1

%{

#include <stdio.h>

#include <ctype.h>

int line\_count = 0, signed\_int\_count = 0, signed\_float\_count = 0;

%}

%%

"int"|"float"|"char"|"double"|"if"|"else"|"while"|"for"|"do"|"return"|"switch"|"case"|"break"|"continue"|"void"|"class"|"public"|"private"|"protected"|"try"|"catch"|"throw"|"const"|"new"|"delete"|"this"|"namespace"|"template" { printf("C++ reserved word: %s\n", yytext); }

"def"|"import"|"from"|"as"|"pass"|"break"|"continue"|"return"|"yield"|"class"|"try"|"except"|"raise"|"global"|"nonlocal"|"lambda"|"with"|"True"|"False"|"None" { printf("Python reserved word: %s\n", yytext); }

"scanf" { printf("printf"); }

(0|1){1,6} {

    if (yytext[0] != yytext[yyleng - 1])

        printf("Valid binary string (1-6 length) with different start/end: %s\n", yytext);

}

^[aeiouAEIOU][a-zA-Z]\* {

    printf("String begins with a vowel: %s\n", yytext);

}

[+-]?[0-9]+ { signed\_int\_count++; printf("Signed integer: %s\n", yytext); }

[+-]?[0-9]+\.[0-9]+ { signed\_float\_count++; printf("Signed floating-point number: %s\n", yytext); }

\n { line\_count++; }

"+"|"-"|"\*"|"/"|"%" { printf("Arithmetic operator: %s\n", yytext); }

"="|"=="|"!="|"<"|">"|"<="|">=" { printf("Relational operator: %s\n", yytext); }

"&&"|"||"|"!" { printf("Logical operator: %s\n", yytext); }

"&"|"|"|"^"|"<<"|">>" { printf("Bitwise operator: %s\n", yytext); }

"++"|"--" { printf("Increment/Decrement operator: %s\n", yytext); }

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

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

"//".\* { }

"/\*"[^"\*/"]\*"\*/"    { }

[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,10} {

    printf("Valid email address: %s\n", yytext);

}

. { /\* Catch-all rule to handle unmatched text \*/ }

%%

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

{

    yylex();

    printf("Total lines: %d\n", line\_count);

    printf("Total signed integers: %d\n", signed\_int\_count);

    printf("Total signed floating-point numbers: %d\n", signed\_float\_count);

    return 0;

}

2

%{

#include "y.tab.h"

%}

%%

[0-9]+(\.[0-9]+)?      { yylval.val = atof(yytext); return NUMBER; }  // Numbers (integers and floats)

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

"("                    { return LPAREN; }

")"                    { return RPAREN; }

"+"                    { return PLUS; }

"\*"                    { return TIMES; }

"/"                    { return DIVIDE; }

\n                     { return EOL; }

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

%%

int yywrap() {

    return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void yyerror(const char \*s);

int yylex();

%}

%union {

    double val;  // Use double to handle floating-point calculations

}

%token <val> NUMBER

%token PLUS MINUS TIMES DIVIDE

%token LPAREN RPAREN

%token EOL

%type <val> expr term factor

**%%**

input:

    /\* Empty production rule \*/

    | input line

;

line:

    expr EOL       { printf("Valid expression. Result: %f\n", $1); }

    | EOL          { /\* Ignore empty lines \*/ }

;

expr:

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

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

    | term             { $$ = $1; }

;

term:

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

    | term DIVIDE factor {

        if ($3 == 0) {

            yyerror("Division by zero");

            YYABORT;

        } else {

            $$ = $1 / $3;

        }

    }

    | factor           { $$ = $1; }

;

factor:

    LPAREN expr RPAREN { $$ = $2; }

    | NUMBER           { $$ = $1; }

;

**%%**

void yyerror(const char \*s) {

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

}

int main() {

    printf("Enter an arithmetic expression (Ctrl+D to exit):\n");

    yyparse();

    return 0;

}

3

%{

#include "y.tab.h"

%}

%%

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

[\n]                { return EOL; }

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

"+"                 { return PLUS; }

"-"                 { return MINUS; }

"\*"                 { return TIMES; }

"/"                 { return DIVIDE; }

"("                 { return LPAREN; }

")"                 { return RPAREN; }

%%

int yywrap() {

    return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void yyerror(const char \*s);

int yylex();

#define MAX\_POSTFIX\_LENGTH 100

char postfix[MAX\_POSTFIX\_LENGTH];

int postfix\_index = 0;

void add\_to\_postfix(char \*str) {

    if (postfix\_index + strlen(str) < MAX\_POSTFIX\_LENGTH) {

        strcat(postfix, str);

        postfix\_index += strlen(str);

    } else {

        yyerror("Postfix expression too long.");

    }

}

%}

%token NUMBER

%token PLUS MINUS TIMES DIVIDE

%token LPAREN RPAREN

%token EOL

**%%**

input:

