REPORT CSE439
SPRING 2024

Submitted by:

Mohamed Ahmed Esmat 21P0144

Abdallah Ahmed Hamdy 21P0333

Ahmed Hossam Eldin Ibrahim 21P0271

Youssef Mohammed Alsayyd 21P0094

Saleh Ahmed Saleh Elsayed 21P0324

LIST OF TABLES

List Of Figures

Contents

Language Specifications Document

Keywords:

Keyword	Description
alignas	Alignment specifier (C23)
alignof	Alignment specifier (C23)
auto	Denotes a variable whose type is deduced from its initializer (C23)

bool	Boolean type (C23)
break	Exits a switch or loop statement
case	Keyword used in switch statements
const	Declares a variable that cannot be modified after initialization
constexpr	Constant expression (C23)
continue	Continues to the next iteration of a loop
default	The default keyword is used in switch statements
do	Keyword used in do-while loops
else	Keyword used in if and switch statements
enum	Enumerated type
extern	Declares a variable or function that is defined in another file
false	Boolean constant (C23)
for	Keyword used in for loops
goto	Transfers control to a labeled statement
if	Keyword used in if statements
inline	(C99) Suggests that the function should be

	inserted inline in the call site
register	Suggests that a variable should be stored in a register
restrict	(C99) Restricts the memory that a pointer can access
return	Returns a value from a function
signed	Signed integer type
sizeof	Gives the size of an expression in bytes
static	Declares a variable that has static storage duration
static_assert	Checks a constant expression at compile time (C23)
struct	Keyword used to define a structure
switch	Keyword used in switch statements

thread_local	Thread-local storage (C11)
true	Boolean constant (C23)
typedef	Creates a synonym for a type
typeof	(C23) Returns the type of an expression
typeof_unqual	(C23) Returns the unqualified type of an expression
union	Keyword used to define a union
unsigned	Unsigned integer type

void	Represents the absence of a type
volatile	Declares a variable that may be modified by an external source
while	Keyword used in while loops
_Alignas	Alignment specifier (C11)
_Alignof	Alignment specifier (C11)
_Atomic	Atomic operations (C11)
_Bool	Boolean type (C99)
_Complex	Complex number type (C99)
_Decimal128	Decimal floating-point type (C23)
_Decimal32	Decimal floating-point type (C23)
_Decimal64	Decimal floating-point type (C23)
_Generic	(C11) Used with typeof to specify a generic type
_Imaginary	Imaginary number type (C99)
_Noreturn	Indicates that a function does not return (C11)
_Static_assert	Checks a constant expression at compile time (C11)
_Thread_local	Thread-local storage (C11)

Table 1

Variable & Function Identifiers:

-Valid Characters: Identifiers can contain letters (both uppercase and lowercase), digits, and underscores (_), and must begin with a letter or underscore.

-Case Sensitivity: C++ is case-sensitive, so uppercase and lowercase letters are considered distinct.

-Reserved Keywords: Identifiers cannot be the same as reserved keywords in C++.

-No Special Characters: Identifiers cannot contain special characters or spaces

Type	Example
Variable Identifier	‘int age’ (age)
Function Identifier	‘void displayMsg()’ (displayMsg)

Table 2

Functions:

Type	Example
Function Declaration	int sum(int x, int y);
Function Definition	int sum(int x, int y) { return x + y; }
Function Call	int = sum(1, 2)
Function Body	{ return x + y; }
Function Parameters	(int x, int y)
Return Type	int
Return Statement	return x + y;
Local Variable (inside main)	int x = 1;
Function Prototype (Declaration at any scope)	int sum(int, int);
File Scope	Functions must be defined at file scope.
No Nested Functions	Nested functions are not allowed in standard C.

Function Access to Variables	Functions cannot directly access local variables from the caller (except through parameters).

Table 3

Data Types:

Type	Definition
char	Character type
double	Double-precision floating-point type
float	Single-precision floating-point type
int	Integer type
long	Long integer type
short	Short integer type
signed	Signed integer type
unsigned	Unsigned integer type
void	Represents the absence of a type

Table 4

Statements:

Assignment Statement	Basic operation	A=b
	addition assignment	a += b
	subtraction assignment	a -= b
	multiplication assignment	a *= b
	division assignment	a /= b
	modulo assignment	a %= b
	bitwise AND assignment	a &= b
	bitwise OR assignment	a |= b
	bitwise XOR assignment	a ^= b

	bitwise left shift assignment/	a <<= b
	bitwise right shift assignment	a >>= b

Table 5

Return Statement	return expression; return ; 1) Returns the result of the expression to the caller and terminates the current function. This is only valid if the function return type is not void. 2) Terminates the current function. This is only valid if the function return type is void.

Iterative Statement	for (initialization; condition; increment/decrement) {
	
	// Code block
	
	}
	
	while (condition) {
	
	// Code block
	}
	
	do
	
	{
	// Code block
	} while (condition
	);

Conditional Statements	if (condition) {
	// Code block
	} else if (condition) {
	// Code block
	} else {
	// Code block
	}
	switch (*expression*) *statement*
	case *constant-expression* : *statement*
	default : *statement*
	continue ;
	
	The continue statement causes a jump, as if by goto, to the end of the loop body (it may only appear within the loop body of for, while, and do- while loops).
	break ;

Table 6

	After this statement the control is transferred to the statement or declaration immediately following the enclosing loop or switch, as if by goto.
Function Call Statement	function_name(arguments);

Table 7

Expressions:

Arithmetic:

<table>
<tr><th>Operator</th><th>Expression</th><th>Description</th></tr>
<tr><td>+</td><td>X + Y</td><td>Addition of X and Y</td></tr>
<tr><td>-</td><td>X - Y</td><td>Subtraction of Y from X</td></tr>
<tr><td>*</td><td>X * Y</td><td>Multiplication of X and Y</td></tr>
<tr><td>/</td><td>X / Y</td><td>Division of X by Y</td></tr>
<tr><td>%</td><td>X % Y</td><td>Remainder of X divided by Y</td></tr>
<tr><td>-</td><td>- X</td><td>Negation of X</td></tr>
<tr><td>++</td><td>++X</td><td>Pre-increment of X by 1</td></tr>
<tr><td>++</td><td>X++</td><td>Post-increment of X by 1</td></tr>
<tr><td>--</td><td>--X</td><td>Pre-decrement of X by 1</td></tr>
<tr><td>--</td><td>X--</td><td>Post-decrement of X by 1</td></tr>
<tr><td>+</td><td>+X</td><td>Value of X after promotions</td></tr>
<tr><td>-</td><td>-X</td><td>Negative of X</td></tr>
<tr><td>~</td><td>~X</td><td>Bitwise NOT of X</td></tr>
<tr><td>&</td><td>X & Y</td><td>Bitwise AND of X and Y</td></tr>
<tr><td>|</td><td>X | Y</td><td>Bitwise OR of X and Y</td></tr>
<tr><td>^</td><td>X ^ Y</td><td>Bitwise XOR of X and Y</td></tr>
<tr><td><<</td><td>X << Y</td><td>X left shifted by Y</td></tr>
<tr><td>>></td><td>X >> Y</td><td>X right shifted by Y</td></tr>
</table>

Table 8

Boolean:

Operator	Expression	Description
==	X == Y	Equality of x and y
!=	X != Y	Inequality of x and y
>	X > Y	X is greater than Y
<	X < Y	X is less than Y
>=	X >= Y	X is greater than or equal to Y
<=	X <= Y	X is less than or equal to Y
!	!X	Logical negation of x
&&	X && Y	Logical conjunction of x and y
||	X || Y	Logical disjunction of x and y

Table 9

Token Specifications and Patterns

Keyword Pattern

```
 Regex
keywordPattern("\\b(alignas|alignof|auto|bool|break|case|const|constexpr|continue|defaul
t|do|else|enum|extern|false|for|goto|if|inline|register|restrict|return|signed|sizeof|static|
static_assert|struct|switch|thread_local|true|typedef|typeof|typeof_unqual|union|unsigned
|void|volatile|while|_Alignas|_Alignof|_Atomic|_Bool|_Complex|_Decimal128|_Decimal32|
_Decimal64|_Generic|_Imaginary|_Noreturn|_Static_assert|_Thread_local)\\b");
```

Data type Pattern

```
_dataTypePattern("\\b(char|double|float|int|long|short|signed|unsigned|void)\\b");
```

```
regex arithPattern("(\\+\\+|\\-\\-|\\+|\\-
|\\*|\\/|\\%|\\~|\\<\\<|\\>\\>|\\^|([^\\&]|^)\\&([^\\&]|$)|([^\\|]|^)\\|([^\\|]|$))");
```

Bool Pattern

```
regex
boolPattern("(\\=\\=|\\!\\=|\\<\\=|\\>\\=|([^\\>]|^)\\>([^\\>]|$)|([^\\<]|^)\\<([^\\<]|$)|\\!|\
\&\\&|\\|\\|)");
```

Assignment Pattern

```
_regex assignmentPattern("((\\+=)|(-=)|(\\*=)|(\\/=)|(\\%=)|(&=)|(\\|=)|(\\^=)|(<<=)|(>>=)|(=))");
```

Identifier Pattern

```
_regex identifierPattern("^[_a-zA-Z][_a-zA-Z0-9]*$");
```

Punctuation Pattern

```
_regex punctuationPattern("(\\?|\\-\\>|\\:\\:|\\{|\\}|\\(|\\)|\\[|\\]|\\;|\\,|\\.|\\:)");
```

Decimal regex

```
_regex decimal_regex("^[-+]?[1-9][0-9]*\\.?[0-9]*$");
```

Binary regex

```
_regex binary_regex("^0b[01]+$");
```

Octal regex

```
_regex octal_regex("^0[0-7]*$");
```

Hex regex

```
_regex hex_regex("^0x[a-fA-F0-9]+$");
```

```
}
```

String regex

```
_regex string_regex("\"(\\\\.|[^\"])*\"");
```

Char regex

```
 regex char_regex("'(\\\\.|[^'])*'");
```

Lexical Analyzer

Print Tokens

```
void printTokens(const vector<pair<string, string>>& tokens) {
    cout << "Tokens\n";
    for (const auto& token : tokens) {
    if (token.second != "") {
    cout << "<" << token.first << ", " << token.second << ">" << "\n";
    }
    else {
    cout << "<" << token.first << ">" << "\n";
    }
    }
}
```

This function prints tokens, where each token is represented as a pair of strings (token type and token value). It iterates through the vector of token pairs and prints them in the format "<token_type, token_value>", omitting the value if it's empty.

Print Errors

```
void printErrors() {

   cout << "Errors\n";
```

This function prints errors stored in a container named `errors`. It iterates through the container and prints each error message followed by a newline.

Print Lexemes

```
void printLexemes() {
   cout << "Lexemes\n";
   for (const string& lex : lexemes) {
          cout << lex << endl;
   }
}
```

This function prints lexemes stored in a container named lexemes. It iterates through the container and prints each lexeme followed by a newline.

Remove Comments

```
string removeComments(string code) {
          regex
commentPattern("(\\/\\*([^*]|[\r\n]|(\\*+([^*/]|[\r\n])))*\\*\\/)|(\\/\\/.*)|#[^\\n]*");
          return regex_replace(code, commentPattern, "");
}
```

This function removes comments from the input code string. It uses regular expressions to match and replace different types of comments, including multiline (/* */), single line (//), and preprocessor directives (#).

Remove Extra Spaces

```
string removeExtraSpaces(string code) {
          regex spacePattern("\\s+");
          return regex_replace(code, spacePattern, " ");
```

}

This function removes extra spaces from the input code string. It uses a regular expression to match sequences of whitespace characters and replaces them with a single space.

Extract Preprocessors

```
string extractPreprocessors(string code) {
    regex preprocessorPattern("#[^\\n]*");
    string result;
    smatch match;
    while (regex_search(code, match, preprocessorPattern)) {
    result += match.str() + "\n";
    code = match.suffix().str();
    }
    return result;
}
```

This function extracts preprocessor directives from the input code string. It uses a regular expression to match preprocessor directives starting with # and accumulates them into a result string, each followed by a newline.

Is Valid Identifier

```
bool isValidIdentifier(const string& str) {
        return regex_match(str, identifierPattern);
}
```

This function checks whether a given string is a valid identifier according to some pattern defined elsewhere in the code. It uses a regular expression match to determine whether the string conforms to the identifier pattern.

Process Token

```
void processToken(const string& temp, vector<pair<string, string>>& tokens) {
        if (regex_match(temp, keywordPattern)) {
    tokens.push_back(make_pair(temp, ""));
    lexemes.push_back(temp);
        }
  else if (regex_match(temp, dataTypePattern)) {
    tokens.push_back(make_pair(temp, ""));
    lexemes.push_back(temp);
        }
  else if (regex_match(temp, arithPattern)) {
    tokens.push_back(make_pair(temp, ""));
    lexemes.push_back(temp);
        }
  else if (regex_match(temp, boolPattern)) {
```

```
  tokens.push_back(make_pair(temp, ""));
  lexemes.push_back(temp);
     }
else if (regex_match(temp, assignmentPattern)) {
  tokens.push_back(make_pair(temp, ""));
  lexemes.push_back(temp);
     }
else if (regex_match(temp, punctuationPattern)) {
  tokens.push_back(make_pair(temp, ""));
  lexemes.push_back(temp);
     }
else {
  if (isValidIdentifier(temp)) {
    auto it = find_if(symbolTableVector.begin(), symbolTableVector.end(),
     [&](const pair<string, string>& entry) { return entry.first == temp; });
    if (it == symbolTableVector.end()) {
      symbolTableVector.push_back(make_pair(temp, to_string(counter++)));
      tokens.push_back(make_pair("id", symbolTableVector.back().second));
     lexemes.push_back(temp);
    } else {
      tokens.push_back(make_pair("id", it->second));
     lexemes.push_back(temp);
```

```
            }
        }
        else {
            errors.push_back(temp);
            lexemes.push_back(temp);
        }
    }
}
```

This function processes a token represented by the string `temp`. It matches the token against various patterns (such as keywords, data types, arithmetic operators, etc.) using regular expressions. Depending on the match, it adds the token to the `tokens` vector along with its type (or an empty string if the token has no associated value), and also adds the token to the `lexemes` vector.

