Compiler Design Assignment

Name: Suchitra Shankar
Class: 6J
SRN: PES2UG23CS608

---

1. Objective

The objective of this assignment was to implement a lexical analyzer using **Flex** and a syntax analyzer using **Bison (Yacc)** to validate the syntax of a simplified subset of the C programming language.

The analyzer reads C-like source code from standard input and determines whether the syntax is valid according to the defined grammar.

2. Lexical Analyzer Design (Flex)

The lexical analyzer scans the input program and converts it into tokens which are then passed to the parser.

Tokens Recognized

The lexer recognizes:

- **Data types:** `int`, `float`, `char`, `double`
- **Control keywords:** `if`, `else`, `while`, `do`, `for`, `switch`, `case`, `default`, `break`
- **Program-level tokens:** `main`, `#include`
- **Identifiers and numeric constants**
- **String literals and header files** (e.g., `<stdio.h>`)
- **Operators:**
  `+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&&`, `||`, `!`, `++`, `--`, `=`
- **Delimiters:**
  `( ) { } [ ] ; ,`

Regular definitions such as `digit`, `letter`, `id`, and `number` were used for clarity and maintainability. Whitespace and comments are ignored during tokenization.
Line numbers are tracked using `yylineno` to support accurate error reporting in the parser. The lexer returns token definitions declared in `parser.tab.h`, ensuring correct integration between Flex and Bison.

3. Syntax Analyzer Design (Bison)

The parser validates whether the sequence of tokens satisfies the defined grammar rules.

The grammar handles:

- Variable declarations (including multiple declarators and array dimensions)
- Assignment statements
- Arithmetic, relational, and logical expressions
- Nested blocks using `{ ... }`
- Control structures:
  - `if`
  - `if-else`
  - `while`
  - `for`
  - `do-while`
  - `switch-case-default`
  - `break`
- Optional `#include` directives and `main(...)` function structure

This ensures that a reasonably broad subset of C syntax can be validated.

4. Operator Precedence and Associativity

The classic dangling-else ambiguity is resolved using:

```
25   %nonassoc IFX
26   %nonassoc ELSE
27
```

This ensures that each `else` is associated with the nearest unmatched `if`, removing ambiguity in nested conditional statements.

5. Error Handling

Syntax errors are handled using the `yyerror()` function.

Errors are reported in the format:

Error: <error>, line number: <line no>,token: <token>

```
❯ ./a.out < wrong.c
Error: syntax error, line number: 14,token: c
Error: syntax error, line number: 16,token: default
Error: syntax error, line number: 18,token: }
```

This allows the parser to continue processing input and detect multiple syntax errors instead of terminating immediately.

If no errors are found, the program prints:

Valid syntax

```
❯ ./a.out < input.c
Valid syntax
```

6. Build Automation

makefile.mk

```
BISON := bison
FLEX  := flex
CC    := gcc
CFLAGS := -Wall -g
LDFLAGS := -lfl

TARGET := a.out
BISON_SRC := parser.y
FLEX_SRC  := lexer.l
INPUT := wrong.c

.PHONY: all clean run

all: $(TARGET)

$(TARGET): parser.tab.c lex.yy.c
    $(CC) $(CFLAGS) -o $@ parser.tab.c lex.yy.c $(LDFLAGS)


parser.tab.c parser.tab.h: $(BISON_SRC)
    $(BISON) -d $<

lex.yy.c: $(FLEX_SRC) parser.tab.h
    $(FLEX) $<


run: $(TARGET)
    ./$(TARGET) < $(INPUT)

clean:
    rm -f $(TARGET) parser.tab.c parser.tab.h lex.yy.c
```

A `makefile.mk` was created to automate:

- Generating parser files using Bison (`parser.tab.c`, `parser.tab.h`)
- Generating lexer files using Flex (`lex.yy.c`)
- Compiling and linking the analyzer
- Running with test input

- Cleaning generated files

This ensures structured and reproducible builds.

7. Conclusion

This assignment demonstrates the successful integration of Flex and Bison to implement a working lexical and syntax analyzer for a C-like language subset.

