Lexical Analysis & Parsing Project

AIN SHAMS UNIVERSITY FACULTY OF ENGINEERING I-CREDIT HOURS ENGINEERING PROGRAMS COMPUTER ENGINEERING AND SOFTWARE SYSTEMS PROGRAM

2023-2024

Lexical Analysis & Parsing Project

DESIGN OF COMPILERS

CSE 439

Code & Demo Link: CLICK HERE

Name	**ID**
Omar Alaa Eldin Fareed Elnahass	21P0197
AbdulRahman Hesham Kamel Seleim Abduallah	21P0153
Ahmed Nezar Ahmed Hussien	21P0025
Kirollos Ehab Magdy Halim	21P0006
Tsneam Ahmed Eliwa Zky Mohamed Ghoname	21P0284

Course Coordinator:

Dr. Wafaa Samy & Eng. Ahmed Salama

Contents

1 INTRODUCTION

Within Computer Engineering, compiler development is an essential endeavour that makes it easier to convert readable source code into executable programs. Of all the programming languages, C is the most popular and reliable because of its efficiency and versatility. Because C is used widely in many different fields, it is necessary to optimise the compilation process for C-programmers.

This Project is focused on improving and perfecting the two core stages of the compilation process: syntax analysis and lexical analysis. These stages serve as the foundation for the compiler's interpretation and processing of the complexities of C code, converting it into an organised representation that is advantageous for further compilation steps.

2 C-LANGUAGE SPECIFICATION

2.1 Keywords

Keyword	Usage
auto	The auto keyword is rarely used in modern C programming. Historically, it was used to declare automatic variables with local scope, but since the C99 standard, it is not commonly used, as variables are by default automatic unless specified otherwise.
break	The break statement is used to exit from a loop or switch statement prematurely, causing the control to move to the next statement after the loop or switch.
case	The case keyword is used in a switch statement to define different possible values for the expression being tested. It is followed by constants and a colon.
char	char is a keyword used to declare a character data type. It typically represents a single byte and is used to store individual characters.
const	The const keyword is used to declare constants. Once a variable is declared as const, its value cannot be changed during the program execution.
continue	The continue statement is used to skip the rest of the code inside a loop and move to the next iteration of the loop.
default	In a switch statement, default is used to define the code that should be executed if none of the case values match the switch expression.
do	The do keyword is used to start a do-while loop. It ensures that the loop body is executed at least once before checking the loop condition.
double	double is used to declare a double-precision floating-point variable, capable of storing numbers with decimal points.
while	The while keyword is used to create a while loop. It repeatedly executes a block of code as long as the specified condition is true.
volatile	The volatile keyword is used to indicate to the compiler that a variable's value may change at any time without any action being taken by the code the compiler finds nearby.
void	void is used as a return type for functions that do not return a value. It is also used to indicate that a function takes no parameters.
unsigned	unsigned is used to declare variables that can only hold non-negative values (zero or positive).

Table 1: C Programming Keywords and Their Usage - Part 1

Keyword	Usage
switch	The switch statement is used to select one of many code blocks to be executed, based on the value of an expression.
struct	The struct keyword is used to define a structure, which is a user-defined data type that can contain members of different data types.
static	The static keyword is used to declare static variables, which retain their values between function calls.
sizeof	sizeof is an operator that is used to determine the size, in bytes, of a data type or a variable.
signed	signed is used to declare variables of signed data types, which can represent both positive and negative values.
short	short is used to declare variables of short integer data type, which typically uses fewer bytes than a regular integer.
return	The return statement is used to exit a function and return a value to the calling function.
register	The register keyword suggests to the compiler that a variable should be stored in a CPU register for faster access.
restrict	Tells the compiler that a particular pointer is the only one that can access the memory it points to during its lifetime, applied to pointer declarations.
long	long is used to declare variables of long integer data type, capable of holding larger integer values than a regular integer.
int	int is used to declare variables of integer data type, representing whole numbers.
if	The if keyword is used to conditionally execute a block of code based on the evaluation of a specified expression.
inline	Inlining is a compiler optimization technique where the compiler replaces a function call with the actual body of the function at the call site.
goto	The goto statement is used to transfer control to a labeled statement in the program. It is generally discouraged due to its potential to make code less readable and harder to maintain.
for	The for keyword is used to create a for loop, which allows for concise control flow and iteration over a range of values.
float	float is used to declare variables of single-precision floating-point data type.
extern	The extern keyword is used to declare variables and functions that are defined in other source files.
enum	The enum keyword is used to define an enumeration, a user-defined data type that consists of named integer constants.
else	The else keyword is used in conjunction with the if statement to specify a block of code that should be executed if the if condition is false.
union	The union keyword is used to define a data structure that can hold members of different data types, but only one member can be accessed at a time.
typedef	The typedef keyword is used to create aliases or alternative names for existing data types, making code more readable.

Table 2: C Programming Keywords and Their Usage - Part 2

2.2 Variable Identifiers

Variable identifiers are names given to memory locations where data is stored. They are used to represent different types of values such as integers, floating-point numbers, characters, etc. C variables must be identified with unique names. Variable names can be short, but it is better to be more descriptive to make your code more readable and understandable. For example, age, length, width.Variable identifiers follow certain rules and conventions:

1. Naming rules:
   - Names can contain letters, digits and underscores.
   - Names must begin with a letter or an underscore ('_').
   - Names are case-sensitive (myVar and myvar are different variables)
   - Names cannot contain whitespaces or special characters like '@','!','%', '#', 'ˆ', '&', ''.
   - Reserved words (such as int) cannot be used as names.
2. Naming Conventions :
   - It's a common convention to use meaningful names that reflect the purpose of the variable to make your code more readable and understandable. For example, age, length, width.
   - Variable names are usually written in lowercase letters, with words separated by underscores
   - Constants (variables whose values never change) are often written in all uppercase letters, with words separated by underscores.
3. Examples:

