Compiler Design Lab 1

Team members

Roll Number	Name
106120051	Kiran Srinivasan
106120053	Krishnadas Nair
106120087	Pranav Kamath
106120033	Donald Xavier Anto

Aim

To create develop components of own programming language using regular expressions and generating a lexical analyzer using Lex tool

Code

lexer.l

```
/* DEFINITIONS */

%{
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include "colors.h"

#define TOK(t) Token(t, string(yytext, yyleng))

using namespace std;

extern FILE* yyin;

enum TokenType {
  /* datatypes */

  BOOL, INT, FLOAT, CHAR, STRING, DATATYPE, VOID,

  /* punctuation */

  LPAREN, RPAREN, LBRACE, RBRACE, LBOX, RBOX, SEMICOLON, COLON, DOT, COMMA,
DOLLAR, HASH, TILDE, ARROW,

  /* loops */
  WITH, LOOP, UNTIL, UPDATE, EXIT, SKIP,

  /* conditional */

  IF, ELSE,

  /* others */

  RETURN, ID, ALLOC, _NULL,

  /* literals */

  TRUE, FALSE, INTLIT, FLOATLIT, CHARLIT, STRINGLIT,


  /* binary operations */

  ASSIGN, EQUALS, NEQUALS, LT, GT, LTE, GTE, PLUS, MINUS, MUL, DIV, MOD,
AND, OR, PLUSEQUALS, MINUSEQUALS, MULEQUALS, DIVEQUALS, MODEQUALS,

  /* unary operators */

  NOT, INCR, DECR

};

struct Loc {
  int col = 0, line = 0;
} l;

#define INC l.col += yyleng

struct Token {
  TokenType type;
  string lexeme;
  Loc location;
  Token(TokenType type , string lexeme) {
    this->type = type;
    this->lexeme = lexeme;
    location = l;
  }
};

vector<Token> tokens;
unordered_map<TokenType, string> tokenMap = {
  /*  datatypes */

  {BOOL, "BOOL"},
  {INT, "INT"},
  {FLOAT, "FLOAT"},
  {CHAR, "CHAR"},
  {STRING, "STRING"},
  {DATATYPE, "DATATYPE"},
  {VOID, "VOID"},
  /* punctuation */

  {LPAREN, "LPAREN"},
  {RPAREN, "RPAREN"},
  {LBRACE, "LBRACE"},
  {RBRACE, "RBRACE"},
  {LBOX, "LBOX"},
  {RBOX, "RBOX"},
  {SEMICOLON,"SEMICOLON"},
  {COLON, "COLON"},
  {DOT, "DOT"},
  {COMMA, "COMMA"},
  {DOLLAR, "DOLLAR"},
  {HASH, "HASH"},
  {TILDE, "TILDE"},
  {ARROW, "ARROW"},

  /* loops */

  {WITH, "WITH"},
  {LOOP, "LOOP"},
  {UNTIL, "UNTIL"},
  {UPDATE, "UPDATE"},
  {EXIT, "EXIT"},
  {SKIP, "SKIP"},

  /* conditional */

  {IF, "IF"},
  {ELSE, "ELSE"},

  /* others */

  {RETURN, "RETURN"},
  {ID, "ID"},
  {ALLOC, "ALLOC"},
  {_NULL, "NULL"},

  /* literals */

  {TRUE, "TRUE"},
  {FALSE, "FALSE"},
  {INTLIT, "INTLIT"},
  {FLOATLIT, "FLOATLIT"},
  {CHARLIT, "CHARLIT"},
  {STRINGLIT, "STRINGLIT"},

  /* binary operations */

  {ASSIGN, "ASSIGN"},
  {EQUALS, "EQUALS"},
  {NEQUALS, "NEQUALS"},
  {LT, "LT"},
  {GT, "GT"},
  {LTE, "LTE"},
  {GTE, "GTE"},
  {PLUS, "PLUS"},
  {MINUS, "MINUS"},
  {MUL, "MUL"},
  {DIV, "DIV"},
  {MOD, "MOD"},
  {AND, "AND"},
  {OR, "OR"},
  {PLUSEQUALS, "PLUSEQUALS"},
  {MINUSEQUALS, "MINUSEQUALS"},
  {MULEQUALS, "MULEQUALS"},
  {DIVEQUALS, "DIVEQUALS"},
  {MODEQUALS, "MODEQUALS"},

  /* unary operators */

  {NOT, "NOT"},
  {INCR, "INCR"},
  {DECR, "DECR"}

};


/*extern int yywrap() {
  return 1;
}*/


bool hasError = false;

%}

/* RULES */

%%


[ \t]                   { l.col++; }
[\n]                    { l.line++; l.col = 0; }


"<bool>"                { INC; tokens.push_back(TOK(BOOL)); }
"<int>"                 { INC; tokens.push_back(TOK(INT)); }
