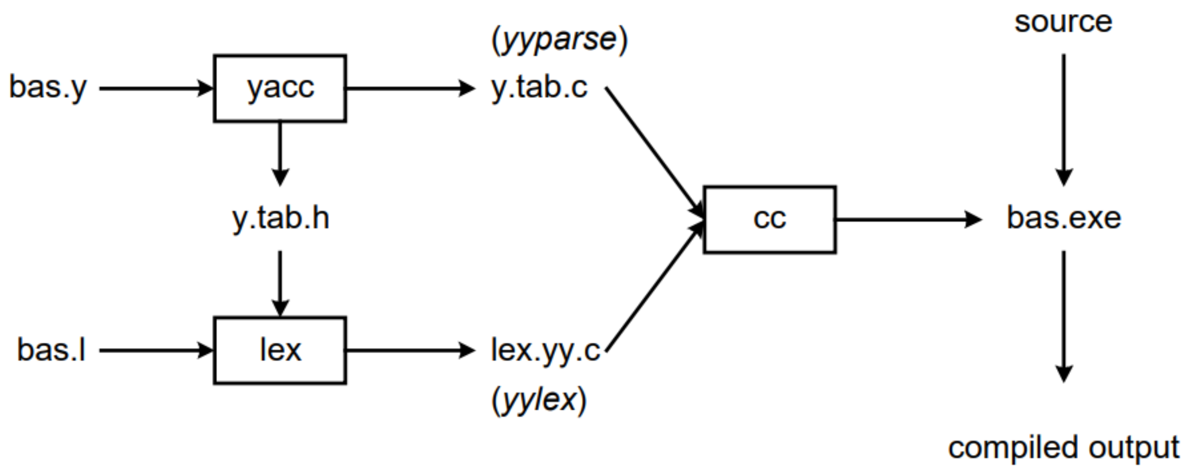
LAB SESSION 7: YET ANOTHER COMPILER COMPILER

AIM: To learn YACC tool for syntax and semantic analysis and implement basic YACC Programs.

PROBLEM DEFINITION: Write YACC programs to:

1. Validate syntax of declaration statement for given input.
2. Parse an arithmetic expression of the form: num1 op num2
3. Evaluate the arithmetic expressions
4. Parse assignment statement

THEORY: Yacc generates C code for a syntax analyzer, or parser. Yacc uses grammar rules that allow it to analyze tokens from lex and create a syntax tree. A syntax tree imposes a hierarchical structure on tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly.



We'll assume our goal is to write a BASIC compiler. First, we need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y). Commands to create our compiler, bas.exe, are listed below:

1. yacc –d bas.y # create y.tab.h, y.tab.c

2. lex bas.l # create lex.yy.c
3. cc lex.yy.c y.tab.c –obas.exe # compile/link

Yacc reads the grammar descriptions in bas.y and generates a parser, function yyparse, in file y.tab.c. Included in file bas.y are token declarations. These are converted to constant definitions by yacc and placed in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, function yylex, in file lex.yy.c.
Finally, the lexer and parser are compiled and linked together to form the executable, bas.exe. From main, we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique was pioneered by John Backus and Peter Naur, and used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

1. E -> E + E
2. E -> E * E
3. E -> id

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

E -> E * E (r2)
-> E * z (r3)
-> E + E * z (r1)
-> E + y * z (r3)
-> x + y * z (r3)

At each step we expanded a term, replacing the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression, we actually need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar, we need to reduce an expression to a single nonterminal. This is known as bottom-up or shift-reduce parsing, and uses a stack for storing terms. Here is the same derivation, but in reverse order:

1. .x+y*z shift
2. x.+y*z reduce(r3)
3. E.+y*z shift
4. E+.y*z shift
5. E+y.*z reduce(r3)
6. E+E.*z shift
7. E+E*.z shift
8. E+E*z. reduce(r3)
9. E+E*E. reduce(r2) emit multiply
10. E + E . reduce(r1) emit add
11. E . accept

Terms to the left of the dot are on the stack, while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production, we replace the matched tokens on the stack with the lhs of the production. Conceptually, the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a handle, and we are reducing the handle to the lhs of the production. This process continues until we have shifted all input to the stack, and only the starting nonterminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack, changing x to E. We continue shifting and reducing, until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction.

Similarly, the add instruction is emitted in step 10. Thus, multiply has a higher precedence than addition.

Consider, however, the shift at step 6. Instead of shifting, we could have reduced,applying rule r1. This would result in addition having a higher precedence than multiplication. This is known as a shift-reduce conflict. Our grammar is ambiguous, as there is more than one possible derivation that will yield the expression. In this case, operator precedence is affected. As another example, associativity in the rule

E -> E + E

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar, or supply yacc with directives that indicate which operator has precedence. The latter method is simpler, and will be demonstrated in the practice section.