The implementation supports:

- Comprehensive token recognition
- Grammar-based syntax validation
- Multiple control structures
- Expression parsing
- Structured error reporting with basic recovery

Overall, the project reinforces key compiler design concepts including lexical analysis, context-free grammar parsing, ambiguity resolution, and error handling.

Lexer.l

lexer.l

```
%{
#include <stdio.h>
#include "parser.tab.h"
void yyerror(const char *s);

static void count_newlines(const char *text) {
    const char *p = text;
    while (*p) {
        if (*p == '\n') {
            yylineno++;
        }
        p++;
    }
}
%}
%option noinput nounput

digit      [0-9]
letter     [a-zA-Z]
id         ({letter}|_)({letter}|{digit}|_)*
number     {digit}+
header     \<{id}\.h\>
string     \"([^\"\n])*\"

%%


"int"         { return INT; }
"char"        { return CHAR; }
"float"       { return FLOAT; }
"double"      { return DOUBLE; }
"while"       { return WHILE; }
"for"         { return FOR; }
"switch"      { return SWITCH; }
"case"        { return CASE; }
"default"     { return DEFAULT; }
"break"       { return BREAK; }
"do"          { return DO; }
"if"          { return IF; }
"else"        { return ELSE; }
"main"        { return MAIN; }
"#include"    { return INCLUDE; }

"//".*        ;


"/*"([^*]|\*+[^*/])*\*+"/" { count_newlines(yytext); }

"=="          { return EQCOMP; }
">="          { return GREATEREQ; }
"<="          { return LESSEREQ; }
"!="          { return NOTEQ; }
"&&"          { return ANDAND; }
"||"          { return ORor; }
```

lexer.l

```
53   "&&"          { return ANDAND; }
54   "||"          { return OROR; }
55   "++"          { return INC; }
56   "--"          { return DEC; }
57
58   ">"           { return *yytext; }
59   "<"           { return *yytext; }
60   "="           { return *yytext; }
61   "!"           { return *yytext; }
62   "+"           { return *yytext; }
63   "-"           { return *yytext; }
64   "*"           { return *yytext; }
65   "/"           { return *yytext; }
66
67
68   "("           { return *yytext; }
69   ")"           { return *yytext; }
70   "["           { return *yytext; }
71   "]"           { return *yytext; }
72   "{"           { return *yytext; }
73   "}"           { return *yytext; }
74   ";"           { return *yytext; }
75   ","           { return *yytext; }
76
77   {string}      { return STRLITERAL; }
78   {number}      { return NUM; }
79
80
81   {header}      { return HEADER; }
82   {id}          { return ID; }
83
84
85   "\n"          { yylineno++; }
86   [ \t\r]+      ;
87
88
89   .             { return *yytext; }
90
91   %%
92
93
94   int yywrap() {
95       return 1;
96   }
```

Parser.y

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

extern int yylex();
extern int yylineno;
extern char *yytext;
int had_error = 0;

void yyerror(const char *s);
%}

/* ----------- TOKENS (must match lexer) ----------- */

%token INT FLOAT CHAR DOUBLE
%token IF ELSE DO WHILE FOR
%token SWITCH CASE DEFAULT BREAK
%token ID NUM
%token MAIN INCLUDE INC DEC STRLITERAL HEADER
%token EQCOMP NOTEQ GREATEREQ LESSEREQ
%token ANDAND OROR

/* Precedence */
%nonassoc IFX
%nonassoc ELSE

%start program

%%

program
	: include_list_opt external_list
	;

external_list
	: external_list external
	| /* empty */
	;

external
	: statement
	| main_function
	;
```

```
include_list_opt
        : /* empty */
        | include_list
        ;

include_list
        : include_list include_stmt
        | include_stmt
        ;

include_stmt
        : INCLUDE HEADER
        | INCLUDE STRLITERAL
        ;

main_function
        : type MAIN '(' parameter_list_opt ')' block
        ;

parameter_list_opt
        : /* empty */
        | parameter_list
        ;

parameter_list
        : parameter_list ',' parameter
        | parameter
        ;

parameter
        : type ID
        | type
        ;

stmt_list
        : stmt_list statement
        | /* empty */
        ;

statement
        : declaration ';'
        | expression ';'
        | ';'
        | if_stmt
        | while_stmt
```