"<float>"               { INC; tokens.push_back(TOK(FLOAT)); }
"<char>"                { INC; tokens.push_back(TOK(CHAR)); }
"<string>"              { INC; tokens.push_back(TOK(STRING)); }
"<datatype>"                { INC; tokens.push_back(TOK(DATATYPE)); }
"<void>"                { INC; tokens.push_back(TOK(VOID)); }


"if"                    { INC; tokens.push_back(TOK(IF)); }
"else"                  { INC; tokens.push_back(TOK(ELSE)); }
```

```
"with"                  { INC; tokens.push_back(TOK(WITH)); }
"loop"                  { INC; tokens.push_back(TOK(LOOP)); }
"until"                 { INC; tokens.push_back(TOK(UNTIL)); }
"update"                { INC; tokens.push_back(TOK(UPDATE)); }
"exit"                  { INC; tokens.push_back(TOK(EXIT)); }
"skip"                  { INC; tokens.push_back(TOK(SKIP)); }


"true"                  { INC; tokens.push_back(TOK(TRUE)); }
"false"                 { INC; tokens.push_back(TOK(FALSE)); }
[0-9]+                  { INC; tokens.push_back(TOK(INTLIT)); }
[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)? { INC; tokens.push_back(TOK(FLOATLIT)); }
\"([^\\\"]|\\.)*\"          { INC; tokens.push_back(TOK(STRINGLIT)); }
\'([^\\\"]|\\.)\'           { INC; tokens.push_back(TOK(CHARLIT)); }


"return"                { INC; tokens.push_back(TOK(RETURN)); }
"alloc"                 { INC; tokens.push_back(TOK(ALLOC)); }
"null"                  { INC; tokens.push_back(TOK(_NULL)); }
[_A-Za-z][_A-Za-z0-9]*          { INC; tokens.push_back(TOK(ID)); }


"("                 { INC; tokens.push_back(TOK(LPAREN)); }
")"                 { INC; tokens.push_back(TOK(RPAREN)); }
"{"                 { INC; tokens.push_back(TOK(LBRACE)); }
"}"                 { INC; tokens.push_back(TOK(RBRACE)); }
"["                 { INC; tokens.push_back(TOK(LBOX)); }
"]"                 { INC; tokens.push_back(TOK(RBOX)); }
";"                 { INC; tokens.push_back(TOK(SEMICOLON)); }
":"                 { INC; tokens.push_back(TOK(COLON)); }
"."                 { INC; tokens.push_back(TOK(DOT)); }
","                 { INC; tokens.push_back(TOK(COMMA)); }
"$"                 { INC; tokens.push_back(TOK(DOLLAR)); }
"#"                 { INC; tokens.push_back(TOK(HASH)); }
"~"                 { INC; tokens.push_back(TOK(TILDE)); }
"->"                    { INC; tokens.push_back(TOK(ARROW)); }


"="                 { INC; tokens.push_back(TOK(ASSIGN)); }
"=="                    { INC; tokens.push_back(TOK(EQUALS)); }
"!="                    { INC; tokens.push_back(TOK(NEQUALS)); }
"<"                 { INC; tokens.push_back(TOK(LT)); }
">"                 { INC; tokens.push_back(TOK(GT)); }
"<="                    { INC; tokens.push_back(TOK(LTE)); }
">="                    { INC; tokens.push_back(TOK(GTE)); }
"+"                 { INC; tokens.push_back(TOK(PLUS)); }
"-"                 { INC; tokens.push_back(TOK(MINUS)); }
"*"                 { INC; tokens.push_back(TOK(MUL)); }
"/"                 { INC; tokens.push_back(TOK(DIV)); }
"%"                 { INC; tokens.push_back(TOK(MOD)); }
"+="                    { INC; tokens.push_back(TOK(PLUSEQUALS)); }
```