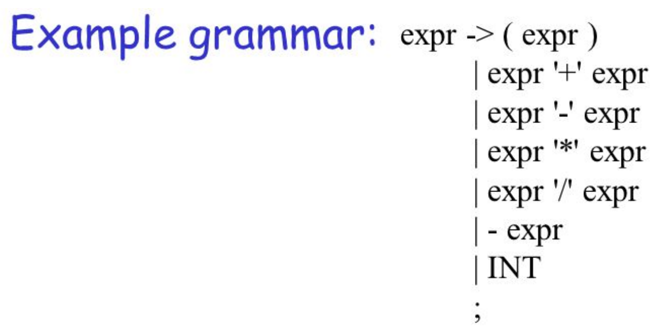
The following grammar has a reduce-reduce conflict. With an id on the stack, we may reduce to T, or reduce to E.

E -> T

E -> id

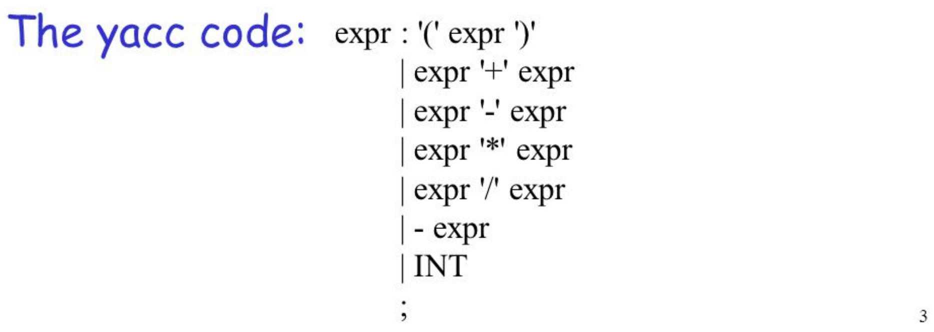
T -> id

Yacc takes a default action when there is a conflict. For shift-reduce conflicts, yacc will shift. For reduce-reduce conflicts, it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.



Example grammar:

```
expr -> ( expr )
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | - expr
      | INT
      ;
```



The yacc code:

```
expr : '(' expr ')'
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | - expr
     | INT
     ;
```

3

PROGRAM 1:
Lex file:

```
%{
#include "decl.tab.h"
#include <stdlib.h>
#include <string.h>

/* Simple typedef name table */
#define MAX_TYPENAMES 1024
static char* type_names[MAX_TYPENAMES];
static int type_name_count = 0;
static int typedef_mode = 0; /* set when scanning a typedef declaration */

static int is_typedef_name(const char* s) {
	for (int i = 0; i < type_name_count; ++i) {
		if (strcmp(type_names[i], s) == 0) return 1;
	}
	return 0;
}

static void add_typedef_name(const char* s) {
	if (is_typedef_name(s)) return;
	if (type_name_count < MAX_TYPENAMES) {
		type_names[type_name_count++] = _strdup(s);
	}
}

int yywrap(void){ return 1; }
%}


%%

[a-zA-Z_][a-zA-Z0-9_]*    {
	/* Keywords for basic and modifier types */
	if (strcmp(yytext, "int") == 0) return INT;
	if (strcmp(yytext, "float") == 0) return FLOAT;
	if (strcmp(yytext, "double") == 0) return DOUBLE;
	if (strcmp(yytext, "char") == 0) return CHAR;
	if (strcmp(yytext, "short") == 0) return SHORT;
	if (strcmp(yytext, "long") == 0) return LONG;
	if (strcmp(yytext, "signed") == 0) return SIGNED;
```

```
        if (strcmp(yytext, "unsigned") == 0) return UNSIGNED;
        if (strcmp(yytext, "void") == 0) return VOID;
        if (strcmp(yytext, "const") == 0) return CONST;
        if (strcmp(yytext, "volatile") == 0) return VOLATILE;
        if (strcmp(yytext, "typedef") == 0) { typedef_mode = 1; return TYPEDEF; }
        if (strcmp(yytext, "static") == 0) return STATIC;
        if (strcmp(yytext, "extern") == 0) return EXTERN;
        if (strcmp(yytext, "register") == 0) return REGISTER;
        if (strcmp(yytext, "struct") == 0) return STRUCT;
        if (strcmp(yytext, "union") == 0) return UNION;
        if (strcmp(yytext, "enum") == 0) return ENUM;

        /* typedef-name handling */
        if (typedef_mode) { add_typedef_name(yytext); return TYPE_NAME; }
        if (is_typedef_name(yytext)) return TYPE_NAME;
        return ID;
}
","         { return COMMA; }
";"         { typedef_mode = 0; return SEMICOLON; }
"="         { return ASSIGN; }
"*"         { return ASTERISK; }
"["         { return LBRACKET; }
"]"         { return RBRACKET; }
"("         { return LPAREN; }
")"         { return RPAREN; }
"{"         { return LBRACE; }
"}"         { return RBRACE; }
        [0-9]+\.[0-9]*([eE][+-]?[0-9]+)?   { return FLOATCONST; }
        \.[0-9]+([eE][+-]?[0-9]+)?         { return FLOATCONST; }
        [0-9]+([eE][+-]?[0-9]+)            { return FLOATCONST; }

[0-9]+                      { return NUMBER; }

        \"([^\\\"\n]|\\.)*\"         { return STRINGLIT; }
'(\\.|[^\\\n])'             { return CHARCONST; }

[ \t\n\r]+    { /* skip whitespace */ }
.           { return INVALID; }
%%
```