```
        | for_stmt
        | do_while_stmt
        | switch_stmt
        | BREAK ';'
        | block
        | error ';' { yyerrok; }
        ;

block
        : '{' stmt_list '}'
        ;

declaration
        : type init_declarator_list
        ;

type
        : INT
        | FLOAT
        | CHAR
        | DOUBLE
        ;

init_declarator_list
        : init_declarator_list ',' init_declarator
        | init_declarator
        ;

init_declarator
        : ID array_dims_opt initializer_opt
        ;

array_dims_opt
        : /* empty */
        | array_dims_opt '[' NUM ']'
        ;

initializer_opt
        : /* empty */
        | '=' expression
        ;

if_stmt
        : IF '(' expression ')' statement %prec IFX
        | IF '(' expression ')' statement ELSE statement
        ;
```

```
while_stmt
	: WHILE '(' expression ')' statement
	;

for_stmt
	: FOR '(' opt_for_expr_list ';' opt_for_condition ';' opt_for_expr_list ')'
statement
	;

do_while_stmt
	: DO statement WHILE '(' expression ')' ';'
	;

opt_for_expr_list
	: /* empty */
	| for_expr_list
	;

for_expr_list
	: for_expr_list ',' assignment_expression
	| assignment_expression
	;

opt_for_condition
	: /* empty */
	| expression
	;

switch_stmt
	: SWITCH '(' expression ')' '{' switch_sections_opt '}'
	;

switch_sections_opt
	: /* empty */
	| switch_sections
	;

switch_sections
	: switch_sections switch_section
	| switch_section
	;

switch_section
	: CASE NUM ':' stmt_list
	| DEFAULT ':' stmt_list
```

```
        ;

expression
        : assignment_expression
        ;

assignment_expression
        : ID '=' assignment_expression
        | logical_or_expression
        ;

logical_or_expression
        : logical_or_expression OROR logical_and_expression
        | logical_and_expression
        ;

logical_and_expression
        : logical_and_expression ANDAND equality_expression
        | equality_expression
        ;

equality_expression
        : equality_expression EQCOMP relational_expression
        | equality_expression NOTEQ relational_expression
        | relational_expression
        ;

relational_expression
        : relational_expression '<' additive_expression
        | relational_expression '>' additive_expression
        | relational_expression GREATEREQ additive_expression
        | relational_expression LESSEREQ additive_expression
        | additive_expression
        ;

additive_expression
        : additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression
        | multiplicative_expression
        ;