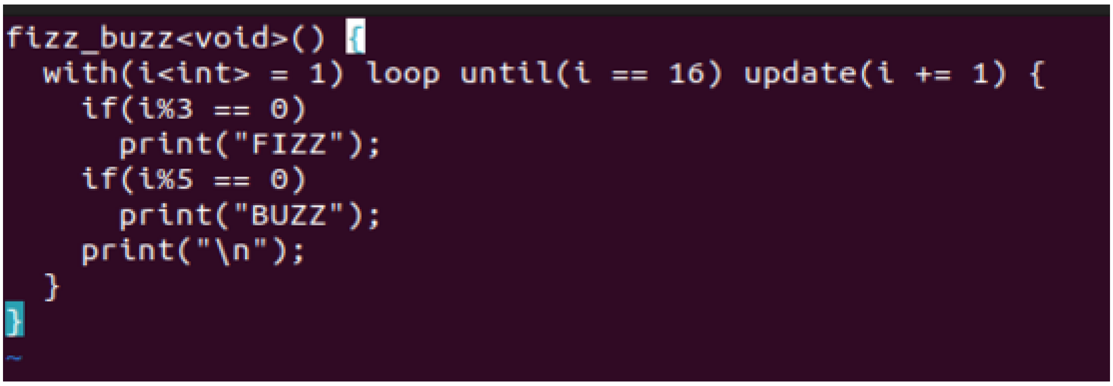
```
"-="                    { INC; tokens.push_back(TOK(MINUSEQUALS)); }
"*="                    { INC; tokens.push_back(TOK(MULEQUALS)); }
"/="                    { INC; tokens.push_back(TOK(DIVEQUALS)); }
"%="                    { INC; tokens.push_back(TOK(MODEQUALS)); }
"&&"                    { INC; tokens.push_back(TOK(AND)); }
"||"                    { INC; tokens.push_back(TOK(OR)); }


"!"                 { INC; tokens.push_back(TOK(NOT)); }
"++"                    { INC; tokens.push_back(TOK(INCR)); }
"--"                    { INC; tokens.push_back(TOK(DECR)); }



.                   {
                      cerr << COLOR_BEGIN << RESET << FG << RED << COLOR_END << "\n\n\nError processing lexeme: ";
                      cout << COLOR_BEGIN << BOLD_ON << FG << MAGENTA << COLOR_END << string(yytext, yyleng);
                      cout << COLOR_BEGIN << RESET << FG << CYAN << COLOR_END << " at line: ";
                      cout << COLOR_BEGIN << BOLD_ON << FG << MAGENTA << COLOR_END << l.line;
                      cout << COLOR_BEGIN << RESET << FG << CYAN << COLOR_END << ", column: ";
                      cout << COLOR_BEGIN << BOLD_ON << FG << MAGENTA << COLOR_END << l.col;
                      hasError = true;
                      yyterminate();
                    }




%%

/* USER SUBROUTINES */

int main() {

  yyin = fopen("input.ergo", "r");
  yylex();
  if(hasError) {
    cerr << COLOR_BEGIN << FG << RED << COLOR_END << "\nParser terminates with error!\n\n\n";
    return 1;
  }

  cout << COLOR_BEGIN << UNDERLINE_ON << BOLD_ON << FG << GREEN << COLOR_END;
  cout << "\n\n\nLEXEME\t\t\tToken\t\t\tLine\t\t\tColumn";
  cout << COLOR_BEGIN << RESET << COLOR_END;
  int tabs;
  for(Token token : tokens) {
    cout << COLOR_BEGIN << ITALICS_ON << FG << WHITE << COLOR_END << endl << token.lexeme;
    tabs = 3 - token.lexeme.size()/8;
    for(int i = 0; i < tabs; ++i)
      cout << "\t";
    cout << COLOR_BEGIN << BOLD_ON << FG << MAGENTA << COLOR_END <<
tokenMap[token.type];
    tabs = 3 - tokenMap[token.type].size()/8;
    for(int i = 0; i < tabs; ++i)
      cout << "\t";
    cout << COLOR_BEGIN << RESET << FG << CYAN << COLOR_END;
    cout << token.location.line << "\t\t\t" << token.location.col;
  }
  cout << "\n\n\n";
  return 0;
}
```

colors.h (for formatting output)

```
#ifndef COLORS
#define COLOR_BEGIN     "\033["
#define COLOR_END       "m"
#define RESET           ";0"
#define BOLD_ON         ";1"
#define UNDERLINE_ON    ";4"
#define ITALICS_ON  ";3"
#define BOLD_OFF        ";21"
#define UNDERLINE_OFF   ";24"
#define FG              ";3"
#define BG              ";4"
#define BLACK           "0"
#define RED             "1"
#define GREEN           "2"
#define YELLOW          "3"
#define BLUE            "4"
#define MAGENTA         "5"
#define CYAN            "6"
#define WHITE           "7"
#endif
```