YACC file:

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
void yyerror(const char *s);
static int g_error = 0;
%}

%union {
   int ival; /* 1 if declarator denotes a function, else 0 */
}

%token VOID CHAR SHORT INT LONG FLOAT DOUBLE SIGNED UNSIGNED
%token CONST VOLATILE TYPEDEF STATIC EXTERN REGISTER
%token STRUCT UNION ENUM
%token TYPE_NAME
%token ID NUMBER FLOATCONST CHARCONST STRINGLIT
%token COMMA SEMICOLON ASSIGN ASTERISK LBRACKET RBRACKET LPAREN
RPAREN LBRACE RBRACE INVALID

%type <ival> declarator direct_declarator

%%

decl
     : decl_specifiers init_declarator_list SEMICOLON    { if (!g_error) printf("Valid
declaration\n"); }
   | decl_specifiers SEMICOLON                 { if (!g_error) printf("Valid declaration\n");
}
   ;

/* declaration specifiers: storage class | type specifiers | type qualifiers */
decl_specifiers
   : decl_specifiers decl_specifier
   | decl_specifier
   ;

decl_specifier
   : storage_class_specifier
   | type_qualifier
   | type_token
   | struct_or_union_specifier
```

```
    | enum_specifier
    ;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | REGISTER
    ;

type_token
    : VOID
    | CHAR
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | TYPE_NAME
    ;

/* struct/union definitions or tags */
struct_or_union_specifier
    : STRUCT opt_id LBRACE member_declaration_list_opt RBRACE
    | UNION  opt_id LBRACE member_declaration_list_opt RBRACE
    ;

opt_id
    : /* empty */
    | ID
    ;

member_declaration_list_opt
    : /* empty */
    | member_declaration_list
    ;

member_declaration_list
    : member_declaration
    | member_declaration_list member_declaration
    ;

member_declaration
    : decl_specifiers member_declarator_list_opt SEMICOLON
    ;

member_declarator_list_opt
    : /* empty */
    | member_declarator_list
    ;

member_declarator_list
    : member_declarator
    | member_declarator_list COMMA member_declarator
    ;

member_declarator
    : declarator
    ;

/* enum definitions or tags */
enum_specifier
    : ENUM opt_id LBRACE enumerator_list_opt RBRACE
    | ENUM opt_id LBRACE enumerator_list COMMA RBRACE
    | ENUM ID
    ;

enumerator_list_opt
    : /* empty */
    | enumerator_list
    ;

enumerator_list
    : enumerator
    | enumerator_list COMMA enumerator
    ;

enumerator
    : ID
    | ID ASSIGN const_expr_opt
    ;
```

```
type_qualifier
   : CONST
   | VOLATILE
   ;

init_declarator_list
   : init_declarator
   | init_declarator_list COMMA init_declarator
   ;

/* Disallow initializers for function declarators by only allowing assignment
  to non-function declarators */
init_declarator
   : declarator
    | declarator ASSIGN initializer   { if ($1) yyerror("function declarator cannot be
initialized"); }
   ;

/* pointer and array declarators like: *p, **q, a[10], b[3][4] */
declarator
   : pointer_opt direct_declarator   { $$ = $2; }
   ;

pointer_opt
   : /* empty */
   | pointer
   ;

/* pointer with optional qualifiers after each '*' */
pointer
   : ASTERISK type_qual_list_opt
   | ASTERISK type_qual_list_opt pointer
   ;

type_qual_list_opt
   : /* empty */
   | type_qual_list
   ;

type_qual_list
   : type_qualifier
   | type_qual_list type_qualifier
```

```
    ;

direct_declarator
    : ID                                        { $$ = 0; }
    | LPAREN declarator RPAREN                  { $$ = $2; }
    | direct_declarator LBRACKET const_expr_opt RBRACKET   { $$ = $1; }
    | direct_declarator LPAREN parameter_list_opt RPAREN   { $$ = 1; }
    ;

const_expr_opt
    : /* empty */
    | NUMBER
    | ID
    ;

/* keep initializer simple (constants or identifiers) */
initializer
    : NUMBER
    | FLOATCONST
    | CHARCONST
    | STRINGLIT
    | ID
    ;

/* parameters: simplified */
parameter_list_opt
    : /* empty */
    | parameter_list
    ;

parameter_list
    : parameter_declaration
    | parameter_list COMMA parameter_declaration
    ;

parameter_declaration
    : decl_specifiers declarator
    | decl_specifiers
    ;

