }

")" { return RPAREN; }

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

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

[ \t\n]+ ; /\* Skip whitespace \*/

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

/\* Parser (Yacc/Bison) for switch-case statements \*/

/\* switchparse.y \*/

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

extern int yylex();

%}

%token SWITCH CASE DEFAULT BREAK LBRACE RBRACE COLON SEMICOLON LPAREN RPAREN

%token NUMBER IDENTIFIER

%%

program:

switch\_statement

;

switch\_statement:

SWITCH LPAREN IDENTIFIER RPAREN LBRACE case\_list RBRACE

{ printf("Valid switch-case statement\n"); }

;

case\_list:

case\_statement

| case\_list case\_statement

| case\_list default\_statement

;

case\_statement:

CASE NUMBER COLON statement\_list

;

default\_statement:

DEFAULT COLON statement\_list

;

statement\_list:

statement

| statement\_list statement

;

statement:

BREAK SEMICOLON

| IDENTIFIER SEMICOLON

;

%%

void yyerror(const char \*s) {

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

}

int main() {

printf("Enter a switch-case statement:\n");

yyparse();

return 0;

}

/\* Example usage:

To compile:

flex switchlex.l

bison -d switchparse.y

gcc -o parser switchparser.tab.c [lex.yy](http://lex.yy).c  
parser.exe

Valid input example:

switch(x) {

case 1: break;

case 2: doSomething; break;

default: break;

}

\*/

**22. Writing a code optimization program that performs strength reduction**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h> // <-- FIX for isalpha, isalnum, isdigit

typedef enum {

TOKEN\_IDENTIFIER,

TOKEN\_NUMBER,

TOKEN\_MULTIPLY,

TOKEN\_ASSIGN,

TOKEN\_SEMICOLON,

TOKEN\_EOF

} TokenType;

typedef struct {

TokenType type;

char value[100];

} Token;

typedef struct {

Token tokens[100];

int count;

} TokenList;

// Simple parser

TokenList parseExpression(const char\* input) {

TokenList list = {0};

char buffer[100];

int i = 0, j = 0;

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

if (isalpha(input[i])) {

j = 0;

while (isalnum(input[i])) {

buffer[j++] = input[i++];

}

buffer[j] = '\0';

list.tokens[list.count].type = TOKEN\_IDENTIFIER;

strcpy(list.tokens[list.count].value, buffer);

list.count++;

} else if (isdigit(input[i])) {

j = 0;

while (isdigit(input[i])) {

buffer[j++] = input[i++];

}

buffer[j] = '\0';

list.tokens[list.count].type = TOKEN\_NUMBER;

strcpy(list.tokens[list.count].value, buffer);

list.count++;

} else if (input[i] == '\*') {

list.tokens[list.count].type = TOKEN\_MULTIPLY;

strcpy(list.tokens[list.count].value, "\*");

list.count++;

i++;

} else if (input[i] == '=') {

list.tokens[list.count].type = TOKEN\_ASSIGN;

strcpy(list.tokens[list.count].value, "=");

list.count++;

i++;

} else if (input[i] == ';') {

list.tokens[list.count].type = TOKEN\_SEMICOLON;

strcpy(list.tokens[list.count].value, ";");

list.count++;

i++;

} else {

i++; // Skip whitespace

}

}

list.tokens[list.count].type = TOKEN\_EOF;

return list;

}

int isPowerOfTwo(int num) {

return num > 0 && (num & (num - 1)) == 0;

}

// Renamed to avoid conflict with <math.h>

int log2int(int num) {

int result = 0;

while (num >>= 1) {

result++;

}

return result;

}

void optimizeExpression(TokenList\* tokens) {

for (int i = 0; i < tokens->count - 4; i++) {

if (tokens->tokens[i].type == TOKEN\_IDENTIFIER &&

tokens->tokens[i+1].type == TOKEN\_ASSIGN &&

tokens->tokens[i+2].type == TOKEN\_IDENTIFIER &&

tokens->tokens[i+3].type == TOKEN\_MULTIPLY &&

tokens->tokens[i+4].type == TOKEN\_NUMBER) {

int num = atoi(tokens->tokens[i+4].value);

if (isPowerOfTwo(num)) {

int shift = log2int(num);

printf("Optimized: %s = %s << %d; (instead of %s = %s \* %s;)\n",

tokens->tokens[i].value,

tokens->tokens[i+2].value,

shift,

tokens->tokens[i].value,

tokens->tokens[i+2].value,

tokens->tokens[i+4].value);

} else {

printf("No optimization possible for: %s = %s \* %s;\n",

tokens->tokens[i].value,

tokens->tokens[i+2].value,

tokens->tokens[i+4].value);

}

}

}

}

int main() {

char input[100];

printf("Enter an expression to optimize (e.g., 'a = b \* 8;'): ");

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

TokenList tokens = parseExpression(input);

optimizeExpression(&tokens);

printf("\nMore examples of strength reduction:\n");

printf("1. x \* 4 → x << 2\n");

printf("2. x \* 8 → x << 3\n");

printf("3. x / 4 → x >> 2\n");

printf("4. x \* 5 → (x << 2) + x\n");

return 0;

}