3a valid arithmetic

%{

#include "y.tab.h" // Include the Yacc header for token definitions

#include <stdio.h>

%}

%%

"=" { printf("\nOperator is EQUAL\n"); return EQUAL; }

"+" { printf("\nOperator is PLUS\n"); return PLUS; }

"-" { printf("\nOperator is MINUS\n"); return MINUS; }

"/" { printf("\nOperator is DIVISION\n"); return DIVISION; }

"\*" { printf("\nOperator is MULTIPLICATION\n"); return MULTIPLICATION; }

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

printf("\nIdentifier is %s\n", yytext);

return ID;

}

[0-9]+ {

printf("\nNumber is %s\n", yytext);

return NUM;

}

\n { return 0; } // Handle newlines

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

%%

int yywrap() {

return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

extern int yylex(); // Declare lexer function

extern int yyparse(); // Declare parser function

extern FILE \*yyin; // Input file for lexer

%}

%token ID NUM

%token EQUAL PLUS MINUS DIVISION MULTIPLICATION

%%

program:

statements

;

statements:

statement

| statements statement

;

statement:

ID '=' expr { printf("\nValid assignment: %s = %d\n", $1, $3); }

| expr { printf("\nValid arithmetic expression\n"); }

;

expr:

expr '+' term { $$ = $1 + $3; }

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

| expr '\*' term { $$ = $1 \* $3; }

| expr '/' term { $$ = $1 / $3; }

| term

;

term:

ID { $$ = 1; } // Placeholder for identifier value (you can expand to handle variables)

| NUM { $$ = atoi(yytext); } // Convert string to integer for NUM

;

%%

int main() {

yyin = fopen("input.txt", "r"); // Open the input file

if (!yyin) {

printf("Failed to open input file.\n");

return 1;

}

yyparse(); // Start the parser

fclose(yyin); // Close the input file

return 0;

}

void yyerror(char \*s) {

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

}

3b valid variable

%{

#include "y.tab.h" // Include the header file generated by Bison for token definitions

%}

%%

[a-zA-Z] { return LET; } // Matches a letter (for variable names)

[0-9] { return DIG; } // Matches a digit

. { return yytext[0]; } // Return any other character as is

\n { return 0; } // Return 0 at the end of line (new line)

%%

int yywrap() {

return 1; // Return 1 when done processing input

}

%{

#include <stdio.h>

#include <stdlib.h>

extern int yylex(); // Declare lexer function

extern int yyparse(); // Declare parser function

%}

%token LET DIG // Define tokens for LET and DIG

%%

variable:

var { printf("Valid variable\n"); }

;

var:

LET var\_tail // Variable starts with a letter, followed by var\_tail (which can be letter/digit)

;

var\_tail:

LET var\_tail // Letters followed by more letters/digits

| DIG var\_tail // Digits followed by more letters/digits

| /\* empty \*/ // Empty case allows just a letter without further characters

;

%%

int main() {

printf("Enter the variable:\n");

yyparse(); // Start parsing

return 0;

}

int yyerror(const char \*s) {

printf("Invalid variable\n");

exit(1); // Exit with error if variable is invalid

}

3c control structures

%{

#include "y.tab.h" // Include Bison-generated header for token definitions

%}

alpha [A-Za-z] // Alphabet character

digit [0-9] // Digit character

%%

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

if { return IF; }

then { return THEN; }

else { return ELSE; }

"<=" { return LE; }

">=" { return GE; }

"==" { return EQ; }

"!=" { return NE; }

"||" { return OR; }

"&&" { return AND; }

{digit}+({alpha}|{digit})\* { return ID; } // Variable/identifier

{digit}+ { return NUM; } // Numeric value

. { return yytext[0]; } // Return other characters as they are

%%

int yywrap() {

return 1; // Indicate the end of input

}

%{

#include <stdio.h>

#include <stdlib.h>

extern int yylex(); // Declare lexer function

extern int yyparse(); // Declare parser function

%}

%token ID NUM IF THEN LE GE EQ NE OR AND ELSE

%right '='