%%
void yyerror(const char *s) {
    g_error = 1;
    printf("Invalid declaration\n");
}

int main() {
    printf("Enter declaration: ");
    g_error = 0;
    yyparse();
    return 0;
}
```

all.tst:

```
int a[];
int a, b, c;
char c = 'a';
int , a;
int * const p;
enum E;
int f();
int (*fp)(int, char);
int f(int a, char b);
float x = 3.14, y;
int f() = 3;
short [10] a;
double m[3][4], *p = 0;
int a, ;
void *ptr;
```

OUTPUT 1:

```
PS C:\Users\Joseph\Desktop\compiler design\expt7> cd "c:\Users\Joseph\Desktop\compiler design\expt7\tests"; if (Test-Path .\decl\all.out) { Remove-Item .\decl\all.out }; Get-Content .\decl\all.tst | ForEach-Object { $_ | ..\decl.exe 2>$null } | Set-Content -Encoding ascii .\decl\all.out; Compare-Object (Get-Content .\decl\all.actual) (Get-Content .\decl\all.out)

PS C:\Users\Joseph\Desktop\compiler design\expt7\tests> # From c:\Users\Joseph\Desktop\compiler design\expt7\tests

PS C:\Users\Joseph\Desktop\compiler design\expt7\tests> Get-Content .\decl\all.tst | ForEach-Object { $_ | ..\decl.exe 2>$null } | Set-Content -Encoding ascii .\decl\all.out
PS C:\Users\Joseph\Desktop\compiler design\expt7\tests> Compare-Object (Get-Content .\decl\all.actual) (Get-Content .\decl\all.out)
PS C:\Users\Joseph\Desktop\compiler design\expt7\tests>
```

all.out:

```
Enter declaration: Valid declaration
Enter declaration: Valid declaration
Enter declaration: Valid declaration
Enter declaration: Invalid declaration
Enter declaration: Valid declaration
Enter declaration: Valid declaration
Enter declaration: Valid declaration
Enter declaration: Valid declaration
Enter declaration: Valid declaration
Enter declaration: Valid declaration
Enter declaration: Invalid declaration
Enter declaration: Invalid declaration
Enter declaration: Valid declaration
Enter declaration: Invalid declaration
Enter declaration: Valid declaration
```

PROGRAM 2:

Lex file:

```
%{
#include "binexpr.tab.h"
#include <stdlib.h>
int yywrap(void){ return 1; }
%}

%%
[+-]?([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][+-]?[0-9]+)?   { yylval = strtod(yytext, NULL);
return NUM; }
[ \t]+    ;
"\r\n"   { return '\n'; }
"\n"     { return '\n'; }
"\r"     ;
"+"      { return '+'; }
"-"      { return '-'; }
"*"      { return '*'; }
"/"      { return '/'; }
```

```
"%"      { return '%'; }
"^"      { return '^'; }
.        { return yytext[0]; }
%%
```

YACC file:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int yylex(void);
void yyerror(const char *s);

/* Print helper: show integer without .0, otherwise compact float */
static void print_num(double v) {
  long long iv = (long long)v;
  if (v == (double)iv) {
    printf("%lld", iv);
  } else {
    printf("%g", v);
  }
}
%}

%define api.value.type {double}

%token NUM

%%

input:
    /* empty */
   | input line
   ;

line:
    NUM '+' NUM '\n'   { double r = $1 + $3; printf("Parsed: "); print_num($1); printf(" + "); print_num($3); printf(" => Result = "); print_num(r); printf("\n"); }
   | NUM '-' NUM '\n'   { double r = $1 - $3; printf("Parsed: "); print_num($1); printf(" - "); print_num($3); printf(" => Result = "); print_num(r); printf("\n"); }
   | NUM '*' NUM '\n'   { double r = $1 * $3; printf("Parsed: "); print_num($1); printf(" * "); print_num($3); printf(" => Result = "); print_num(r); printf("\n"); }
```

```
    | NUM '/' NUM '\n'    { if($3==0.0) { printf("Error: division by zero\n"); } else { double r = $1 / $3; printf("Parsed: "); print_num($1); printf(" / "); print_num($3); printf(" => Result = "); print_num(r); printf("\n"); } }
  | NUM '%' NUM '\n'    { if($3==0.0) { printf("Error: modulo by zero\n"); } else { double r = fmod($1, $3); printf("Parsed: "); print_num($1); printf(" %% "); print_num($3); printf(" => Result = "); print_num(r); printf("\n"); } }
    | NUM '^' NUM '\n'    { double r = pow($1, $3); printf("Parsed: "); print_num($1); printf(" ^ "); print_num($3); printf(" => Result = "); print_num(r); printf("\n"); }
    ;

%%

void yyerror(const char *s) { fprintf(stderr, "Invalid input\n"); }

int main(void) {
   printf("Enter binary expressions (num1 op num2). Ctrl+Z then Enter to quit.\n");
   yyparse();
   return 0;
}
```

all.tst:

```
2 + 3
-3 - -4.5
12.5 * 2
10 / 4
1 / 2
5 / 0
5 % 2
5.5 % 2
5 % 0
2 ^ 3
-2 ^ 2
2 ^ -3
+1e2 / 4e1
```