multiplicative_expression
        : multiplicative_expression '*' unary_expression
        | multiplicative_expression '/' unary_expression
        | unary_expression
        ;
```

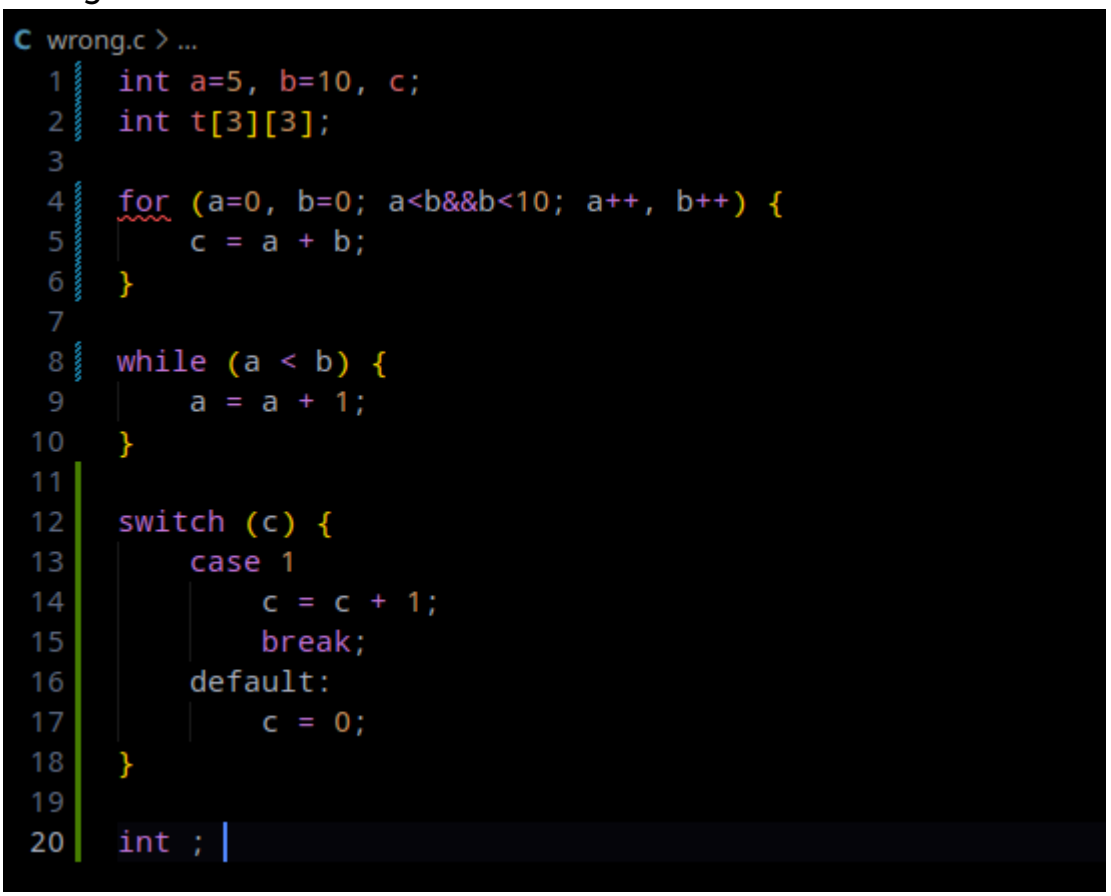
```
unary_expression
        : '!' unary_expression
        | '+' unary_expression
        | '-' unary_expression
        | postfix_expression
        ;

postfix_expression
        : postfix_expression INC
        | postfix_expression DEC
        | primary_expression
        ;

primary_expression
        : '(' expression ')'
        | ID
        | NUM
        ;

%%

void yyerror(const char *s)
{
    had_error = 1;
    fprintf(stderr,
            "Error: %s, line number: %d,token: %s\n",
            s, yylineno, yytext);
}

int main()
{
    if (yyparse() == 0 && !had_error)
        printf("Valid syntax\n");
    return 0;
}
```

Input.c

input.c > ...

```c
int a=5, b=10, c, d=10;
int i=0, j=0, p=5, q=5;
int arr1[15];
int arr2[10][10];
int arr3[1][2][3][4];
int m[4][4], n[5];
float x;
double y;
char ch;

c = a + b * 2;
x = 3;
y = 4;
ch = 1;

if (c > 10) {
    x = x + 1;
} else {
    x = x - 1;
}

while (a < b) {
    a = a + 1;
    b = b - 1;
}

for(i=0, j=0; i<p&&j<q; i++, j++) {
    c = c + i;
}

do {
    a = a + 1;
    b = b - 1;
} while (a < b);

switch (c) {
    case 1:
        c = c + 1;
        break;
    case 2:
        c = c - 1;
        break;
    default:
        c = 0;
}
```

Wrong.c


wrong.c > ...
1 int a=5, b=10, c;
2 int t[3][3];
3
4 for (a=0, b=0; a<b&&b<10; a++, b++) {
5 c = a + b;
6 }
7
8 while (a < b) {
9 a = a + 1;
10 }
11
12 switch (c) {
13 case 1
14 c = c + 1;
15 break;
16 default:
17 c = 0;
18 }
19
20 int ;