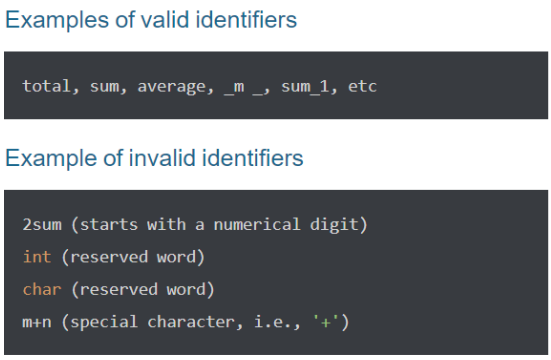


Figure 1: variable identifiers

4. Declaration:
   Before using variables.It needs to be declarared with its type you can refer to data types section for more information about different data types in C.Data type is written before variable identifier.
5. Scope :
   - The scope of a variable determines where in the program it can be accessed.
   - Variables can have either local or global scope.
   - The local scope is limited to the code or function in which the variable is declared.
   - Global variables can be used anywhere throughout the program.Global variables have a lifetime equivalent to the entire execution of the program.
6. LifeTime:
   - The lifetime of a variable is the duration for which the variable exists in memory.
   - Local variables have a lifetime limited to the execution of the block or function in which they are declared.
   - Global variable activates when they are called else they vanish when the function executes.Global variables have a lifetime equivalent to the entire execution of the program.

2.3 Function Identifiers

Function identifiers in the C programming language serve as unique names for defining and calling functions, essential components of program logic and structure. These identifiers adhere to specific naming conventions, beginning with a letter or underscore followed by letters, digits, or underscores. Function identifiers are case-sensitive, distinguishing between uppercase and lowercase characters. They must not coincide with reserved keywords or standard library functions, ensuring clarity and avoiding conflicts within the codebase. Function prototypes precede their usage, declaring the return type and parameter list, while function definitions provide the implementation. Scoped within blocks by default, function identifiers may have external linkage for broader accessibility or internal linkage for encapsulation within a translation unit. The table below delineates the detailed specifications and differences among function identifiers, elucidating their role and behavior within C programs.

Specification	**Description**
Naming Convention	Function identifiers must begin with a letter or underscore, followed by letters, digits, or underscores.
Length Limitations	Function identifiers can have any length, but only the first 31 characters are significant.
Case Sensitivity	Uppercase and lowercase letters are distinct in function identifiers.
Reserved Keywords	Function identifiers cannot coincide with C keywords.
Standard Library Functions	Certain function identifiers from the standard C library are reserved for predefined functions.
Function Prototypes	Functions must be declared before being used, specifying return type and parameter types.
Function Definition	Functions are defined with the syntax: `return_type function_name(parameter_list) { }.`
Scope	Function identifiers have block scope by default, accessible only within the block where they are defined.
External Linkage	Functions declared at file scope have external linkage by default, accessible from other translation units.
Static Functions	Functions declared with the static keyword have internal linkage, limited to the translation unit.

Table 3: Function Identifier Specifications in C Programming Language

Following the table, it's essential to recognize that adherence to these specifications is paramount for ensuring code correctness, maintainability, and portability in C programming. By strictly adhering to the defined naming conventions, developers can enhance code readability and reduce the likelihood of naming conflicts. Understanding the length limitations underscores the importance of choosing descriptive yet concise function names. The case sensitivity of identifiers highlights the need for consistency in naming conventions throughout the codebase.

Moreover, avoiding reserved keywords and standard library function names as identifiers prevents unintentional errors and ambiguities in code interpretation. Proper usage of function prototypes facilitates modular programming, enabling the compiler to perform type checking and ensure function compatibility. Clear function definitions, coupled with well-defined scopes and linkage, contribute to the modularity and organization of code.

By comprehensively understanding and applying these specifications, developers can create robust and maintainable C programs, fostering collaboration, extensibility, and ease of maintenance. Embracing these best practices not only improves individual coding proficiency but also contributes to the collective advancement of software engineering standards within the C programming community.

2.4 Data Types

Data types in the C programming language are foundational elements that define the characteristics and set of values that variables can hold. At its core, C provides a variety of built-in data types that allow programmers to perform mathematical, logical, and various other operations in an efficient manner. These data types are categorized into several groups:

Table 4: Comparison of C basic data types size and range on 32-bit and 64-bit CPUs.

C Basic Data Type	**32-bit CPU**		**64-bit CPU**	
(lr)2-3 (lr)4-5	**Size (bytes)**	**Range**	**Size (bytes)**	**Range**
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2.147e9 to 2.147e9	4	-2.147e9 to 2.147e9
long	4	-2.147e9 to 2.147e9	8	-9.223e18 to 9.223e18
long long	8	-9.223e18 to 9.223e18	8	-9.223e18 to 9.223e18
float	4	3.4e-38 to 3.4e+38	4	3.4e-38 to 3.4e+38
double	8	1.7e-308 to 1.7e+308	8	1.7e-308 to 1.7e+308

2.4.1 Basic data types

These include integer types (`int`), character types (`char`), floating-point types (`float`, `double`), and the `void` type, which represents the absence of value.

2.4.2 Derived data types

These types are formed from the basic data types and include arrays (collections of elements of a single type), structures (`struct` - a complex data type that can hold members of different types), unions (similar to structures, but the members share the same memory location), and pointers (variables that store memory addresses).

2.4.3 Enumeration data type

Defined by the keyword `enum`, this type allows the programmer to assign symbolic names to integral values, thereby increasing the readability of the code.

2.4.4 Type qualifiers

These include `const`, `volatile`, and `restrict`, which provide additional information about the variables they precede. For instance, `const` indicates that the variable's value cannot be modified after initialization.

Understanding data types is crucial in C programming as it directly impacts the memory usage, performance, and portability of software.

2.4.5 Floating-point types & Complex numbers

Floating-point types in C, such as `float`, `double`, and `long double`, are used for representing real numbers with fractional parts. These types adhere to the IEEE-754 standard, with `float` typically corresponding to the binary32 format and `double` to the binary64 format. The `long double` type's representation can vary, with some systems using an extended precision format like the 80-bit x87 format. These types can represent special values such as infinity, negative zero, and NaN (Not-a-Number), each having unique behaviors in mathematical operations and comparisons.