OUTPUT 2:

```
PS C:\Users\Joseph\Desktop\compiler design\expt7> cd "c:\Users\Joseph\Desktop\compiler design\expt7"; if (Test-Path .\binexpr.exe) { Write-Output "exists" } else { bison -d binexpr.y; flex binexpr.l; gcc lex.yy.c
binexpr.tab.c -o binexpr.exe; if (Test-Path .\binexpr.exe) { Write-Output "built" } else
{ Write-Output "missing" } }
exists
PS C:\Users\Joseph\Desktop\compiler design\expt7> cd
"c:\Users\Joseph\Desktop\compiler design\expt7"; Get-Content
"tests\binexpr\all.tst" | .\binexpr.exe 2>$null | Out-File -Encoding ascii
"tests\binexpr\all.out"
PS C:\Users\Joseph\Desktop\compiler design\expt7> cd
"c:\Users\Joseph\Desktop\compiler design\expt7"; $diff = Compare-Object (Get-
Content "tests\binexpr\all.actual") (Get-Content "tests\binexpr\all.out"); if ($diff) {
Write-Output "FAIL" } else { Write-Output "PASS" }
PASS
PS C:\Users\Joseph\Desktop\compiler design\expt7>
```

all.out:

```
Enter binary expressions (num1 op num2). Ctrl+Z then Enter to quit.
Parsed: 2 + 3 => Result = 5
Parsed: -3 - -4.5 => Result = 1.5
Parsed: 12.5 * 2 => Result = 25
Parsed: 10 / 4 => Result = 2.5
Parsed: 1 / 2 => Result = 0.5
Error: division by zero
Parsed: 5 % 2 => Result = 1
Parsed: 5.5 % 2 => Result = 1.5
Error: modulo by zero
Parsed: 2 ^ 3 => Result = 8
Parsed: -2 ^ 2 => Result = 4
Parsed: 2 ^ -3 => Result = 0.125
Parsed: 100 / 40 => Result = 2.5
```

```
PS C:\Users\Joseph\Desktop\compiler design\expt7> echo "12 + 5 + 3" |
.\binexpr.exe
Enter binary expressions (num1 op num2). Ctrl+Z then Enter to quit.
Invalid input
PS C:\Users\Joseph\Desktop\compiler design\expt7> echo "12*3" | .\binexpr.exe
Enter binary expressions (num1 op num2). Ctrl+Z then Enter to quit.
Parsed: 12 * 3 => Result = 36
PS C:\Users\Joseph\Desktop\compiler design\expt7> echo "--4" | .\binexpr.exe
Enter binary expressions (num1 op num2). Ctrl+Z then Enter to quit.
Invalid input
```

PROGRAM 3:
Lex file:
```
%{
#include "expr.tab.h"
#include <stdlib.h>
int lookup(const char* name);

/* yylval is a union declared by Bison in expr.y */

/* Provide yywrap so linking doesn't require -lfl on Windows/mingw */
int yywrap(void) { return 1; }
%}

%%
[0-9]+(\.[0-9]*)?([eE][+-]?[0-9]+)?  { yylval.d = strtod(yytext, NULL); return NUM; }
\.[0-9]+([eE][+-]?[0-9]+)?          { yylval.d = strtod(yytext, NULL); return NUM; }
"++"     { return INC; }
"--"     { return DEC; }
[A-Za-z_][A-Za-z0-9_]* { yylval.sym = lookup(yytext); return ID; }
[ \t]+    ;
"\n"     { return '\n'; }
"+"      { return '+'; }
"-"      { return '-'; }
"*"      { return '*'; }
"/"      { return '/'; }
"%"      { return '%'; }
"^"      { return '^'; }
"="      { return '='; }
"("      { return '('; }
")"      { return ')'; }
.        { return yytext[0]; }
%%
```

