<u>LAB 10</u> **<u>INTRODUCTION TO BISON</u>**

ADITYA AGARWAL
220905106
ROLL NO. 14

Q - Write a bison program,
1. To check a valid declaration statement.
2. To check a valid decision making statements.
3. To evaluate an arithmetic expression involving operations +,-,* and /.
4. To validate a simple calculator using postfix notation. The grammar rules are as follows –
input → input line | ε
line → '\n' | exp '\n'
exp → num | exp exp '+'
| exp exp '-'
| exp exp '*'
| exp exp '/'
| exp exp '^'
| exp 'n'

<u>SOL -</u>

<u>BISON.Y</u>

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

/* Function prototypes */
void yyerror(char *s);
int yylex(void);
extern char* yytext;
%}

/* Token declarations */
%token INT FLOAT CHAR DOUBLE
%token IF ELSE WHILE FOR DO RETURN
%token NUMBER ID STRING
%token SEMICOLON COMMA
%token EQ NE LT GT LE GE AND OR NOT
```

```
%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
%token POSTFIX_MODE INFIX_MODE DECLARATION_MODE
DECISION_MODE
%token NEWLINE

/* Operator precedence and associativity for infix mode */
%left '+' '-'
%left '*' '/'
%right '^'
%right UMINUS

/* Type for the values being calculated */
%union {
   double val;
   char *str;
}

%type <val> expr_infix expr_postfix number
%type <str> id

%%
/* Starting rule */
start   : NEWLINE
       | command
       ;

command : DECLARATION_MODE declaration { printf("Valid declaration
statement\n"); }
       | DECISION_MODE decision { printf("Valid decision making statement\n"); }
       | INFIX_MODE expr_infix NEWLINE { printf("Infix expression result: %.2f\n",
$2); }
       | POSTFIX_MODE expr_postfix NEWLINE { printf("Postfix expression result:
%.2f\n", $2); }
       ;

/* Declaration statement validation */
declaration : type var_list SEMICOLON
           ;

type    : INT
       | FLOAT
       | CHAR
       | DOUBLE
       ;

var_list : variable
        | var_list COMMA variable
        ;
```

```
variable : id
        | id LBRACKET NUMBER RBRACKET  /* Array declaration */
        ;

id      : ID { $$ = strdup(yytext); }
       ;

/* Decision making statement validation */
decision : if_stmt
        | while_stmt
        | for_stmt
        | do_while_stmt
        ;

if_stmt : IF LPAREN condition RPAREN block
       | IF LPAREN condition RPAREN block ELSE block
       ;

while_stmt : WHILE LPAREN condition RPAREN block
          ;

for_stmt : FOR LPAREN for_init condition SEMICOLON expression RPAREN block
        ;

for_init : assign SEMICOLON
        | SEMICOLON
        ;

do_while_stmt : DO block WHILE LPAREN condition RPAREN SEMICOLON
             ;

block   : LBRACE statements RBRACE
       | statement
       ;

statements : statement
          | statements statement
          | /* empty */
          ;

statement : expression SEMICOLON
         | decision
         | declaration
         | RETURN expression SEMICOLON
         | SEMICOLON
         ;
```

```
condition : expression
         | expression rel_op expression
         | expression log_op expression
         | NOT expression
         ;

assign  : id '=' expression
       ;

rel_op  : EQ | NE | LT | GT | LE | GE
       ;

log_op  : AND | OR
       ;

/* Common expression rules */
expression : expr_infix
         ;

/* Infix expression evaluation */
expr_infix : number              { $$ = $1; }
         | expr_infix '+' expr_infix { $$ = $1 + $3; }
         | expr_infix '-' expr_infix { $$ = $1 - $3; }
         | expr_infix '*' expr_infix { $$ = $1 * $3; }
         | expr_infix '/' expr_infix {
                               if ($3 == 0) {
                                   yyerror("Division by zero");
                                   $$ = 0;
                               } else {
                                   $$ = $1 / $3;
                               }
                             }
         | expr_infix '^' expr_infix { $$ = pow($1, $3); }
         | '-' expr_infix %prec UMINUS { $$ = -$2; }
         | LPAREN expr_infix RPAREN { $$ = $2; }
         ;

/* Postfix expression evaluation */
expr_postfix : number              { $$ = $1; }
           | expr_postfix expr_postfix '+' { $$ = $1 + $2; }
           | expr_postfix expr_postfix '-' { $$ = $1 - $2; }
           | expr_postfix expr_postfix '*' { $$ = $1 * $2; }
           | expr_postfix expr_postfix '/' {
                                   if ($2 == 0) {
                                       yyerror("Division by zero");
                                       $$ = 0;
                                   } else {
                                       $$ = $1 / $2;
```

```
                                      }
                                    }
            | expr_postfix expr_postfix '^' { $$ = pow($1, $2); }
            | expr_postfix 'n'            { $$ = -$1; }  /* Unary negation */
            ;

number  : NUMBER { $$ = atof(yytext); }
        ;

%%

void yyerror(char *s) {
    printf("Error: %s\n", s);
}

int main() {
    printf("Multi-purpose Parser\n");
    printf("Commands:\n");
    printf("1. To validate declaration: 'decl' followed by a declaration statement\n");
    printf("2. To validate decision making: 'decision' followed by a control structure\
n");
    printf("3. To evaluate infix expression: 'infix' followed by an expression\n");
    printf("4. To evaluate postfix expression: 'postfix' followed by an expression\n");
    printf("Example: 'infix 2 + 3 * 4'\n");

    yyparse();
    return 0;
}
```

BISON.L

```
%{
#include "bison.tab.h"
#include <string.h>
#include <stdlib.h>
extern YYSTYPE yylval;
char* yytext;
%}