C also supports complex number types (`float _Complex`, `double _Complex`, and `long double _Complex`) for mathematical computations involving complex numbers, which consist of a real part and an imaginary part. These types are defined in the `<complex.h>` header and can be used with arithmetic operators and specific mathematical functions designed for complex numbers.

2.4.6 Integer types & Character types

C provides a range of integer types (`int`, `short`, `long`, `long long`) and their signed and unsigned variants to accommodate different ranges of values. The size and range of these types can vary between systems but are designed to be at least 16, 32, and 64 bits for `short`, `int/long`, and `long long`, respectively. The `char` type, used for character representation, can be `signed` or `unsigned`, and its size is guaranteed to be at least 8 bits.

Character types in C (`char`, `signed char`, `unsigned char`) are essentially small integers and can represent characters using character encoding like ASCII. The `char` type's signedness can vary by implementation, but it is distinct from `signed char` and `unsigned char`. C also includes wide character types (`wchar_t`, `char16_t`, `char32_t`) for representing larger character sets, such as Unicode.

2.4.7 Data models & Compatibility

The sizes of fundamental types in C can vary based on the data model used by a system (e.g., LP64, where `int` is 32 bits and `long` and pointers are 64 bits, commonly used in Unix and Unix-like systems). These variations impact the portability of code across different platforms.

C's type system allows for compatible types, meaning that different declarations of the same object or function do not need to use identical types, just compatible ones. This is particularly relevant for arrays, pointers, structures, unions, and function types, where compatibility rules ensure that types with the same structure or behavior are treated as equivalent, even if their declarations vary slightly.

2.5 Functions

- A function is a group of statements that together perform a task. Every C program has at least one function, which is `main()`, and all the most trivial programs can define additional functions.
- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.
- A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.
- The C standard library provides numerous built-in functions that your program can call. For example, `strcat()` to concatenate two strings, `memcpy()` to copy one memory location to another location, and many more functions.
- A function can also be referred to as a method or a sub-routine or a procedure, etc.

2.5.1 Defining a function

The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list ) {
   body of the function
}
```

2.5.2 Parts of a function

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function:

Element	Description
Return Type	A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the `return_type` is the keyword `void`.
Function Name	This is the actual name of the function. The function name and the parameter list together constitute the function signature.
Parameters	A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as an actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
Function Body	The function body contains a collection of statements that define what the function does.

Table 5: Components of a Function in C Programming

2.5.3 Example

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

2.5.4 Function declarations

- A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration has the following parts:

```
return_type function_name( parameter list );
```

- For the above defined function `max()`, the function declaration is as follows:

```
int max(int num1, int num2);
```

- Parameter names are not important in function declaration, only their type is required, so the following is also a valid declaration:

```
int max(int, int);
```

- Function declaration is required when you define a function in one source file and you call that function in another file. In such a case, you should declare the function at the top of the file calling the function.

2.5.5 Calling a function

- While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.
- When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.
- To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example:

```c
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

- We have kept `max()` along with `main()` and compiled the source code. While running the final executable, it would produce the following result:

```
Max value is : 200
```

2.5.6 Function arguments

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.
- Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways in which arguments can be passed to a function:
  - **Call by value:** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

```c
// C program to show use of call by value
#include <stdio.h>

void swap(int var1, int var2)
{
    int temp = var1;
    var1 = var2;
    var2 = temp;
}
// Driver code
int main()
{
    int var1 = 3, var2 = 2;
    printf("Before swap Value of var1 and var2 is: %d, %d\n",
                    var1, var2);
    swap(var1, var2);
    printf("After swap Value of var1 and var2 is: %d, %d",
                    var1, var2);
    return 0;}
```

  - **Call by reference:** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

```c
// C program to show use of call by Reference
#include <stdio.h>

void swap(int *var1, int *var2)
{
    int temp = *var1;
    *var1 = *var2;
    *var2 = temp;
}
// Driver code
int main()
{
    int var1 = 3, var2 = 2;
    printf("Before swap Value of var1 and var2 is: %d, %d\n",
                    var1, var2);
    swap(&var1, &var2);
    printf("After swap Value of var1 and var2 is: %d, %d",
                    var1, var2);
    return 0;}
```

- By default, C uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

2.5.7 Inline functions

Inline Function are those function whose definitions are small and be substituted at the place where its function call is happened. Function substitution is totally compiler choice. Let's take below example:

```c
#include <stdio.h>

// Inline function in C
inline int foo()
{
        return 2;
}

// Driver code
int main()
{

        int ret;

        // inline function call
        ret = foo();

        printf("Output is: %d\n", ret);
        return 0;
}
```

2.5.8 Advantages & Disadvantages of functions in C

Advantages	Disadvantages
1. The function can reduce the repetition of the same statements in the program. 2. The function makes code readable by providing modularity to our program. 3. There is no fixed number of calling functions; it can be called as many times as you want. 4. The function reduces the size of the program. 5. Once the function is declared, you can just use it without thinking about the internal working of the function.	1. Cannot return multiple values. 2. Memory and time overhead due to stack frame allocation and transfer of program control.

Table 6: Advantages and Disadvantages of Functions in C

2.6 Statements

2.6.1 Assignment statement

In C, assignment statements are used to assign values to variables. An assignment statement typically consists of a variable on the left-hand side and an expression on the right-hand side, separated by the assignment operator (=).

- Syntax: variable = expression;
  - The variable is on the left, and the expression is on the right of the assignment operator.
- Assignment Operator (=):
  - Used to assign the value of the expression on the right to the variable on the left.
- Type Compatibility:
  - The type of the expression must be compatible with the type of the variable.
- Evaluation Order:
  - The expression on the right is evaluated before the assignment takes place.
- Examples:

1. Simple assignment: int x = 10;

```c
#include <stdio.h>
int main() {
    int x;          // Declare variable
    x = 10;         // Assign value to x
    printf("x: %d\n", x);
    return 0;
}
```

2. Expression on the right: x = y + 5;

```c
#include <stdio.h>
int main() {
    int x = 5, y = 7;
    int result = x + y;   // Expression on the right
    printf("Result: %d\n", result);
    return 0;
}
```

3. Compound assignment: x += 2; (equivalent to x = x + 2;)

```c
#include <stdio.h>
  int main() {
      int x = 10;
      x += 5;             // Compound assignment equivalent to x = x + 5;
      printf("Updated x: %d\n", x); return 0;}
```

4. Type Compatibility:

```c
#include <stdio.h>
int main() {
    int x = 10;
    double y = 5.5;
    // x = y;    // Error: Type mismatch
    printf("x: %d\n", x);
    return 0;
}
```

2.6.2 Declaration statement

A declaration statement in C is used to declare a variable and specify its data type. It informs the compiler about the existence of a variable, allowing the program to reserve memory space for that variable.