YACC file:
```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

int yylex(void);
void yyerror(const char *s);

/* Track whether the current input line had an error (syntax or semantic) */
static int had_error_line = 0;

/* Simple symbol table for identifiers */
#define MAXSYMS 1024
#define NAMELEN 63
typedef struct { int in_use; char name[NAMELEN+1]; double val; } Sym;
static Sym syms[MAXSYMS];

int lookup(const char* name) {
   for (int i = 0; i < MAXSYMS; ++i) {
      if (syms[i].in_use && strcmp(syms[i].name, name) == 0) return i;
   }
   for (int i = 0; i < MAXSYMS; ++i) {
      if (!syms[i].in_use) {
         syms[i].in_use = 1;
         strncpy(syms[i].name, name, NAMELEN);
         syms[i].name[NAMELEN] = '\0';
         syms[i].val = 0.0;
         return i;
      }
   }
   yyerror("symbol table full");
   return 0;
}
%}

%union { double d; int sym; }

%token <d> NUM
%token <sym> ID
%token INC DEC
%left '+' '-'
%left '*' '/' '%'
%right UMINUS
%right '^'

%type <d> expr

%%
```

```
input:
      /* empty */
    | input line
    ;

line:
      expr '\n'        { if (!had_error_line) printf("Result = %g\n", $1); had_error_line
= 0; }
     | ID '=' expr '\n'   { if (!had_error_line) { syms[$1].val = $3; printf("%s = %g\n",
syms[$1].name, syms[$1].val); } had_error_line = 0; }
    | error '\n'       { had_error_line = 0; yyerrok; }
    ;

expr:
      expr '+' expr   { $$ = $1 + $3; }
    | expr '-' expr   { $$ = $1 - $3; }
    | expr '*' expr   { $$ = $1 * $3; }
    | expr '/' expr   { if ($3 == 0.0) { yyerror("division by zero"); $$ = 0.0; } else $$ =
$1 / $3; }
    | expr '%' expr   { if ($3 == 0.0) { yyerror("modulo by zero"); $$ = 0.0; } else $$ =
fmod($1, $3); }
    | expr '^' expr   { $$ = pow($1, $3); }
    | '-' expr %prec UMINUS { $$ = -$2; }
    | '(' expr ')'    { $$ = $2; }
    | NUM             { $$ = $1; }
    | ID            { $$ = syms[$1].val; }
    | INC ID         { syms[$2].val += 1.0; $$ = syms[$2].val; }  /* ++x */
    | DEC ID         { syms[$2].val -= 1.0; $$ = syms[$2].val; }  /* --x */
    | ID INC         { $$ = syms[$1].val; syms[$1].val += 1.0; }  /* x++ */
    | ID DEC         { $$ = syms[$1].val; syms[$1].val -= 1.0; }  /* x-- */
    ;

%%

void yyerror(const char *s) {
  had_error_line = 1;
  fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
  printf("Enter expression (press Ctrl+Z then Enter to quit on Windows)\n");
```

```
   yyparse();
   return 0;
}
```

all.tst:

```
3 + 4 * 5
7 / 0
1 / 2
3.5 + 0.5
2.0 * (1.5 + 2.5)
10.0 / 4
x = 5
--x
x--
++x
x++
3 + @
123456 * 0
99999 + 1
1 + 2
2 * (3 + 4)
-5 + 10 / 2
(3 + 4) * 5
2^3
2^3^2
-2^2
(-2)^2
5 % 2
5.5 % 2
5 % 0
-3 + 2
2 * -3
-(3 + 4)

          10     +        20
```

OUTPUT 3:

```
PS C:\Users\Joseph\Desktop\compiler design\expt7> bison -d expr.y
PS C:\Users\Joseph\Desktop\compiler design\expt7> flex expr.l
PS C:\Users\Joseph\Desktop\compiler design\expt7> gcc lex.yy.c expr.tab.c -o expr.exe
PS C:\Users\Joseph\Desktop\compiler design\expt7> Get-Content -LiteralPath 'tests/expr/all.tst' | .\expr.exe 2>$null | Out-File -Encoding ascii -LiteralPath 'tests/expr/all.out'; if ((Compare-Object (Get-Content 'tests/expr/all.actual') (Get-Content 'tests/expr/all.out'))) { 'DIFF' } else { 'MATCH' }
MATCH
```

all.out:

```
Enter expression (press Ctrl+Z then Enter to quit on Windows)
Result = 23
Result = 0.5
Result = 4
Result = 8
Result = 2.5
x = 5
Result = 4
Result = 4
Result = 4
Result = 4
Result = 0
Result = 100000
Result = 3
Result = 14
Result = 0
Result = 35
Result = 8
Result = 512
Result = -4
Result = 4
Result = 1
Result = 1.5
Result = -1
Result = -6
Result = -7
Result = 30
```