%left AND OR

%left '<' '>' LE GE EQ NE

%left '+' '-'

%left '\*' '/'

%right UMINUS

%left '|'

%%

S : ST {

printf("Input Accepted.\n");

exit(0);

};

ST : IF '(' E2 ')' THEN ST1 ELSE ST1 ';'

| IF '(' E2 ')' THEN ST1 ';'

;

ST1 : ST

| E

;

E : ID '=' E { printf("Assignment: %s = %d\n", $1, $3); }

| E '+' E { $$ = $1 + $3; }

| E '-' E { $$ = $1 - $3; }

| E '\*' E { $$ = $1 \* $3; }

| E '/' E { $$ = $1 / $3; }

| E '<' E { $$ = $1 < $3; }

| E '>' E { $$ = $1 > $3; }

| E LE E { $$ = $1 <= $3; }

| E GE E { $$ = $1 >= $3; }

| E EQ E { $$ = $1 == $3; }

| E NE E { $$ = $1 != $3; }

| E OR E { $$ = $1 || $3; }

| E AND E { $$ = $1 && $3; }

| ID { $$ = 1; } // Placeholder for variable value

| NUM { $$ = atoi(yytext); } // Convert NUM to integer

E2 : E '<' E

| E '>' E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

| ID

| NUM

;

%%

int main() {

printf("Enter The Expression: ");

yyparse(); // Start parsing

return 0;

}

int yyerror(const char \*s) {

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

exit(1); // Exit on error

}

5 Type checking

%{

#include <stdio.h>

#include <string.h>

int n; // Number of variables

char vari[15]; // Variable names

char typ[15]; // Variable types (f or i)

int flag = 0; // Flag for checking float type requirement

%}

%%

"int"|"float" {

if (strcmp(yytext, "int") == 0) {

// Handle int type variable

printf("Recognized type: int\n");

} else if (strcmp(yytext, "float") == 0) {

// Handle float type variable

printf("Recognized type: float\n");

}

}

[a-zA-Z] {

// Check if variable is defined

int found = 0;

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

if (vari[i] == yytext[0]) {

found = 1;

if (flag && typ[i] == 'f') {

printf("Variable %c correctly defined as float\n", vari[i]);

} else if (!flag) {

printf("Variable %c correctly defined\n", vari[i]);

} else {

printf("Identifier %c must be a float type\n", vari[i]);

}

break;

}

}

if (!found) {

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

}

}

"/" {

flag = 1; // Set flag for float type if division is encountered

printf("Division operator detected. Expecting float type variables.\n");

}

"+"|"-" {

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

}

[0-9]+ {

// Handle numbers (if needed in the future)

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

}

"=" {

printf("Assignment operator detected\n");

}

"$" {

printf("End of expression detected.\n");

return 0; // End the input parsing

}

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

%%

int main() {

printf("Enter the number of variables: ");

scanf("%d", &n);

getchar(); // Consume the newline character after number input

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

printf("Enter variable[%d]: ", i);

scanf("%c", &vari[i]); // Read the variable name

getchar(); // Consume the newline character

printf("Enter variable-type[%d] (float-f, int-i): ", i);

scanf("%c", &typ[i]); // Read the type (f or i)

getchar(); // Consume the newline character

}

printf("Enter the expression (end with $): ");

yylex(); // Start Lexical Analysis

return 0;

}

6 three address code

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "optimizer.tab.h" // Include the header generated by Bison

%}

%%

// Match identifiers (variable names)

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

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

// Match operators and symbols

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"=" { return '='; }

";" { return ';' }

"(" { return '('; }

")" { return ')'; }

// Ignore whitespace and comments

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

. { return yytext[0]; } // Return single characters as tokens

%%

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern void generate\_code(const char \*op, const char \*arg1, const char \*arg2);

void optimize\_code();

void constant\_folding(char \*op, int left, int right);

void strength\_reduction(char \*op, int left, int right);

void algebraic\_transformation(char \*op, int left, int right);

%}

%union {

int int\_val;

char \*str\_val;

}

%token <int\_val> NUMBER

%token <str\_val> IDENTIFIER

%token '=' '+' '-' '\*' '/' ';' '(' ')'