- Syntax: datatype variable;
  - The datatype indicates the type of the variable being declared.
- Declaration and Initialization:
  - Variables can be declared and initialized in the same statement.
  - Initialization involves assigning an initial value to the variable.
- Multiple Declarations:
  - Multiple variables of the same type can be declared in a single statement.
- Scope:
  - The scope of a variable determines where it can be used within the program.
- Examples:
  1. Simple declaration: int x;

```c
#include <stdio.h>
int main() {
    int x;          // Declare variable
    x = 10;         // Assign value to x later
    printf("x: %d\n", x);
    return 0;
}
```

  2. Declaration with initialization: double pi = 3.14;

```c
#include <stdio.h>
int main() {
    double pi = 3.14;   // Declare and initialize
    printf("Value of pi: %f\n", pi);
    return 0; }
```

  3. Multiple declarations: int a, b, c;

```c
#include <stdio.h>
int main() {
    int a, b, c;    // Declare multiple variables
    a = 1;
    b = 2;
    c = a + b;
    printf("Sum of a and b: %d\n", c);
    return 0;
}
```

  4. Scope of Variable:

```c
#include <stdio.h>
int main() {
    int x = 5;      // Declare and initialize variable x
    {
        int y = 10; // Another variable y with limited scope
        printf("Sum: %d\n", x + y);
    }
    // printf("Sum: %d\n", x + y); // Error: y is not in scope here
    return 0;}
```

2.6.3 Return statement

C Return Statement

- The return statement in C ends the execution of a function and returns control to the function from where it was called.
- The return statement may or may not return a value depending on the return type of the function. For example, `int` returns an integer value, `void` returns nothing, etc.
- In C, we can only return a single value from the function using the return statement, and we have to declare the data type of the return value in the function definition/declaration.
- **Syntax**

```
return return_value;
```

There are various ways to use return statements. A few are mentioned below:

1. Methods not returning a value:

   (a) Not using a return statement in void return type function:
   In C, one cannot skip the return statement when the return type of the function is non-void type. The return statement can be skipped only for void types.
   Syntax:

```
void func()
{
    .
    .
    .
}
```

   (b) - Using the return statement in the void return type function:
   - As void means empty, we don't need to return anything, but we can use the return statement inside void functions as shown below. Although, we still cannot return any value.
   Syntax:

```
void func()
{
    return;
}
```

   - This syntax is used in function as a jump statement in order to break the flow of the function and jump out of it. One can think of it as an alternative to "break statement" to use in functions.
   - But if the return statement tries to return a value in a void return type function, that will lead to errors.
   Incorrect Syntax:

```
void func()
{
    return value;
}
```

2. Methods returning a value:

   For functions that define a non-void return type in the definition and declaration, the return statement must be immediately followed by the return value of that specified return type.

   Syntax:

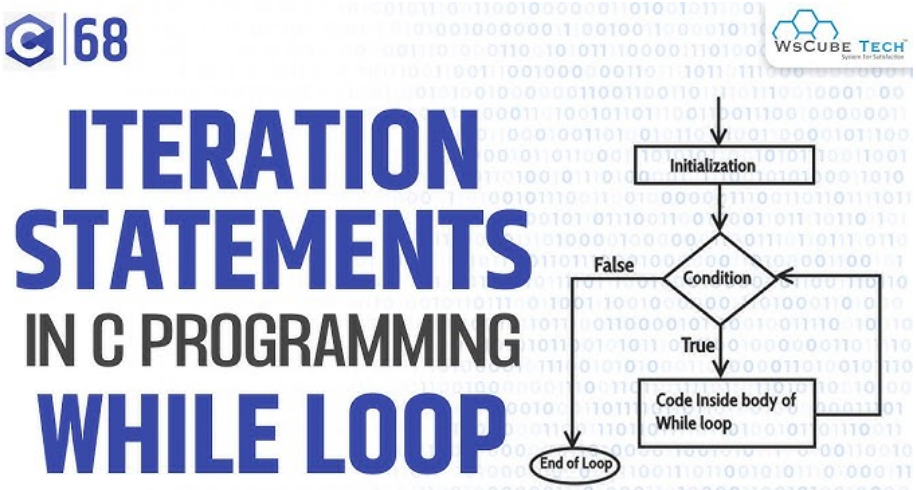
```
return-type func()
{
    return value;
}
```

   Example:

```
// C code to illustrate Methods returning
// a value using return statement

#include <stdio.h>

// non-void return type
// function to calculate sum
int SUM(int a, int b)
{
    int s1 = a + b;

    // method using the return
    // statement to return a value
    return s1;
}

// Driver method
int main()
{
    int num1 = 10;
    int num2 = 10;
    int sum_of = SUM(num1, num2);
    printf("The sum is %d", sum_of);
    return 0;
}
```

2.6.4 Iterative statement

- An iterative statement, often referred to as a loop, in C is a control flow statement that allows the execution of a block of code repeatedly based on a condition. It is used when you need to execute a certain block of code multiple times until a specified condition is met.



- In C, there are several types of iterative statements or loops:

1. **for loop:** Executes a block of code repeatedly until a specified condition evaluates to false. It consists of three parts: initialization, condition, and update.

   ```
   for (initialization; condition; update) {
       // Code to be repeated
   }
   ```

2. **while loop:** Executes a block of code repeatedly as long as a specified condition evaluates to true.

   ```
   while (condition) {
       // Code to be repeated
   }
   ```

3. **do-while loop:** Similar to the while loop, but it always executes the block of code at least once before checking the condition.