Two Char Ops

```
void twoCharOps(string& temp, const string& code, int& i) {
    string multiCharOp;
    string twoCharOps[] = { "||", "&&", "<=", ">=", "==", "!=", "<<", ">>", "++", "--", "-=", "+=",
"*=", "/=", "%=", "&=", "|=", "^=", "->", "::" };
    string threeCharOps[] = { "<<=", ">>=" };

        if (i + 1 < code.size()) {
```

```
        multiCharOp = temp + code[i + 1];
        if (find(begin(twoCharOps), end(twoCharOps), multiCharOp) != end(twoCharOps)) {
          temp = multiCharOp;
          ++i;
          if (i + 1 < code.size()) {
           multiCharOp = temp + code[i + 1];
           if (find(begin(threeCharOps), end(threeCharOps), multiCharOp) != end(threeCharOps)) {
                temp = multiCharOp;
                ++i;
           }
          }
        }
           }
}
```

This function checks for two-character operators in the code starting at index `i`. If it finds a valid two-character operator, it updates the `temp` string to contain the combined operator, and increments the index i accordingly.

Numbers Detector

```
void numbersDetector(string& temp, const string& code, int& i, vector<pair<string, string>>& tokens) {
    string number = temp;
```

```
bool isInvalid = false;

while (i + 1 < code.size() && (isdigit(code[i + 1]) || code[i + 1] == '.' || isalpha(code[i + 1]))) {

  number += code[++i];

      }

      if (regex_match(number, decimal_regex)) {

  tokens.push_back(make_pair(number, "decimal number"));

  lexemes.push_back(number);

      }

else if (regex_match(number, binary_regex)) {

  tokens.push_back(make_pair(number, "binary number"));

  lexemes.push_back(number);

      }

else if (regex_match(number, octal_regex)) {

  tokens.push_back(make_pair(number, "octal number"));

  lexemes.push_back(number);

      }

else if (regex_match(number, hex_regex)) {

  tokens.push_back(make_pair(number, "hexadecimal number"));

  lexemes.push_back(number);

      }

else {

  errors.push_back(number);
```

```
        lexemes.push_back(number);
            }
}
```

This function detects and processes numbers in the code starting from the string `temp`. It parses the number until it reaches a character that is not part of the number. Then, it matches the number against different number patterns (decimal, binary, octal, hexadecimal) using regular expressions. Depending on the match, it adds the number to the `tokens` vector along with its type, or adds it to the `errors` vector if it doesn't match any valid number pattern.

Analyze Code

```
vector<pair<string, string>> analyzeCode(const string& code) {
    vector<pair<string, string>> tokens;
    string separators = "(){}[]?.,;+-*/%~<>^&|!=:\"\'";
    string temp;
            for (int i = 0; i < code.size(); ++i) {
        char c = code[i];
        if (separators.find(c) != string::npos) {
            if (!temp.empty()) {
             processToken(temp, tokens);
             temp.clear();
            }
            temp += c;
            if (c == '\"') {
             int j = i + 1;
             while (j < code.size()) {
                        if (code[j] == '\"' && code[j - 1] != '\\') {
```

```
                break;
                }
                temp += code[j++];
    }
    if (j < code.size()) {
                temp += code[j++];
    }
    i = j - 1;
      tokens.push_back(make_pair(temp, "string"));
    lexemes.push_back(temp);
   }
   else if (c == '\'') {
    int j = i + 1;
    while (j < code.size() && code[j] != '\'') {
                temp += code[j++];
    }
    if (j < code.size()) {
                temp += code[j++];
}
    i = j - 1;
      tokens.push_back(make_pair(temp, "char"));
    lexemes.push_back(temp);
   }
   else if ((c == '-' || c == '+') && i + 1 < code.size() && isdigit(code[i + 1])) {
```

```
     numbersDetector(temp, code, i, tokens);
    }
    else {
     twoCharOps(temp, code, i);
     processToken(temp, tokens);
    }
    temp.clear();
}
else if (c == ' ') {
    if (!temp.empty()) {
     processToken(temp, tokens);
     temp.clear();
    }
}
else if (isdigit(c) && (temp.empty() || isdigit(temp[0]))) {
    temp += c;
    numbersDetector(temp, code, i, tokens);
    temp.clear();
}
else {
    temp += c;
}
     }
```

```
            if (!temp.empty()) {
        processToken(temp, tokens);
            }


    return tokens;
}
```

This function analyzes the input code string and generates tokens along with their types. It iterates through the characters of the code string and processes them according to various rules, including separators, strings, characters, operators, and numbers. It returns a vector of token pairs representing the analyzed tokens.

Print Symbol Table

```
void printSymbolTable(const vector<pair<string, string>>& symbolTable) {
    cout << endl << setw(15) << left << "Identifier" << setw(25) << "Index" << endl;
    cout << "------------------------" << endl;


            for (const auto& entry : symbolTable) {
        cout << setw(15) << left << entry.first << setw(25) << entry.second << endl;
            }
    cout << endl;
}
```

This function prints the symbol table, which is a vector of pairs containing identifiers and their corresponding indices. It prints the identifier and index pairs in a formatted manner, with columns for identifiers and indices.