%%

input:

| input statement

;

statement:

expr ';' { optimize\_code(); }

;

expr:

expr '+' expr { constant\_folding("+", $1, $3); strength\_reduction("+", $1, $3); algebraic\_transformation("+", $1, $3); }

| expr '-' expr { constant\_folding("-", $1, $3); strength\_reduction("-", $1, $3); algebraic\_transformation("-", $1, $3); }

| expr '\*' expr { constant\_folding("\*", $1, $3); strength\_reduction("\*", $1, $3); algebraic\_transformation("\*", $1, $3); }

| expr '/' expr { constant\_folding("/", $1, $3); strength\_reduction("/", $1, $3); algebraic\_transformation("/", $1, $3); }

| '(' expr ')' { $$ = $2; }

| NUMBER { $$ = $1; }

| IDENTIFIER { $$ = $1; }

;

%%

void constant\_folding(char \*op, int left, int right) {

if (left >= 0 && right >= 0) { // Only fold constants

int result;

if (strcmp(op, "+") == 0) result = left + right;

else if (strcmp(op, "-") == 0) result = left - right;

else if (strcmp(op, "\*") == 0) result = left \* right;

else if (strcmp(op, "/") == 0 && right != 0) result = left / right;

else return; // Division by zero guard

// Output the result (or replace the expression with the result)

char temp[20];

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

generate\_code("=", temp, "0");

}

}

void strength\_reduction(char \*op, int left, int right) {

// Example for strength reduction: replacing `x \* 2` with `x + x`

if (strcmp(op, "\*") == 0 && right == 2) {

generate\_code("+", left, left); // Change x \* 2 to x + x

}

}

void algebraic\_transformation(char \*op, int left, int right) {

// Example for algebraic transformation

if (strcmp(op, "+") == 0 && right == 0) {

// Replace `a + 0` with `a`

generate\_code("=", left, "0");

}

}

void generate\_code(const char \*op, const char \*arg1, const char \*arg2) {

printf("%s %s %s\n", op, arg1, arg2); // A simple example of code generation

}

void yyerror(const char \*s) {

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

}

int main(void) {

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

yyparse(); // Start parsing

return 0;

}

3d calculatior

%{

#include "y.tab.h" // Include the YACC header for token definitions

%}

%option noyywrap

%%

// Define patterns for tokens

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

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"(" { return '('; }

")" { return ')'; }

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

. { return yytext[0]; } // Return single characters as tokens

%%

// Main function for the lexer

int main(void) {

return yylex();

}

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

int yylex(); // Lex function declaration

%}

%token NUMBER // Declare the token from the Lex file

%%

// Define grammar rules

expr:

expr '+' expr { $$ = $1 + $3; }

| expr '-' expr { $$ = $1 - $3; }

| expr '\*' expr { $$ = $1 \* $3; }

| expr '/' expr {

if ($3 == 0) {

yyerror("Division by zero!");

$$ = 0; // Handle division by zero

} else {

$$ = $1 / $3;

}

}

| '(' expr ')' { $$ = $2; }

| NUMBER { $$ = $1; }

;

%%

// Error handling function

void yyerror(const char \*s) {

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

}

int main(void) {

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

yyparse(); // Start parsing

return 0;

}

1,2

%{

#include <stdio.h>

#include <string.h>

void print\_token(char \*type, char \*value) {

printf("%s: %s\n", type, value);

}

%}

%option noyywrap

/\* Define the patterns for tokens \*/

%%

"int"|"float"|"double"|"char"|"void" { print\_token("KEYWORD", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { print\_token("IDENTIFIER", yytext); }

"=="|"!="|"<"|"<="|">"|">="|"+"|"-"|"\*"|"/" { print\_token("OPERATOR", yytext); }

[0-9]+ { print\_token("NUMBER", yytext); }

" "|\t|\n { /\* Ignore whitespace \*/ }

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

%%

int main(void) {

yylex(); // Start the lexical analysis

return 0;

}

int yywrap(void) {

return 1; // Indicate end of input

}