   ```
   do {
       // Code to be repeated
   } while (condition);
   ```

- Iterative statements are fundamental in programming and are used extensively to automate repetitive tasks, such as iterating through arrays, processing input data, or implementing algorithms like searching and sorting.

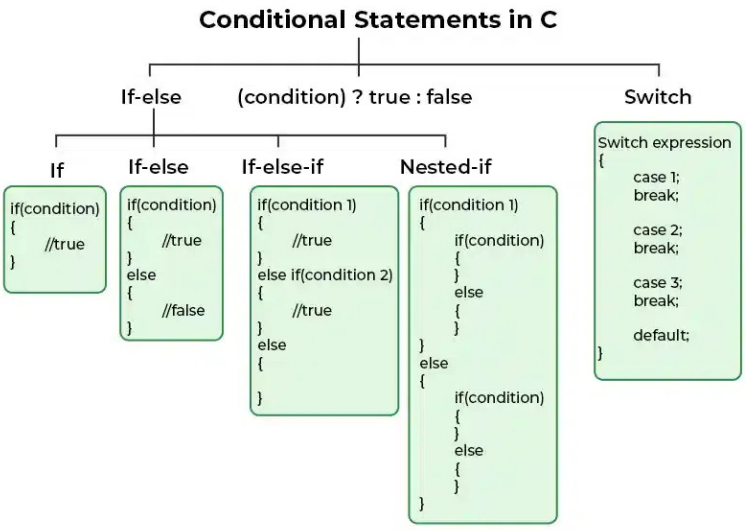
for loop	while loop	do-while loop
• Convenient when the number of iterations is known beforehand. • Initialization, condition, and update are all contained within the loop structure, making it concise and easy to read. • Suitable for iterating over arrays or sequences with a fixed number of elements.	• Provides flexibility as the loop condition can be based on any expression. • Useful when the number of iterations is not fixed or known beforehand. • Can be used to implement infinite loops or loops with complex termination conditions.	• Ensures that the loop body executes at least once, even if the condition is initially false. • Useful when you want to execute the loop body at least once before checking the loop condition. • Often used when input validation is required, as it guarantees the execution of validation code at least once.

Table 7: Comparison of for, while, and do-while loops

2.6.5 Conditional statement

In the C programming language, statements play a crucial role in controlling the flow of execution and performing various operations within a program. Among the types of statements, conditional statements and function call statements are fundamental constructs for decision-making and modular programming, respectively.

Conditional statements allow programmers to execute certain blocks of code based on specific conditions. In C, the most common conditional statements are the if, else if, and else statements.



The 'if' statement evaluates a condition and executes a block of code if the condition is true. If the condition is false, the block is skipped.

The 'else if' statement provides an alternative condition to be checked if the preceding if or else if condition(s) are false.

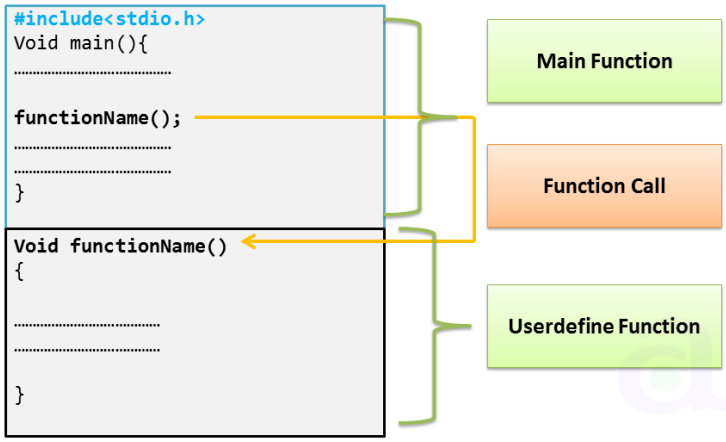
The 'else' statement is optional and provides a default block of code to execute if none of the preceding conditions are true.

Example:

```
int num = 10;
if(num > 0) {
    printf("Positive\n");
} else if(num < 0) {
    printf("Negative\n");
} else {
    printf("Zero\n");
}
```

2.6.6 Function call statement

Function call statements invoke functions, allowing code to be modularized and reused. In C, a function call statement consists of the function name followed by parentheses containing any required arguments.



Example:

```
#include <stdio.h>

// Function prototype
void greet(char name[]);

int main() {
    greet("John"); // Function call statement
    return 0;
}

// Function definition
void greet(char name[]) {
    printf("Hello, %s!\n", name);
}
```

Function call statements are essential for organizing code into manageable units, enhancing readability, and promoting code reuse.

To Sum up, conditional statements in C enable branching based on conditions, facilitating decision-making within programs, while function call statements promote modularity and code reuse by invoking reusable blocks of code. Both constructs are fundamental to the structured design and implementation of C programs.

2.7 Expressions

An expression in C language is combination of constants, variables, operators, and function calls that are evaluated to produce a single value. These expressions are evaluated and can be used to assign values to variables, perform mathematical operations, or execute different actions such as comparison and Boolean logic. They can be as simple as a single variable or a complex combination of operators and operands. For example:

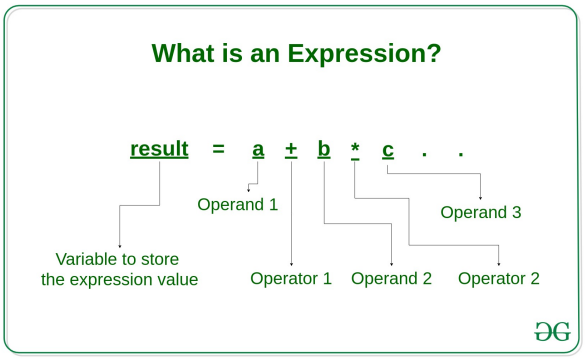


Figure 2: Expression

2.7.1 Arithmetic

An arithmetic expression consists of operands and arithmetic operators. It performs computations on the int, float, or double type values. Arithmetic operations can be performed in a single line of code or multiple lines combined with arithmetic operations such as addition, subtraction, multiplication, and division. The following types of arithmetic expressions are there:

- **integer expression** - an expression containing only integral operands.
- **real expression** - an expression containing only real operands.
- **mixed mode expression** - an expression containing both integral and real operands

Examples:

Example

Evaluation of expression	Description of each operation
6*2/(2+1 * 2/3 +6) +8 * (8/4)	An expression is given.
6*2/(2+2/3 + 6) + 8 * (8/4)	2 is multiplied by 1, giving the value 2.