Test Cases and Output

Keyword Test

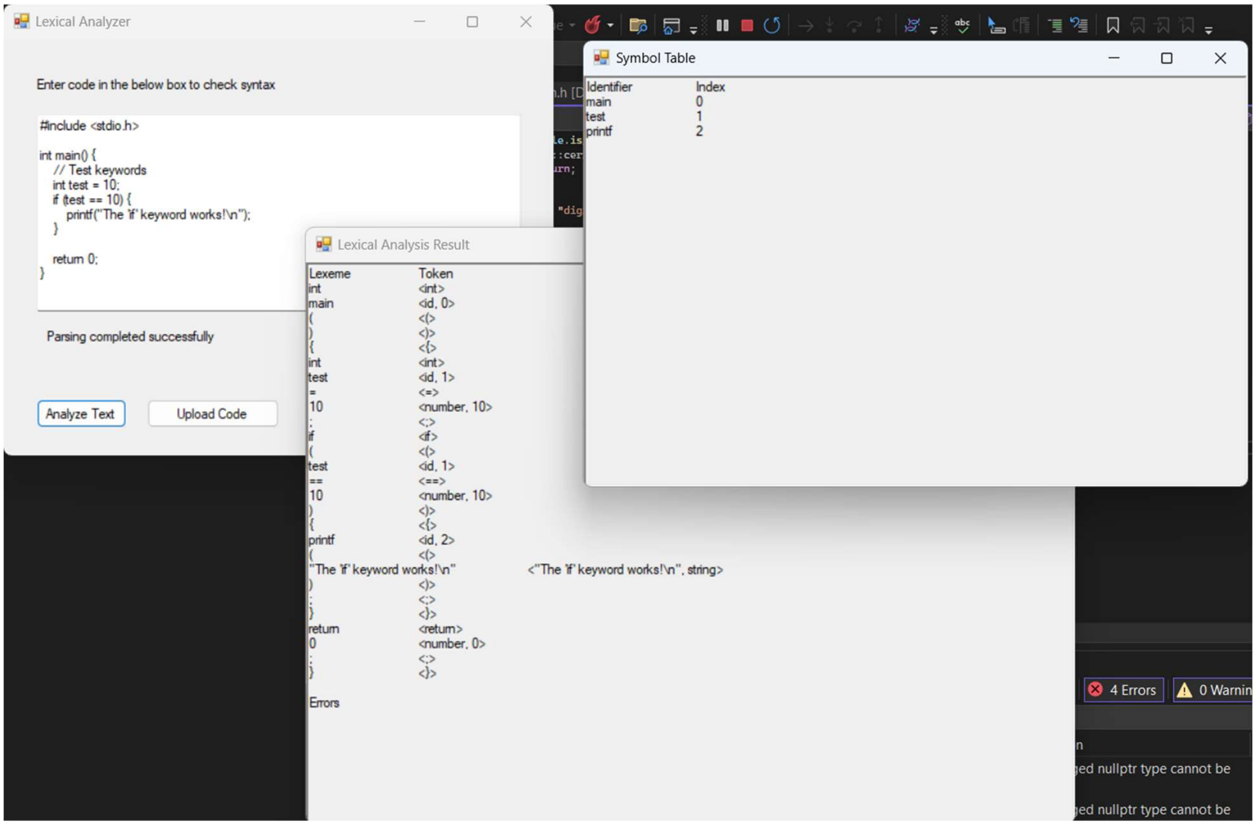

Figure 1

Arithmetic and Digits Test

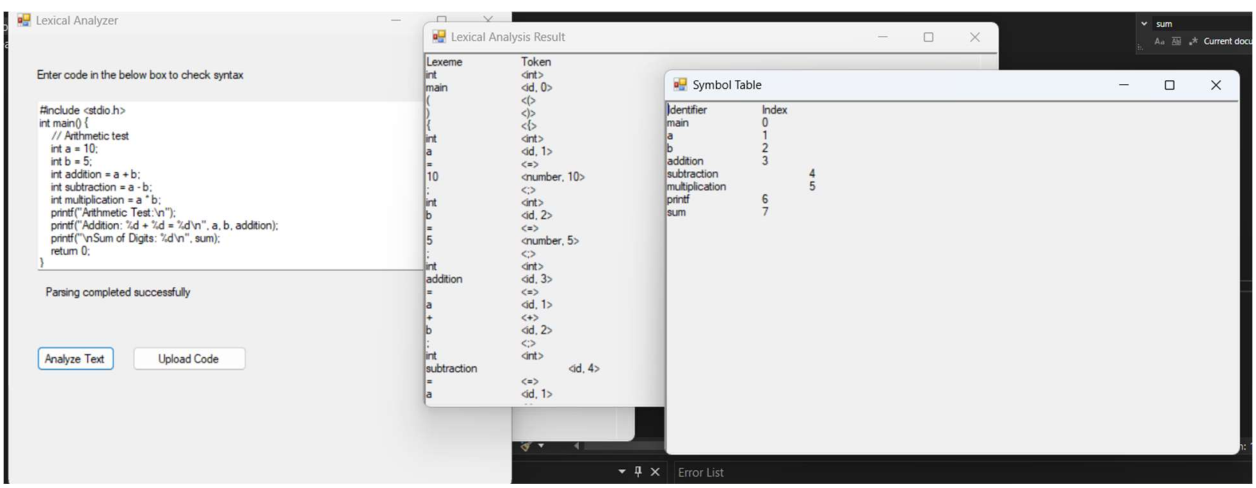

Figure 2

Detect Two Char Ops

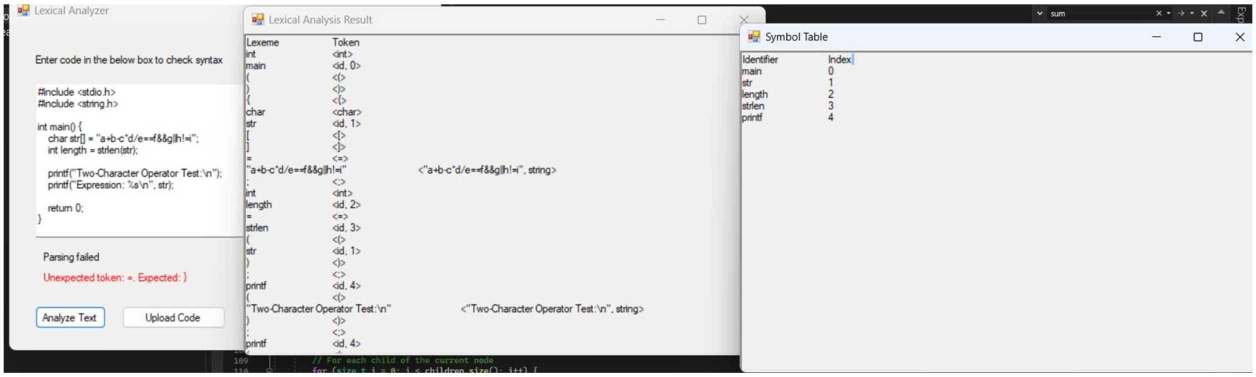

Figure 3

Assignment Test

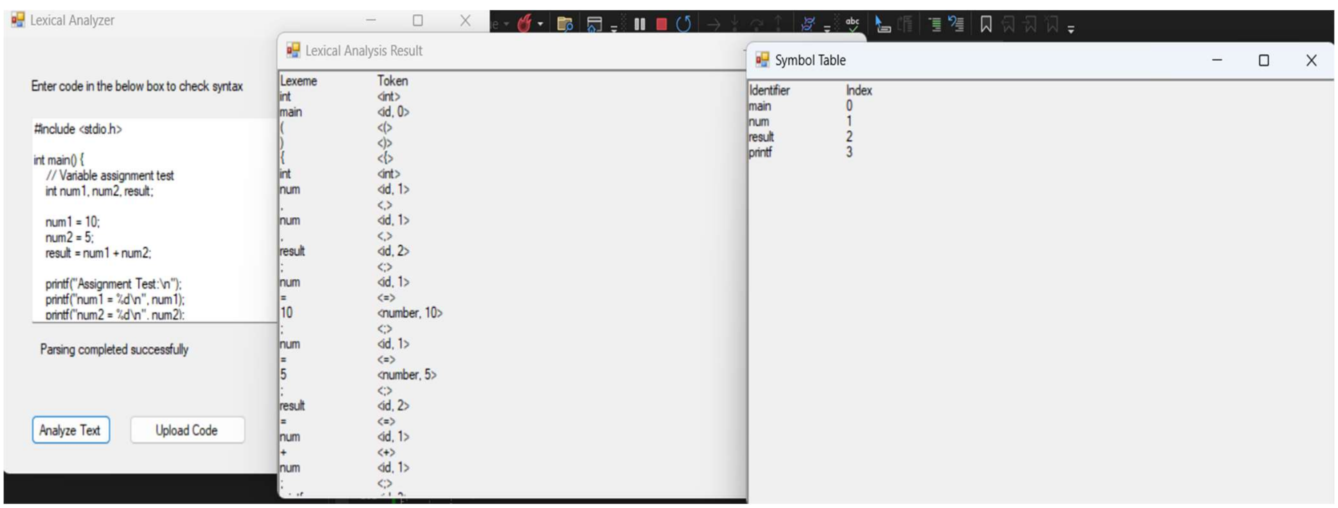

Figure 4

Syntax Analyzer

Grammar Rules:

program -> declarations function_definitions

declarations -> enum_declaration | structure_declaration | array_declaration | pointer_declaration | variable_declaration | ε

function_definitions -> function_definition function_definitions | ε

function_definition -> function_header { body }

function_header -> function_datatype id (parameter_list)

function_datatype -> data_type | void

parameter_list -> parameter sub_parameter_list

sub_parameter_list -> , parameter sub_parameter_list | ε

parameter -> data_type id | ε

body -> single_statement sub_body

sub_body -> body | ε

single_statement -> expression | conditional_statements | iterative_statements | function_calls | return_statement | variable_declaration | array_declaration | pointer_declaration

expression -> boolean_expr | arithmetic_expr | assignment_expr

assignment_expr -> variable assignment_op variable; | variable assignment_op arithmetic_expr;

assignment_op -> '='| '+=' | '-=' | '*=' | '/=' | '%='| '&='| '|='| '^='| '<<=' | '>>='

variable -> number | id

arithmetic_expr -> variable sub_arithmetic_expr

sub_arithmetic_expr -> arithmetic_op arithmetic_expr | ε

arithmetic_op -> '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<' | '>>'

boolean_expr -> variable boolean_op variable; | boolean_op variable;

boolean_op -> '==' | '!=' | '>' | '<' | '>=' | '<=' | '!' | '&&' | '||'

function_call -> id (arguments) ;

arguments -> arg_expression | arg_expression , arguments | ε

arg_expression -> arithmetic_expr | boolean_expr | variable | string

conditional_statements -> if_expr | switch_expr

if_expr -> if (boolean_expr) {body} else_expr | if (boolean_expr) single_statement else_expr

else_expr -> else {body} | else single_statement | ε

switch_expr -> switch(id) { case_expr default_expr }

case_expr -> case const : body break; | case const : body | case_expr | ε

default_expr -> default: body | default: body break;

const -> number | string | char

Iterative_Statements-> for_loop | while_loop | do_while_loop

for_loop -> for (init_expr condition_expr; update_expr) { body}

init_expr -> assignment_expr | variable_declaration | ε

condition_expr -> boolean_expr | ε

update_expr -> variable assignment_op variable | variable assignment_op arithmetic_expr | ε

while_loop -> while (Condition_expr_while) { body }

Condition_expr_while -> boolean_expr | variable

do_while_loop -> do { body } while (Condition_expr_while);

return_statement -> return return_expr ;

return_expr -> arithmetic_expr | boolean_expr | variable | 1 | 0 | ε

variable_declaration -> data_type variable_list ;

data_type -> type_modifier type

type -> int | float | double | char | string

type_modifier -> const | volatile | restrict | long | short | signed | unsigned | ε

variable_list -> id equal_assign | id equal_assign, variable_list

equal_assign -> = const | = id | = arithmetic_expr |ε

enum_declaration -> enum id { enum_constants } ;

enum_constants -> enum_constant | enum_constant , enum_constants enum_constant -> id

structure_declaration -> struct id { member_list } ;

member_list -> member | member ; member_list | ε

member -> data_type id equal_assign

array_declaration -> data_type id array_dimensions;

array_dimensions -> [variable] | [variable] array_dimensions

pointer_declaration -> data_type variable_list_point ;

variable_list_point -> *variable_point | *variable_point, variable_list_point

variable_point -> id | id = &id

Grammar Rules Description:

1. **Program Structure:** A program consists of declarations and function definitions.
2. **Declarations:**Declarations encompass various constructs such as enums, structures, arrays, pointers, and simple variable declarations.
3. **Function Definitions:**Functions are defined with a header specifying return type, function name, and parameter list, followed by a body enclosed in curly braces.
4. **Body:**The body of a function comprises a sequence of statements, including expressions, conditional statements, loops, function calls, and return statements.
5. **Expressions:**Expressions can be arithmetic or boolean, involving variables, constants, and operators.
6. **Arithmetic Expressions:**Arithmetic expressions involve mathematical operations like addition, subtraction, multiplication, and division, possibly with assignment operations.
7. **Boolean Expressions:**Boolean expressions consist of comparisons and logical operations like equality, inequality, greater than, less than, and logical AND/OR.
8. **Function Calls:**Functions are called by their name followed by arguments enclosed in parentheses.
9. **Conditional Statements:**Conditional statements include if-else constructs and switch-case constructs, allowing for branching based on boolean conditions or specific values.
10. **Iterative Statements:**Iterative statements provide looping constructs like for, while, and do-while, allowing repeated execution of a block of code.
11. **Return Statement:**The return statement is used to exit a function and optionally return a value.
12. **Variable Declaration:**Variables are declared with a data type, optional modifiers, and an optional initialization value.
13. **Enum Declaration:**Enumerations are defined with a list of constant values.
14. **Structure Declaration:**Structures are defined with a list of members, each having a data type and an optional initialization value.
15. **Array Declaration:**Arrays are declared with a data type and optional dimensions.
16. **Pointer Declaration**:Pointers are declared with a data type, followed by a list of variables or expressions denoting memory addresses.

Test Cases and Output

Function definition and function call example:

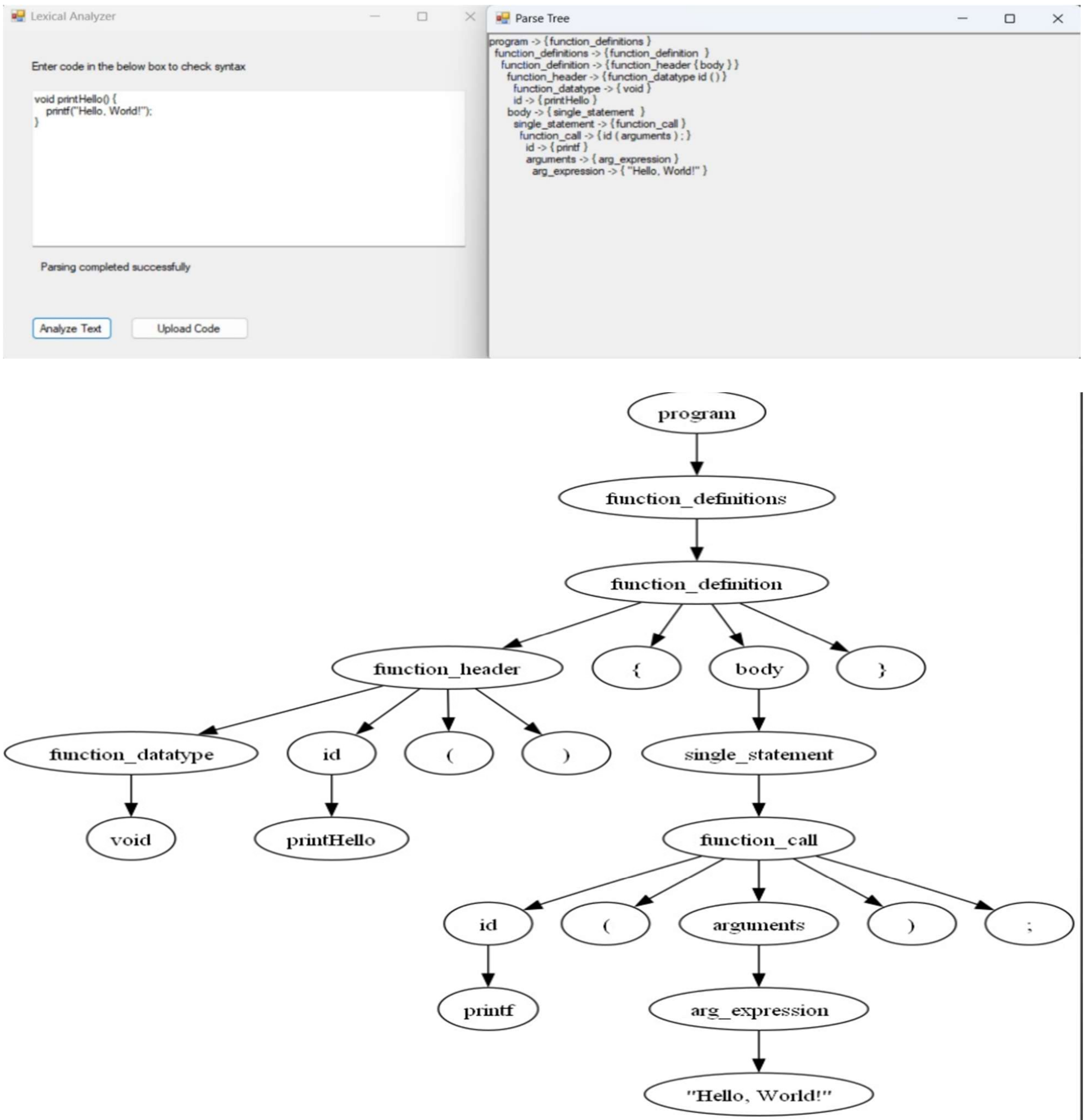

Figure 5

Conditional Statement Example (If Else):

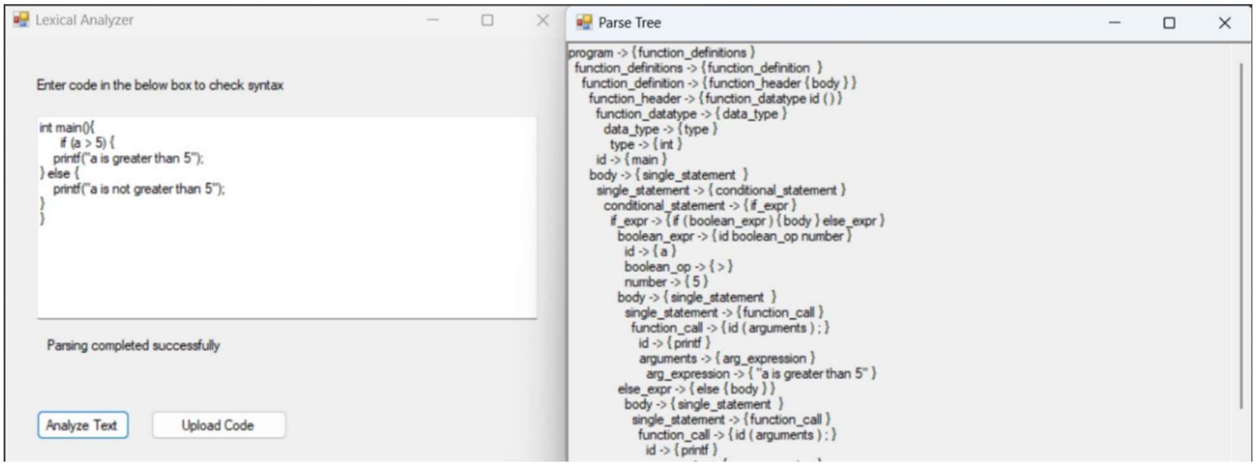

Figure 6

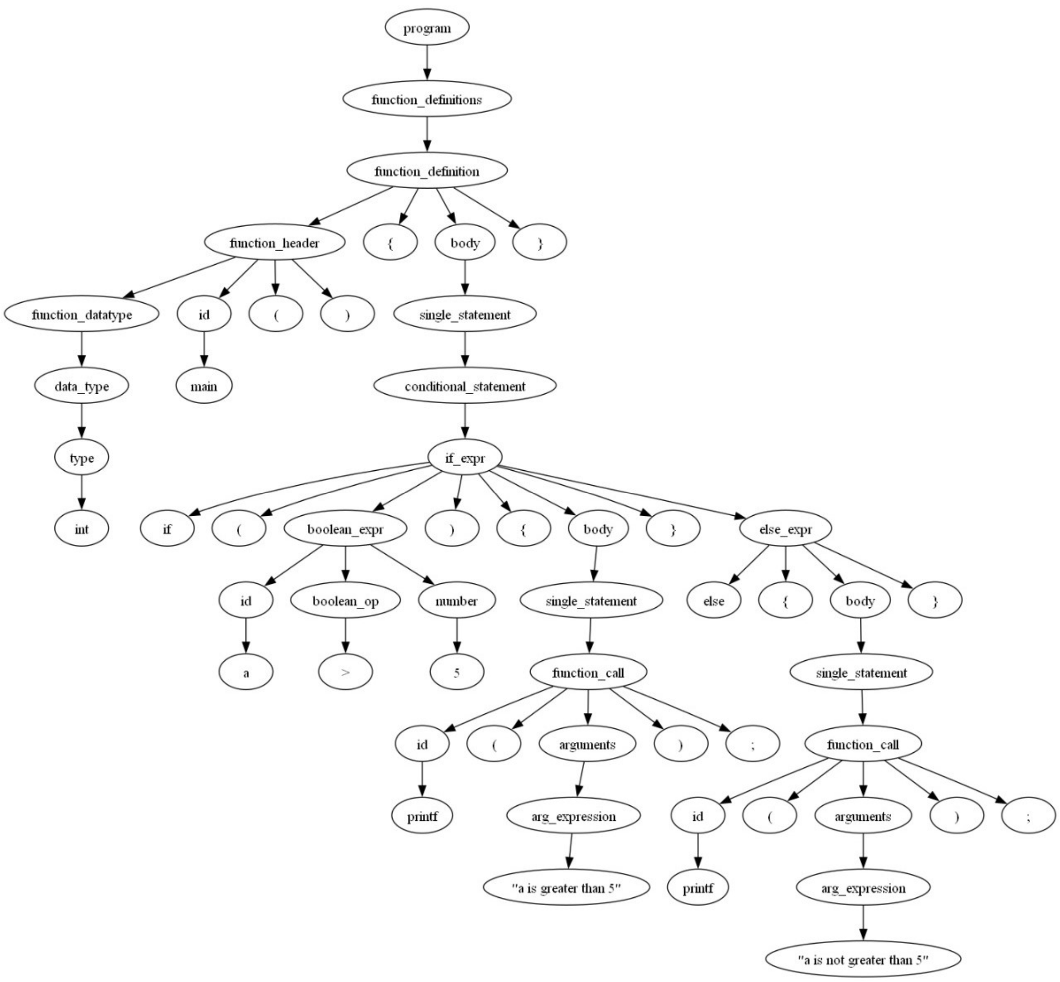

Figure 7

Iterative Statement Example (While):

