CSE 358 - ASSIGNMENT 8

220002018 - Arnav Jain

Question 1

q1.l:

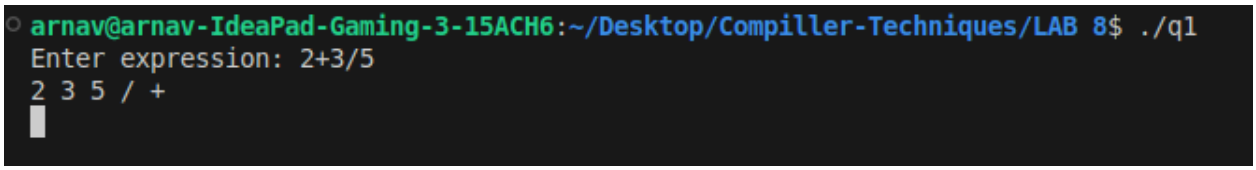
q1.y:

q1.y

You, 1 second ago | 1 author (You)

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

%union { int num; }
%token <num> NUMBER
%type <num> expr
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
input:    /* empty */
        | input expr '\n'  { printf("\n"); }
        ;

expr:    NUMBER           { printf("%d ", $1); }
        | '(' expr ')'    { $$ = $2; }
        | expr '+' expr   { printf("+ "); }
        | expr '-' expr   { printf("- "); }
        | expr '*' expr   { printf("* "); }
        | expr '/' expr   { printf("/ "); }
        | '-' expr %prec UMINUS { printf("- "); $$ = $2; }
        ;
%%

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

int main() {
    printf("Enter expression: ");
    fflush(stdout);  // Force prompt display
    yyparse();
    return 0;
}
```

Output:


arnav@arnav-IdeaPad-Gaming-3-15ACH6:~/Desktop/Compiller-Techniques/LAB 8$ ./q1
Enter expression: 2+3/5
2 3 5 / +

Question 2

q2.l

```
q2.l
You, 2 days ago | 1 author (You)
%{
#include "q2.tab.h"
void yyerror(char *);  // Explicit error declaration
extern int count;       // Access YACC's counter
%}

%%
0               { yylval.digit = 0; return BINARY_DIGIT; }
1               { yylval.digit = 1; return BINARY_DIGIT; }
\.              { return DOT; }
[ \t\n]         ;  // Ignore whitespace
.               { yyerror("Invalid binary character"); }
%%

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

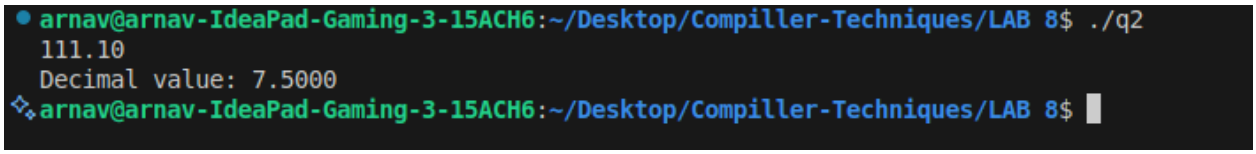
Q2.y

q2.y

You, 2 days ago | 1 author (You)

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

int count = 0;  // Track fractional digit position
%}

%union {
    int digit;
    double val;
}

%token <digit> BINARY_DIGIT
%token DOT
%type <val> number integer_part fractional_part

%%
number:    integer_part DOT fractional_part {
                $$ = $1 + $3;
                printf("Decimal value: %.4f\n", $$);
            }
        ;

integer_part:    /* empty */       { $$ = 0.0; }
            | integer_part BINARY_DIGIT { $$ = $1 * 2 + $2; }
            ;

fractional_part: /* empty */       { $$ = 0.0; count = 0; }
            | fractional_part BINARY_DIGIT {
                count++;
                $$ = $1 + $2 * pow(2, -count);
            }
            ;
%%

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

int main() {
    yyparse();
    return 0;
}
```

Output


arnav@arnav-IdeaPad-Gaming-3-15ACH6:~/Desktop/Compiller-Techniques/LAB 8$ ./q2
111.10
Decimal value: 7.5000
arnav@arnav-IdeaPad-Gaming-3-15ACH6:~/Desktop/Compiller-Techniques/LAB 8$

Question 3

Question 3 : Error Handling and Recovery in Compiler Design

Error Handling

Error Handling in compiler design identifies and reports errors in source code while maintaining the compilation process. The errors include:

i) Lexical Errors : incorrect grammar (eg missing semicolon like int x = 10)

ii) Semantic Errors : Meaning errors (eg. undeclared variable y used)

iii) Runtime Errors : Errors during program execution (eg. division by zero)

iv) Linking Errors : Issues during code linking (eg missing library)

Error Recovery

Methods for recovery after detecting errors :

i) Panic Mode : Discards input until a synchronizing token is found (ie skipping to the next semicolon)

ii) Phrase level : Tries to replace or insert tokens to continue parsing (eg changing { to)).

iii) Error Productions : Allows predefined error handling within grammar rules.

iv) Contextual Recovery : Analyse surrounding code to decide how to recover

v) Backtracking : Tries alternate parsing strategies when one path fails.

Question 4

Question 4 : Detailed Understanding of Compiler Design and Phase

A compiler translates high level source code into machine code or intermediate code to execute the program

Compiler Phases:

i) Lexical Analysis : input → source code
output → Tokens (eg. Keywords, operators)

ii) Syntax Analysis: input → Tokens
output → Parse Tree (structure showing program's syntax)

---

iii) Semantic Analysis: input → Parse Tree
Output → Intermediate Code representation (eg: Three-order Code)

iv) Optimization: Input → Intermediate Code
Output → Optimized Intermediate Code

v) Code Generation: Input → Optimised Intermediate Code
Output → Machine Assembly code

vi) Code Optimizer : (optional) Input → Machine Code
Output → Machine Optimized Code

vii) Code Emission and Linking : Input → Final Code
Output → Executable file.

GitHub

https://github.com/arnavjain2710/Compiller-Techniques/tree/main/LAB%208