Figure 3: Arithmetic Examples

2.7.2 Boolean

Boolean expressions in C evaluate to either true or false. These expressions are commonly used in decision-making constructs like if statements, while loops, and for loops. Boolean expressions often involve comparison operators such as equal to (==), not equal to (!=), greater than (¿), less than (¡), greater than or equal to (¿=), and less than or equal to (¡=). For example:

Logical Expressions	Description
(x > 4) && (x < 6)	This logical expression is used as a test condition to check if the x is greater than 4 and the x is less than 6. The result of the condition is true only when both conditions are true.
x > 10 || y <11	This logical expression is used as a test condition to check if x is greater than 10 or y is less than 11. The result of the test condition is true if either of the conditions holds true value.
! (x > 10) && (y = = 2)	This logical expression is used as a test condition to check if x is not greater than 10 and y is equal to 2. The result of the condition is true if both the conditions are true

Figure 4: Boolean Expressions

2.8 C Operator Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Figure 5: Operators Percedence

2.9 Punctuation

Table 8: Punctuation Symbols in C

Symbol	Description
{ }	Delimits struct or union definitions, enum definitions, compound statements, and initializers.
[ ]	Subscript operator, array declarator, designator for array elements (since C99).
#	Introduces preprocessing directives, stringification operator, and token pasting operator.
##	The processor operator for token pasting.
()	Indicates grouping in expressions, function call operator, delimits operands in sizeof, _Alignof, typeof, expressions.
:	Part of conditional operator, label declaration, introduces width in bit-field member declarations.
;	Indicates the end of a statement (including the init-statement of a for statement), separates clauses in a for statement, and delimits struct-declaration-list.
...	Variadic function parameters, ellipsis in function declarations.
?	Part of conditional Operator
.	Member access operator. In initialization, introduces a designator for a struct/union member (since C99).
− >	Member access operator (pointer-to-member).
	Unary complement operator (bitwise not operator).
!	Logical not operator.
+	Unary plus operator. Binary plus operator.
-	Unary minus operator. Binary minus operator.
*	Indirection operator (dereference). Multiplication operator. Pointer operator in a declarator or type-id. Placeholder for the length of a variable-length array declarator in a function declaration (since C99).
/	Division operator.
%	Modulo operator.
	Bitwise XOR operator.
&	Address-of operator. Bitwise AND operator.
‖	Bitwise OR operator.
=	Simple assignment operator. In initialization, delimits the object and the initializer list. In an enum definition, introduces the value of an enumeration constant.
+ =	Compound assignment operator (addition).
-=	Compound assignment operator (subtraction).
*=	Compound assignment operator (multiplication).
/=	Compound assignment operator (division).
%=	Compound assignment operator (modulo).
=	Compound assignment operator (bitwise XOR).
&=	Compound assignment operator (bitwise AND).
—=	Compound assignment operator (bitwise OR).
= =	Equality operator.
!=	Inequality operator.
<	Less-than operator. Introduces a header name in a include directive, or implementation-defined locations within a pragma directive.

Table 9: Punctuation Symbols in C

Symbol	Description
>	Greater-than operator. Indicates the end of a header name in a include directive, or implementation-defined locations within a pragma directive.
<=	Less-than-or-equal-to operator.
>=	Greater-than-or-equal-to operator.
&&	Logical AND operator.
||	Logical OR operator.
<<	Bitwise shift left operator.
>>	Bitwise shift right operator.
++	Increment operator. Increases the value of a variable by 1.
--	Decrement operator. Decreases the value of a variable by 1.
,	Comma operator. Separates expressions in a list (e.g., function arguments, initializer lists).
<<=	Compound assignment operator.
>>=	Compound assignment operator.
//	Line comments allow you to include notes and explanations within your code. The compiler ignores anything after the // on a line. Use these comments to make your code easier to understand, both for yourself and for others who might work with it.
/**/	Block comments, enclosed within /* and */, let you insert multi-line notes or explanations directly into your code. Anything within these markers is ignored by the compiler. Use block comments for longer descriptions, for commenting out sections of code during testing, or to provide detailed insights into your code's logic.

3 LEXICAL ANALYSIS

3.1 Introduction

Lexical analysis, also known as scanning, is the first phase of a compiler. Its main task is to read the input characters of the source code, grouping them into meaningful sequences called lexemes, and then convert these lexemes into tokens. Tokens are the fundamental units that the syntax analysis phase uses to understand the structure of the programming language.

3.1.1 Lexical Analysis definition

Lexical Analysis,also known as Scanner ,is the initial phase of a compiler.Its primary purpose is to transform a high-level input program into a sequence of tokens.Lexical Analysis can be implemented using Deterministic Finite Automata.
Input: Stream of characters.
Output: The output is a sequence of tokens that is sent to the parser for syntax analysis.

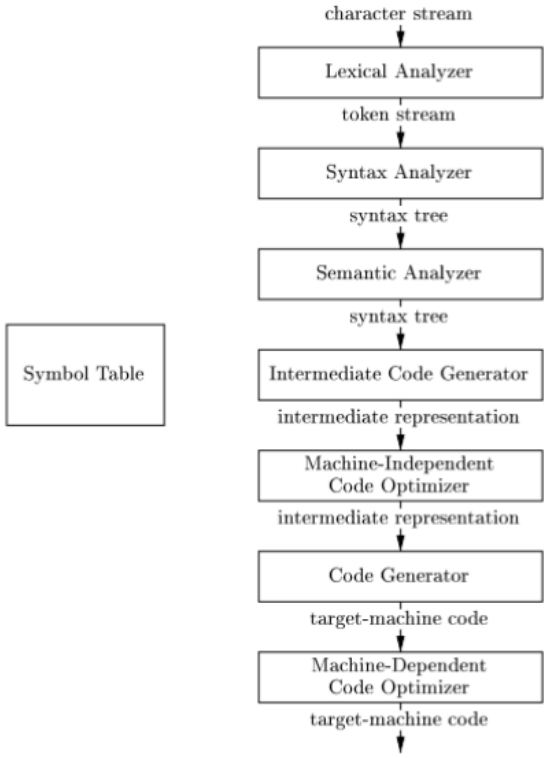


Figure 6: Phases Of compilers

3.1.2 Tokens

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
Examples of tokens include:

- Type tokens (e.g., identifiers, numbers, real values).
- Punctuation tokens (e.g., keywords like “if,” “void,” “return”).
- Alphabetic tokens (e.g., keywords such as “for,” “while,” “if”).