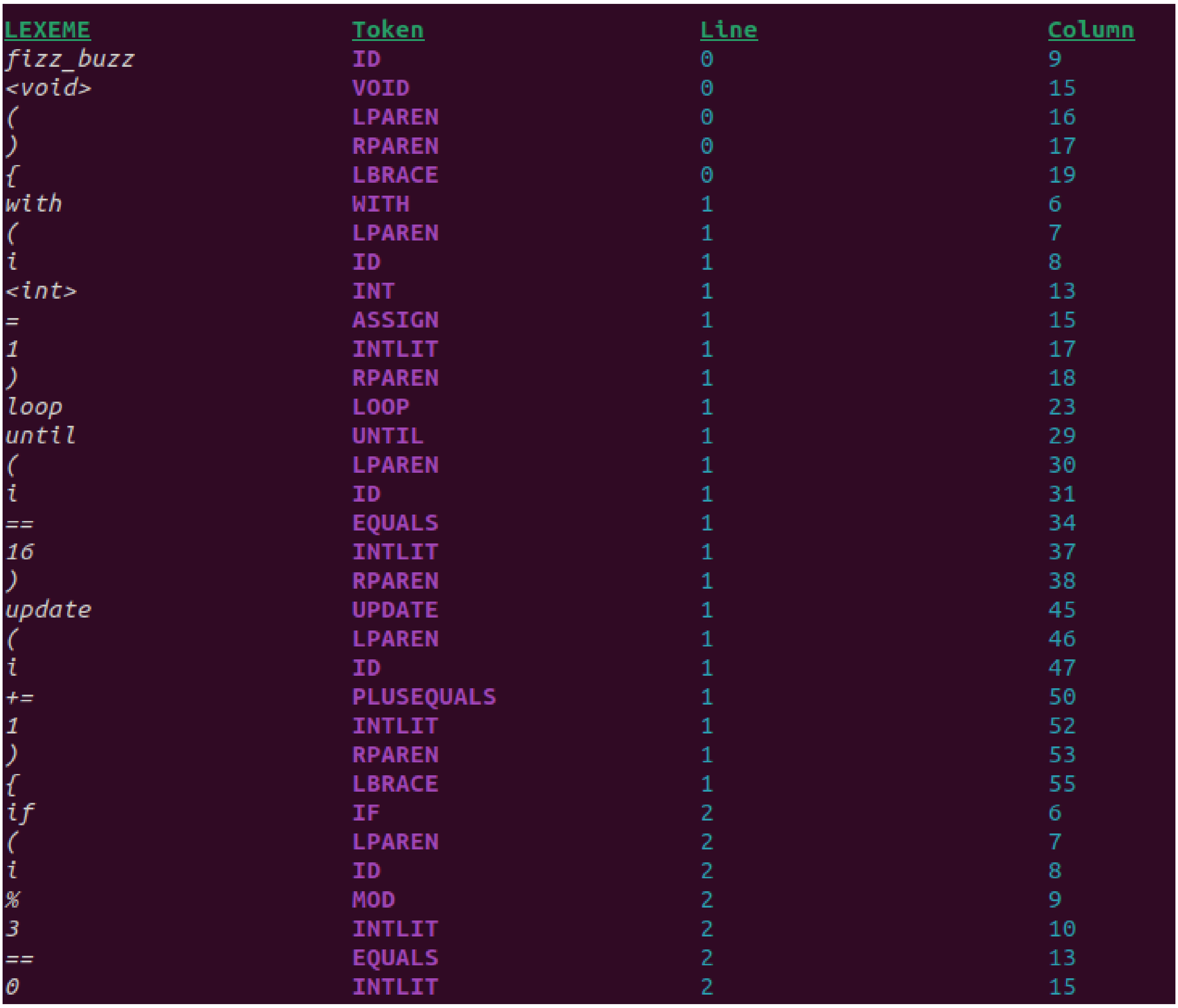
Output

Correct code

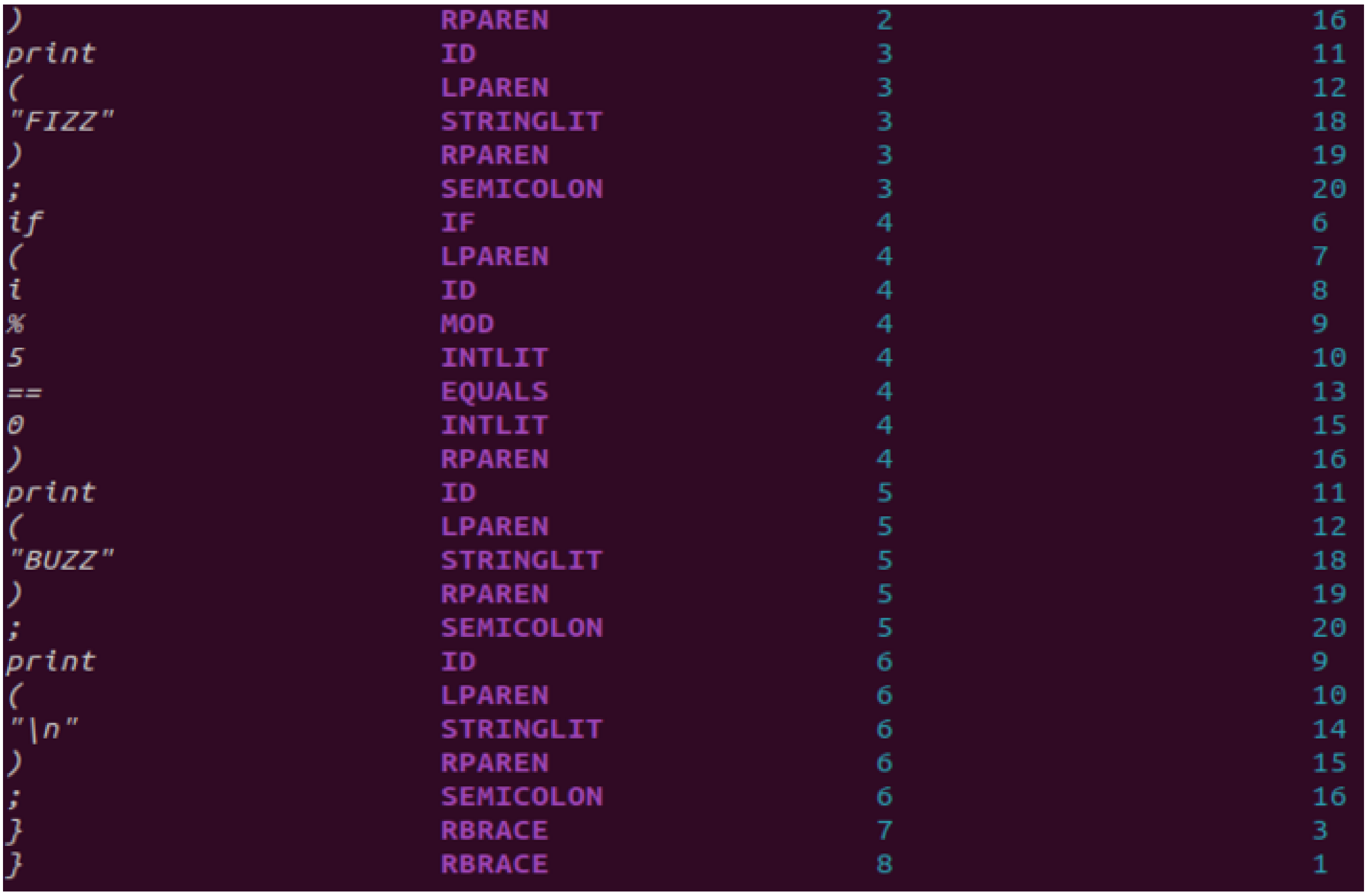


```
fizz_buzz<void>() {
  with(i<int> = 1) loop until(i == 16) update(i += 1) {
    if(i%3 == 0)
      print("FIZZ");
    if(i%5 == 0)
      print("BUZZ");
    print("\n");
  }
}
~
```