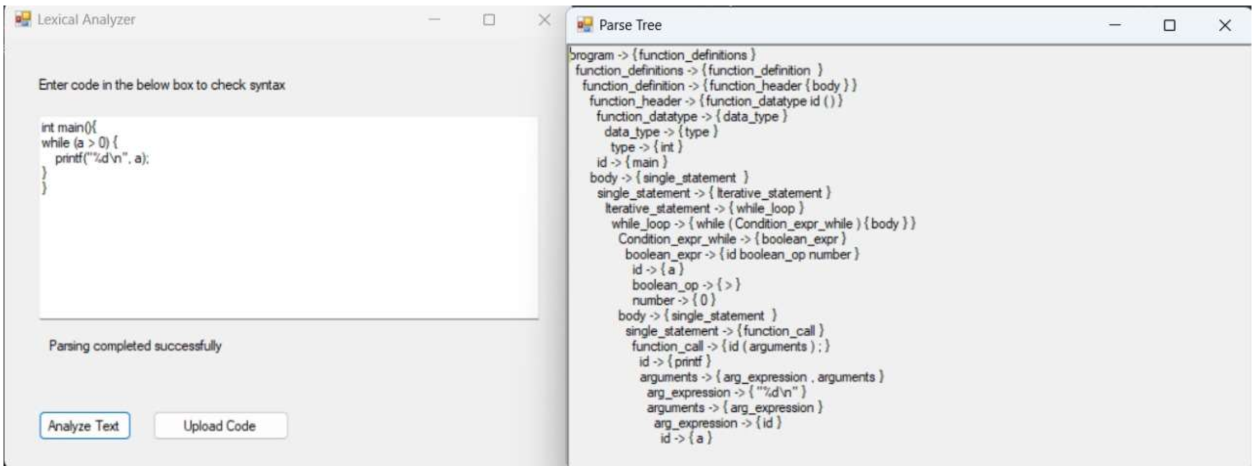

Figure 8

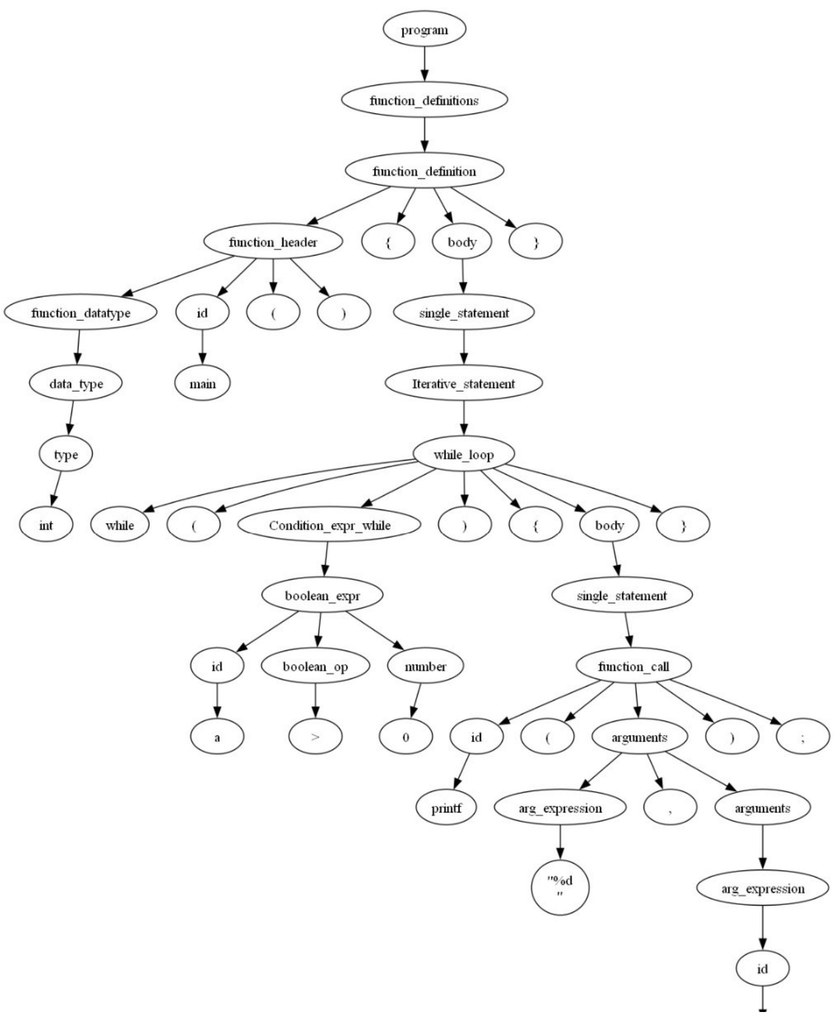

Figure 9

Declaration Statement Example:

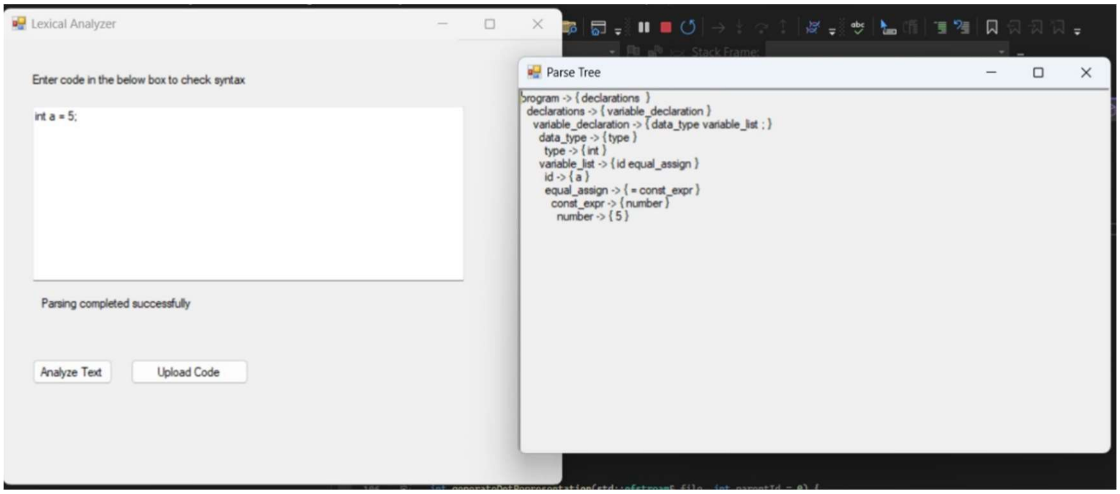

Figure 10

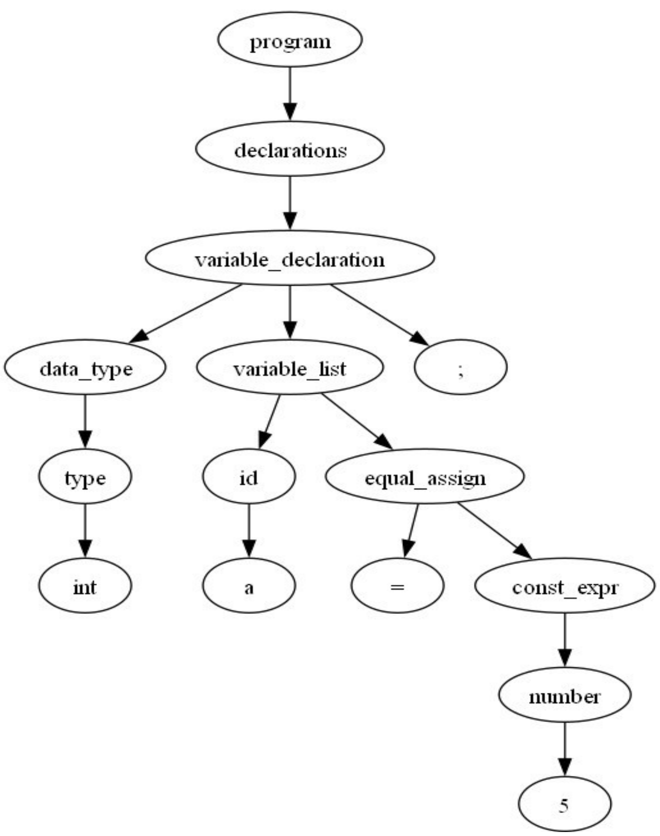

Figure 11

Full Program Example:

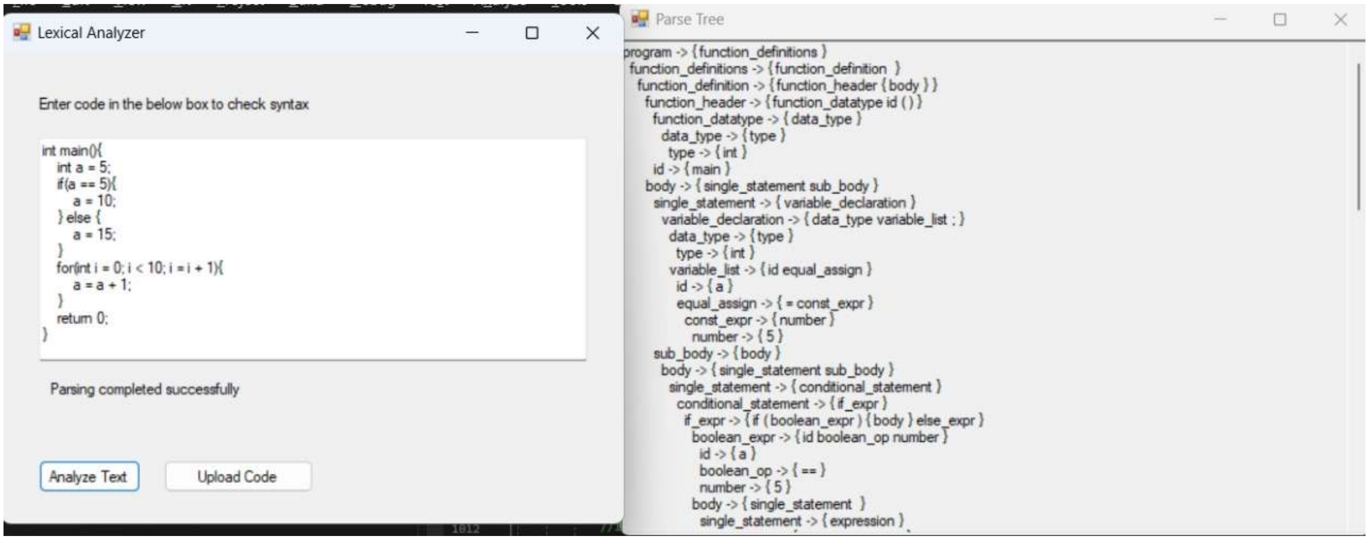

Figure 12

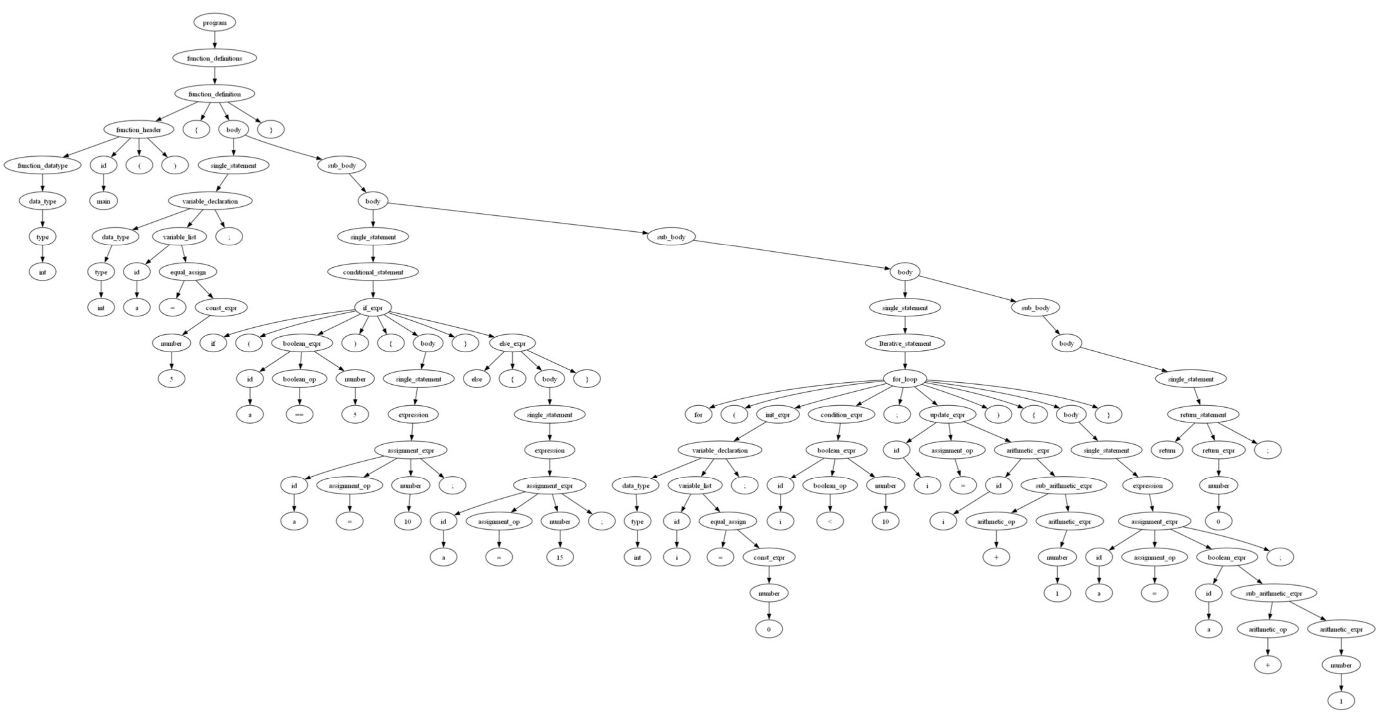

Figure 13

Video and Demo:

https://drive.google.com/file/d/1xFujAAYnMi6d6Gkauaf_Gg0Hp786jj3w/view?usp=sharing