Example of Non-Tokens: Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

3.1.3 Lexemes

A lexeme refers to the sequence of characters matched by a pattern to form a corresponding token. In other words, it's the input characters that make up a single token. **Examples of lexemes include:**

- "float"
- "abs_zero_Kelvin"
- "="
- "-"
- "273"
- ";"

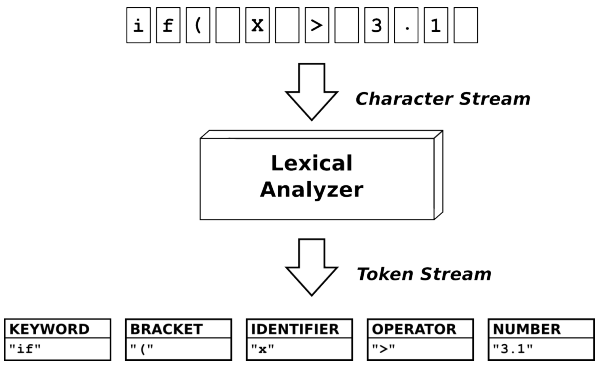


Figure 7: Lexical Analysis Process

3.1.4 How Lexical Analysis works?

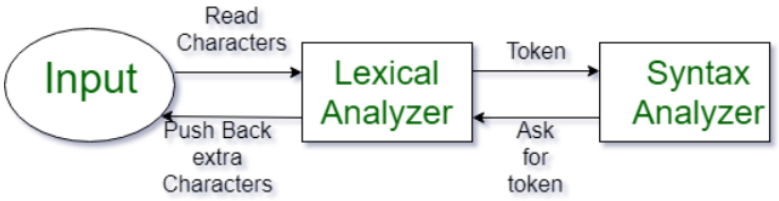


Figure 8: Lexical Analysis

1. **Input Preprocessing:** Clean up the input text by removing comments, whitespace, and non-essential characters.
2. **Tokenization:** Break the input text into a sequence of tokens by matching characters against predefined patterns or regular expressions.
3. **Token Classification:** Determine the type of each token (e.g., keywords, identifiers, operators).
4. **Token Validation:** Check that each token adheres to the language rules (e.g., valid variable names, correct syntax for operators).
5. **Output Generation:** Produce a list of tokens, which can be passed to subsequent compilation stages.

Lexeme	token
while	while
(	lparen
y	identifier
>=	Comparison
t	identifier
)	Rparen
y	identifier
=	Assignment
y	identifier
-	Arithmetic
3	integer
;	Finish of a statement

Figure 9: Lexemes and Tokens

example, consider the program

```
int main()
{
  // 2 variables
  int a, b;
  a = 10;
 return 0;
}
```

All the valid tokens are:

```
'int'  'main'  '('  ')'  '{'  'int'  'a' ','  'b'  ';'
 'a'  '='  '10'  ';' 'return'  '0'  ';'  '}'
```

Figure 10: Valid Tokens Example

3.1.5 Error detection in Lexical Analysis

The lexical analyzer detects errors based on the grammar rules of the programming language (such as C). When an error occurs, it reports the specific row and column numbers where the issue was found.

3.1.6 Advantages of Lexical Analysis

1. **Efficient Parsing:** Lexical analysis tokenizes the input program, making subsequent parsing and semantic analysis more streamlined. By breaking down the code into smaller units (tokens), it simplifies the overall compilation process.

2. **Early Error Detection:** Lexical analysis identifies lexical errors (such as misspelled keywords or undefined symbols) at an early stage in the compilation process. This proactive error checking enhances the overall efficiency of the compiler or interpreter by catching issues sooner.

3. **Enhanced Efficiency:** Once the source code is tokenized, subsequent compilation or interpretation phases operate more efficiently. Parsing and semantic analysis benefit from working with this streamlined tokenized input.

3.1.7 Disadvantages of Lexical Analysis

1. **Limited Context:**Lexical analysis operates based on individual tokens and does not take into account the overall context of the code. This limitation can sometimes lead to ambiguity or misinterpretation of the code's intended meaning, especially in languages with complex syntax or semantics.

2. **Overhead:**While lexical analysis is necessary for the compilation process, it introduces an additional layer of overhead. Tokenizing the source code requires extra computational resources, which can impact the overall performance of the compiler.

3. **Debugging Challenges:**Lexical errors detected during the analysis phase may not always provide clear indications of their origins in the original source code. Debugging such errors can be challenging, especially if they result from subtle mistakes in the lexical analysis process.

3.2 Regular Expressions

3.2.1 Definition

Regular expressions (regex) are patterns used for matching character combinations in strings. They are widely used in text processing and are supported by many programming languages and tools. Here's a basic explanation of some common components of regular expressions:

Components of Regular Expressions:

1. **Literals:** Literals are characters that match themselves. For example, the regular expression `hello` matches the string "hello" exactly.

2. **Metacharacters:** Metacharacters are special characters with predefined meanings in regular expressions. Some common metacharacters include:

   - `.` (dot): Matches any single character except newline.
   - `|` (pipe): Alternation operator, matches either the expression before or after it.
   - `*`: Matches zero or more occurrences of the preceding character or group.
   - `+`: Matches one or more occurrences of the preceding character or group.
   - `?`: Matches zero or one occurrence of the preceding character or group.
   - `^`: Anchors the match to the beginning of the string.
   - `$`: Anchors the match to the end of the string.
   - \: Escape character, allows using metacharacters as literals.
   - `[ ]`: Character class, matches any single character within the brackets.
   - `^[ ]`: Negated character class, matches any single character not within the brackets.