%%
"decl"      { return DECLARATION_MODE; }
"decision"  { return DECISION_MODE; }
"infix"     { return INFIX_MODE; }
"postfix"   { return POSTFIX_MODE; }

"int"       { return INT; }
```

```
"float"     { return FLOAT; }
"char"      { return CHAR; }
"double"    { return DOUBLE; }

"if"        { return IF; }
"else"      { return ELSE; }
"while"     { return WHILE; }
"for"       { return FOR; }
"do"        { return DO; }
"return"    { return RETURN; }

"=="        { return EQ; }
"!="        { return NE; }
"<"         { return LT; }
">"         { return GT; }
"<="        { return LE; }
">="        { return GE; }
"&&"        { return AND; }
"||"        { return OR; }
"!"         { return NOT; }

"("         { return LPAREN; }
")"         { return RPAREN; }
"{"         { return LBRACE; }
"}"         { return RBRACE; }
"["         { return LBRACKET; }
"]"         { return RBRACKET; }
";"         { return SEMICOLON; }
","         { return COMMA; }

[0-9]+(\.[0-9]+)?    { yylval.val = atof(yytext); return NUMBER; }
[a-zA-Z][a-zA-Z0-9_]*  { yylval.str = strdup(yytext); return ID; }
\"[^\"]*\"             { yylval.str = strdup(yytext); return STRING; }

[ \t]      { /* ignore whitespace */ }
\n         { return NEWLINE; }
.          { return yytext[0]; }
%%

int yywrap() {
    return 1;
}
```

COMPILE.SH (OPTIONAL)

```
#!/bin/bash
```

```
# Remove old files
rm -f lex.yy.c bison.tab.c bison.tab.h parser

# Generate parser files with bison
bison -d bison.y

# Generate lexer with flex
flex bison.l

# Compile everything
gcc -Wall -o parser lex.yy.c bison.tab.c -lm

# Test if compilation was successful
if [ -f parser ]; then
   echo "Compilation successful. Run the parser with ./parser"
else
   echo "Compilation failed."
fi
```

EXECUTION COMMMANDS

```
1. bison -d bison.y
2. flex bison.l
3. gcc -o parser lex.yy.c bison.tab.c -lm
4. ./parser
```

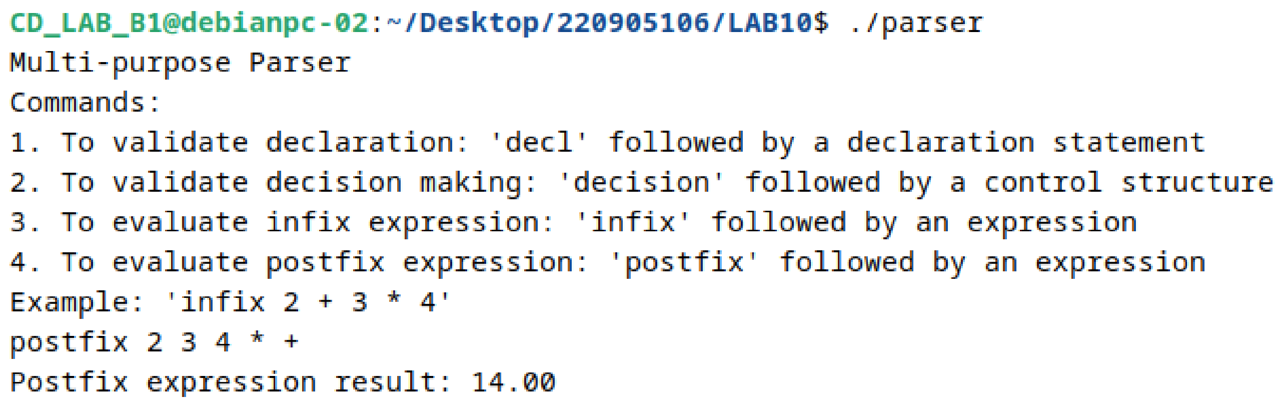
```
Multi-purpose Parser
Commands:
1. To validate declaration: 'decl' followed by a declaration statement
2. To validate decision making: 'decision' followed by a control structure
3. To evaluate infix expression: 'infix' followed by an expression
4. To evaluate postfix expression: 'postfix' followed by an expression
Example: 'infix 2 + 3 * 4'

> decl int a, b;
Valid declaration statement

> decision if (a > b) { return 1; } else { return 0; }
Valid decision making statement

> infix 2 + 3 * 4
Infix expression result: 14.00

> postfix 2 3 4 * +
Postfix expression result: 14.00
```



```
CD_LAB_B1@debianpc-02:~/Desktop/220905106/LAB10$ ./parser
Multi-purpose Parser
Commands:
1. To validate declaration: 'decl' followed by a declaration statement
2. To validate decision making: 'decision' followed by a control structure
3. To evaluate infix expression: 'infix' followed by an expression
4. To evaluate postfix expression: 'postfix' followed by an expression
Example: 'infix 2 + 3 * 4'
postfix 2 3 4 * +
Postfix expression result: 14.00
```