    /\* Empty production rule \*/

    | input line

;

line:

    expr EOL   { printf("Postfix: %s\n", postfix); postfix[0] = '\0'; postfix\_index = 0; }

    | EOL      { /\* Ignore empty lines \*/ }

;

expr:

    expr PLUS term   { add\_to\_postfix(" "); add\_to\_postfix("+"); }

    | expr MINUS term { add\_to\_postfix(" "); add\_to\_postfix("-"); }

    | term

;

term:

    term TIMES factor { add\_to\_postfix(" "); add\_to\_postfix("\*"); }

    | term DIVIDE factor { add\_to\_postfix(" "); add\_to\_postfix("/"); }

    | factor

;

factor:

    LPAREN expr RPAREN

    | NUMBER {

        char buf[20];

        sprintf(buf, "%d ", $1);

        add\_to\_postfix(buf);

    }

;

**%%**

void yyerror(const char \*s) {

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

}

int main() {

    printf("Enter an infix expression (Ctrl+D to exit):\n");

    yyparse();

    return 0;

}

//3 + 4 \* (2 - 1)

6

%{

#include "y.tab.h"

%}

%%

"if"                 { return IF; }

"("                  { return LPAREN; }

")"                  { return RPAREN; }

"{"                  { return LBRACE; }

"}"                  { return RBRACE; }

";"                  { return SEMICOLON; }

"="                  { return ASSIGN; }

"<"                  { return LT; }

">"                  { return GT; }

"=="                 { return EQ; }

"!="                 { return NE; }

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

[0-9]+               { return NUMBER; } // Numbers

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

%%

int yywrap() {

    return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

int yylex();

%}

%token IF LPAREN RPAREN LBRACE RBRACE SEMICOLON ASSIGN

%token LT GT EQ NE

%token IDENTIFIER NUMBER

**%%**

input:

    /\* empty \*/

    | input statement

;

statement:

    if\_statement

;

if\_statement:

    IF LPAREN condition RPAREN LBRACE stmt\_list RBRACE {

        printf("Valid if statement\n");

    }

;

condition:

    IDENTIFIER relational\_operator IDENTIFIER

    | IDENTIFIER relational\_operator NUMBER

    | NUMBER relational\_operator IDENTIFIER

    | NUMBER relational\_operator NUMBER

;

relational\_operator:

    LT

    | GT

    | EQ

    | NE

;

stmt\_list:

    assignment SEMICOLON

    | assignment SEMICOLON stmt\_list

;

assignment:

    IDENTIFIER ASSIGN expr

;

expr:

    IDENTIFIER

    | NUMBER

;

**%%**

void yyerror(const char \*s) {

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

}

int main() {

    printf("Enter an if statement to validate (Ctrl+D to exit):\n");

    yyparse();

    return 0;

}

7

%{

#include "y.tab.h"

%}

%%

"if"                { return IF; }

"elif"              { return ELIF; }

"else"              { return ELSE; }

":"                 { return COLON; }

"=="                { return EQ; }

"!="                { return NE; }

"<"                 { return LT; }

">"                 { return GT; }

"<="                { return LE; }

">="                { return GE; }

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

[0-9]+              { return NUMBER; }

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

\n                  { return NEWLINE; }

%%

int yywrap() {

    return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

int yylex();

%}

%token IF ELIF ELSE COLON NEWLINE

%token IDENTIFIER NUMBER

%token EQ NE LT GT LE GE

**%%**

input:

    /\* empty \*/

    | input statement\_block

;

statement\_block:

    if\_statement

;

if\_statement:

    IF condition COLON NEWLINE stmt\_list elif\_clause else\_clause

    {

        printf("Valid if-elif-else statement\n");

    }

;

condition:

    IDENTIFIER relational\_operator IDENTIFIER

    | IDENTIFIER relational\_operator NUMBER

    | NUMBER relational\_operator IDENTIFIER

    | NUMBER relational\_operator NUMBER

;

relational\_operator:

    EQ

    | NE

    | LT

    | GT

    | LE

    | GE

;

elif\_clause:

    /\* empty \*/

    | ELIF condition COLON NEWLINE stmt\_list elif\_clause

;

else\_clause:

    /\* empty \*/

    | NEWLINE ELSE COLON NEWLINE stmt\_list

;

stmt\_list:

    statement

    | statement NEWLINE stmt\_list

;

statement:

    IDENTIFIER '=' expr

;

expr:

    IDENTIFIER

    | NUMBER

;

**%%**

void yyerror(const char \*s) {

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

}

int main() {

    printf("Enter a Python-style if-elif-else statement to validate (Ctrl+D to exit):\n");

    yyparse();

    return 0;

}

8

%{

#include "y.tab.h"

%}

%%

"["              { return LBRACKET; }

"]"              { return RBRACKET; }

"{"              { return LBRACE; }

"}"              { return RBRACE; }

":"              { return COLON; }

","              { return COMMA; }

[0-9]+           { return NUMBER; }

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

[ \t\n]+         { /\* Skip whitespace and newlines \*/ }

%%

int yywrap() {

    return 1;

}