PROGRAM 4:
Lex file:

```
%{
#include "decl.tab.h"
#include <stdlib.h>
#include <string.h>
```

```
/* Simple typedef name table */
#define MAX_TYPENAMES 1024
static char* type_names[MAX_TYPENAMES];
static int type_name_count = 0;
static int typedef_mode = 0; /* set when scanning a typedef declaration */

static int is_typedef_name(const char* s) {
        for (int i = 0; i < type_name_count; ++i) {
                if (strcmp(type_names[i], s) == 0) return 1;
        }
        return 0;
}

static void add_typedef_name(const char* s) {
        if (is_typedef_name(s)) return;
        if (type_name_count < MAX_TYPENAMES) {
                type_names[type_name_count++] = _strdup(s);
        }
}

int yywrap(void){ return 1; }
%}


%%

[a-zA-Z_][a-zA-Z0-9_]*    {
        /* Keywords for basic and modifier types */
        if (strcmp(yytext, "int") == 0) return INT;
        if (strcmp(yytext, "float") == 0) return FLOAT;
        if (strcmp(yytext, "double") == 0) return DOUBLE;
        if (strcmp(yytext, "char") == 0) return CHAR;
        if (strcmp(yytext, "short") == 0) return SHORT;
        if (strcmp(yytext, "long") == 0) return LONG;
        if (strcmp(yytext, "signed") == 0) return SIGNED;
        if (strcmp(yytext, "unsigned") == 0) return UNSIGNED;
        if (strcmp(yytext, "void") == 0) return VOID;
        if (strcmp(yytext, "const") == 0) return CONST;
        if (strcmp(yytext, "volatile") == 0) return VOLATILE;
        if (strcmp(yytext, "typedef") == 0) { typedef_mode = 1; return TYPEDEF; }
        if (strcmp(yytext, "static") == 0) return STATIC;
        if (strcmp(yytext, "extern") == 0) return EXTERN;
        if (strcmp(yytext, "register") == 0) return REGISTER;
        if (strcmp(yytext, "struct") == 0) return STRUCT;
        if (strcmp(yytext, "union") == 0) return UNION;
        if (strcmp(yytext, "enum") == 0) return ENUM;

        /* typedef-name handling */
        if (typedef_mode) { add_typedef_name(yytext); return TYPE_NAME; }
        if (is_typedef_name(yytext)) return TYPE_NAME;
        return ID;
}
","         { return COMMA; }
";"         { typedef_mode = 0; return SEMICOLON; }
"="         { return ASSIGN; }
"*"         { return ASTERISK; }
"["         { return LBRACKET; }
"]"         { return RBRACKET; }
"("         { return LPAREN; }
")"         { return RPAREN; }
"{"         { return LBRACE; }
"}"         { return RBRACE; }
        [0-9]+\.[0-9]*([eE][+-]?[0-9]+)?   { return FLOATCONST; }
        \.[0-9]+([eE][+-]?[0-9]+)?       { return FLOATCONST; }
        [0-9]+([eE][+-]?[0-9]+)          { return FLOATCONST; }

[0-9]+                      { return NUMBER; }

        \"([^\\\"\n]|\\.)*\"         { return STRINGLIT; }
'(\\.|[^\\\n])'             { return CHARCONST; }

[ \t\n\r]+    { /* skip whitespace */ }
.           { return INVALID; }
%%
```

YACC file:

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
void yyerror(const char *s);
static int g_error = 0;
%}
```

```
%union {
  int ival; /* 1 if declarator denotes a function, else 0 */
}

%token VOID CHAR SHORT INT LONG FLOAT DOUBLE SIGNED UNSIGNED
%token CONST VOLATILE TYPEDEF STATIC EXTERN REGISTER
%token STRUCT UNION ENUM
%token TYPE_NAME
%token ID NUMBER FLOATCONST CHARCONST STRINGLIT
%token COMMA SEMICOLON ASSIGN ASTERISK LBRACKET RBRACKET LPAREN
RPAREN LBRACE RBRACE INVALID

%type <ival> declarator direct_declarator

%%

decl
    : decl_specifiers init_declarator_list SEMICOLON    { if (!g_error) printf("Valid
declaration\n"); }
  | decl_specifiers SEMICOLON                { if (!g_error) printf("Valid declaration\n");
}
  ;

/* declaration specifiers: storage class | type specifiers | type qualifiers */
decl_specifiers
  : decl_specifiers decl_specifier
  | decl_specifier
  ;

decl_specifier
  : storage_class_specifier
  | type_qualifier
  | type_token
  | struct_or_union_specifier
  | enum_specifier
  ;

storage_class_specifier
  : TYPEDEF
  | EXTERN
  | STATIC
  | REGISTER
```

```
   ;

type_token
   : VOID
   | CHAR
   | SHORT
   | INT
   | LONG
   | FLOAT
   | DOUBLE
   | SIGNED
   | UNSIGNED
   | TYPE_NAME
   ;

/* struct/union definitions or tags */
struct_or_union_specifier
   : STRUCT opt_id LBRACE member_declaration_list_opt RBRACE
   | UNION  opt_id LBRACE member_declaration_list_opt RBRACE
   ;

opt_id
   : /* empty */
   | ID
   ;

member_declaration_list_opt
   : /* empty */
   | member_declaration_list
   ;

member_declaration_list
   : member_declaration
   | member_declaration_list member_declaration
   ;

member_declaration
   : decl_specifiers member_declarator_list_opt SEMICOLON
   ;

member_declarator_list_opt
   : /* empty */
```

```
  | member_declarator_list
  ;

member_declarator_list
  : member_declarator
  | member_declarator_list COMMA member_declarator
  ;

member_declarator
  : declarator
  ;