Lexer output

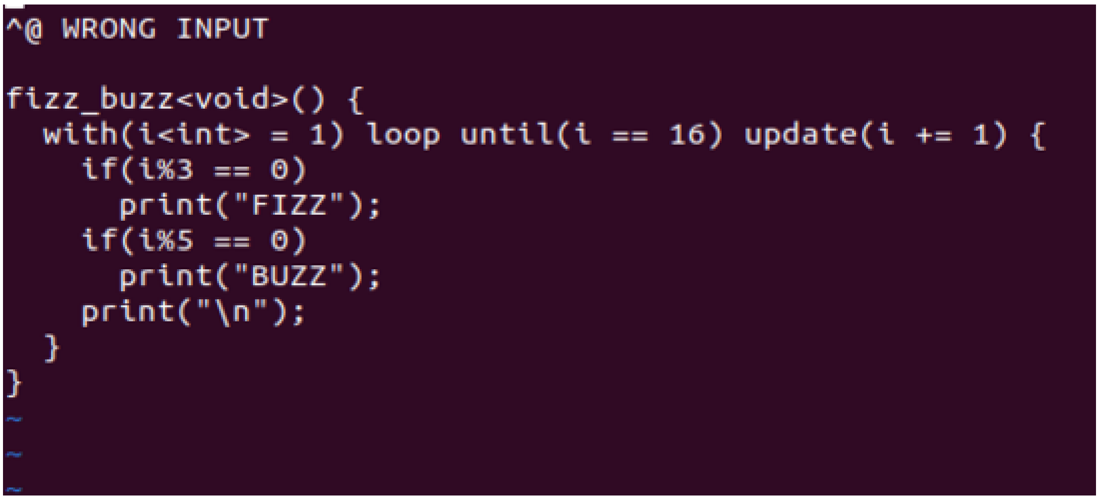


```
LEXEME                  Token                   Line                    Column
fizz_buzz               ID                      0                       9
<void>                  VOID                    0                       15
(                       LPAREN                  0                       16
)                       RPAREN                  0                       17
{                       LBRACE                  0                       19
with                    WITH                    1                       6
(                       LPAREN                  1                       7
i                       ID                      1                       8
<int>                   INT                     1                       13
=                       ASSIGN                  1                       15
1                       INTLIT                  1                       17
)                       RPAREN                  1                       18
loop                    LOOP                    1                       23
until                   UNTIL                   1                       29
(                       LPAREN                  1                       30
i                       ID                      1                       31
==                      EQUALS                  1                       34
16                      INTLIT                  1                       37
)                       RPAREN                  1                       38
update                  UPDATE                  1                       45
(                       LPAREN                  1                       46
i                       ID                      1                       47
+=                      PLUSEQUALS              1                       50
1                       INTLIT                  1                       52
)                       RPAREN                  1                       53
{                       LBRACE                  1                       55
if                      IF                      2                       6
(                       LPAREN                  2                       7
i                       ID                      2                       8
%                       MOD                     2                       9
3                       INTLIT                  2                       10
==                      EQUALS                  2                       13
0                       INTLIT                  2                       15
```



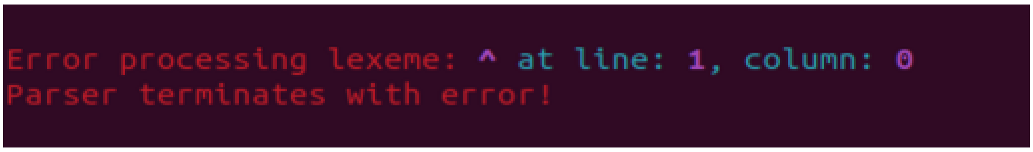
```
)                         RPAREN                    2                         16
print                     ID                        3                         11
(                         LPAREN                    3                         12
"FIZZ"                    STRINGLIT                 3                         18
)                         RPAREN                    3                         19
;                         SEMICOLON                 3                         20
if                        IF                        4                         6
(                         LPAREN                    4                         7
i                         ID                        4                         8
%                         MOD                       4                         9
5                         INTLIT                    4                         10
==                        EQUALS                    4                         13
0                         INTLIT                    4                         15
)                         RPAREN                    4                         16
print                     ID                        5                         11
(                         LPAREN                    5                         12
"BUZZ"                    STRINGLIT                 5                         18
)                         RPAREN                    5                         19
;                         SEMICOLON                 5                         20
print                     ID                        6                         9
(                         LPAREN                    6                         10
"\n"                      STRINGLIT                 6                         14
)                         RPAREN                    6                         15
;                         SEMICOLON                 6                         16
}                         RBRACE                    7                         3
}                         RBRACE                    8                         1
```

Incorrect code



```
^@ WRONG INPUT

fizz_buzz<void>() {
  with(i<int> = 1) loop until(i == 16) update(i += 1) {
    if(i%3 == 0)
      print("FIZZ");
    if(i%5 == 0)
      print("BUZZ");
    print("\n");
  }
}
~
~
~
```

Lexer output



```
Error processing lexeme: ^ at line: 1, column: 0
Parser terminates with error!
```