/* enum definitions or tags */
enum_specifier
  : ENUM opt_id LBRACE enumerator_list_opt RBRACE
  | ENUM opt_id LBRACE enumerator_list COMMA RBRACE
  | ENUM ID
  ;

enumerator_list_opt
  : /* empty */
  | enumerator_list
  ;

enumerator_list
  : enumerator
  | enumerator_list COMMA enumerator
  ;

enumerator
  : ID
  | ID ASSIGN const_expr_opt
  ;

type_qualifier
  : CONST
  | VOLATILE
  ;

init_declarator_list
  : init_declarator
  | init_declarator_list COMMA init_declarator
```

```
    ;

/* Disallow initializers for function declarators by only allowing assignment
   to non-function declarators */
init_declarator
    : declarator
     | declarator ASSIGN initializer   { if ($1) yyerror("function declarator cannot be
initialized"); }
    ;

/* pointer and array declarators like: *p, **q, a[10], b[3][4] */
declarator
    : pointer_opt direct_declarator   { $$ = $2; }
    ;

pointer_opt
    : /* empty */
    | pointer
    ;

/* pointer with optional qualifiers after each '*' */
pointer
    : ASTERISK type_qual_list_opt
    | ASTERISK type_qual_list_opt pointer
    ;

type_qual_list_opt
    : /* empty */
    | type_qual_list
    ;

type_qual_list
    : type_qualifier
    | type_qual_list type_qualifier
    ;

direct_declarator
    : ID                                  { $$ = 0; }
    | LPAREN declarator RPAREN                 { $$ = $2; }
    | direct_declarator LBRACKET const_expr_opt RBRACKET   { $$ = $1; }
    | direct_declarator LPAREN parameter_list_opt RPAREN   { $$ = 1; }
    ;
```

```
const_expr_opt
    : /* empty */
    | NUMBER
    | ID
    ;

/* keep initializer simple (constants or identifiers) */
initializer
    : NUMBER
    | FLOATCONST
    | CHARCONST
    | STRINGLIT
    | ID
    ;

/* parameters: simplified */
parameter_list_opt
    : /* empty */
    | parameter_list
    ;

parameter_list
    : parameter_declaration
    | parameter_list COMMA parameter_declaration
    ;

parameter_declaration
    : decl_specifiers declarator
    | decl_specifiers
    ;

%%
void yyerror(const char *s) {
    g_error = 1;
    printf("Invalid declaration\n");
}

int main() {
    printf("Enter declaration: ");
    g_error = 0;
    yyparse();
    return 0;
}
```

all.tst:

```
x = 3 + 4; y = 3.14; c = 'A'; s = "hi"; b = true;
x = 2 ^ 3;
y = 2 ^ 3 ^ 2;
z = -2 ^ 2;
a = ( -2 ) ^ 2;
b = 5 % 2;
c = 5.5 % 2;
d = 5 % 0;
x = 1 + 2;
y = 2 * (3 + 4);
z = -5 + 10 / 2;
s = "hello";
c = 'Z';
b = false;
d=3.0 + 2.1;
a = 10 / 0; b = 1;
```

OUTPUT 4:

```
PS C:\Users\Joseph\Desktop\compiler design\expt7> Get-Content ".\tests\decl\all.tst" | ForEach-Object { $_ | .\decl.exe 2>$null } | Out-File -Encoding ascii ".\tests\decl\all.out"
PS C:\Users\Joseph\Desktop\compiler design\expt7> C:\Windows\System32\fc.exe ".\tests\assign\all.actual" ".\tests\assign\all.out"
Comparing files .\TESTS\ASSIGN\all.actual and .\TESTS\ASSIGN\ALL.OUT
FC: no differences encountered
```

all.out:

```
Enter assignments like: x = 3 + 4; y = 3.14; c = 'A'; s = "hi"; b = true;
Ctrl+Z then Enter to quit.
x = 7
y = 3.14
c = 'A'
s = "hi"
b = true
x = 8
y = 512
z = -4
```

```
a = 4
b = 1
c = 1.5
d = 0
x = 3
y = 14
z = 0
s = "hello"
c = 'Z'
b = false
d = 5.1
a = 0
b = 1
```

CONCLUSION:

Across expression sub expt and binary expression sub expt we put language theory into practice—clean tokenization, well-chosen precedence/associativity, and a careful split between unary and binary operators—while declare sub expt forced precise handling of C-style types (pointers, arrays, function pointers, qualifiers, initializers) and assignment sub expt tied it together with predictable l-value/r-value semantics, whitespace tolerance, and safe evaluation (e.g., division-by-zero guards). The curated tests and PowerShell runners created a tight feedback loop that surfaced shift/reduce pitfalls early and kept changes honest. The big takeaways are to separate lexing from parsing, favor clarity over clever grammar tricks, and keep semantic actions minimal; natural next steps are an AST, symbol tables, type checking, and constant folding en route to code generation.