3. **Quantifiers:** Quantifiers specify the number of occurrences of the preceding character or group that should be matched. Some common quantifiers include:

   - `*`: Matches zero or more occurrences.
   - `+`: Matches one or more occurrences.
   - `?`: Matches zero or one occurrence.
   - {`n`}: Matches exactly `n` occurrences.
   - {`n,`}: Matches `n` or more occurrences.
   - {`n,m`}: Matches between `n` and `m` occurrences (inclusive).

4. **Grouping:** Parentheses (`( )`) are used to group characters or subexpressions together. They are used to apply quantifiers and other operators to multiple characters.

5. **Anchors:** Anchors are used to specify positions in the string where matches should occur. Common anchors include:
    - `^`: Matches the beginning of a line.
    - `$`: Matches the end of a line.
    - `\b`: Matches a word boundary.
6. **Escape Sequences:** Escape sequences allow including special characters as literals. For example, `\.` matches a literal period instead of the special meaning of `.`.

How Regular Expressions Work: When you apply a regular expression to a string, the regex engine searches for matches by:

- Starting at the beginning of the string (or as specified by anchors).
- Attempting to match the pattern specified by the regular expression.
- Moving along the string one character at a time, trying to match the pattern.
- Backtracking when necessary if a match is not found.
- Returning all matches found or performing the specified action (e.g., replacement).

Operator	**Description**
*	Zero or more occurrences of the preceding element
?	Zero or one occurrence of the preceding element
+	One or more occurrences of the preceding element
\	Escape character for special characters
.	Any single character except newline
()	Grouping for capturing subexpressions
	Character class, matches any character within the brackets
—	Alternation, matches either the expression before or after
{}	Specifies the exact number of occurrences or a range
^	Anchors the match to the beginning of the line
$	Anchors the match to the end of the line

Table 10: Regular Expression Operators

3.2.2 Implemented regular expressions

```
String num_regex = "\\d+(\\.\\d+)?((e|E)(['+']|-)?\\d+)?";
String binary_octal_hex_regex = "(((0b|0B)[0-1]+)|(0[0-7]+)|((0x|0X)[0-9a-fA-F]+))";
String bad_binary_octal_hex_regex = "(0b|0B)[0-9]+|0[0-9]+|(0x|0X)[0-9a-zA-Z]+";
String num_dataType_regex = "(ULL|LL|L|UL|F|ull|ll|l|ul|f)?";
String bad_Identifiers = "\\d+[a-zA-Z_#$@]+\\d*|[a-zA-Z_#$@]+[#$@]+[a-zA-Z_#$@\\d]+";
Pattern tokenPattern = Pattern.compile(
        "\"(?:[^\"\\\\]|\\\\.)*\"|" +             // Match double-quoted strings
        bad_binary_octal_hex_regex + "|" +  // Match binary, octal, decimal, and hexadecimal numbers
        binary_octal_hex_regex + "|" +  // Match binary, octal, decimal, and hexadecimal numbers
        "'.'|" +                    // Match single characters within single quotes
        "\"|'|" +                   // Match bad puncatuation
        bad_Identifiers + "|" +     // Match bad variable names
        num_regex + num_dataType_regex + "|" +           // Match numbers
        "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b(\\()|" + // Match identifiers (Function)
        "\\}(?:\\s*\\w+)?\\s*;|" + // Match End Block of Struct
        "^(struct|typedef( )struct)\\s+(\\w+)\\s*[\\{|;]|" + // Match identifiers (Struct)
        "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b((?!\\()|(?!\\{))|" + // Match identifiers (Variable)
        "\\(|\\)|\\{|\\}|;|,|\\[|\\]|" +    // Match parentheses, braces, semicolon, comma
        "\\+\\+|--|=|≠|≤|≥|&&|" + // Match comparison and logical operators
        "\\-\\>|\\+\\=|\\-\\=|\\*\\=|\\/\\=|\\%\\=|\\&\\=|\\|\\=|\\|\\||" + // Match compound assignment operators
        "\\+|\\*|/|%|\\^\\=|\\<\\<\\=|\\>\\>\\=|\\<\\◁\\>\\>|◁▷|" + // Match arithmetic and bitwise operators
        "\\-|\\+|\\*|\\=|\\&\\&|\\|\\||\\!|\\&|\\||\\^|\\~|\\?|" + // Match miscellaneous operators
        "\\:|\\+\\+|\\-\\-|\\." // Match other operators
);
```

Figure 11: Regex Code

1. `\\d+(\\.\\d+)?((e|E)(['+']|-)?\\d+)?`
   - This regular expression matches numeric literals with optional decimal points and exponent notation.
   - `\\d+`: Matches one or more digits.
   - `(\\.\\d+)?`: Matches an optional decimal point followed by one or more digits.
   - `((e|E)(['+']|-)?\\d+)?`: Matches an optional exponent part, including 'e' or 'E', followed by an optional sign ('+' or '-'), and one or more digits.
2. `(((0b|0B)[0-1]++)|(0[0-7]+)|((0x|0X)[0-9a-fA-F]+))`
   - This regular expression matches binary, octal, and hexadecimal literals.
   - It consists of three main alternatives:
     - `(0b|0B)[0-1]++`: Matches binary literals prefixed with '0b' or '0B', followed by one or more binary digits.
     - `0[0-7]+`: Matches octal literals prefixed with '0', followed by one or more octal digits.
     - `(0x|0X)[0-9a-fA-F]+`: Matches hexadecimal literals prefixed with '0x' or '0X', followed by one or more hexadecimal digits.
3. `(0b|0B)[0-9]+|0[0-9]+|(0x|0X)[0-9a-zA-Z]+`
   - This regular expression matches bad binary, octal, and hexadecimal values in a string.
   - `(0b0B)[0-9]+`:
     - `(0b0B)`—: This is a capturing group that matches either "0b" or "0B", indicating the prefix for binary literals, regardless of case sensitivity.
     - `[0-9]+`: Matches one or more digits. However, in binary literals, only digits 0 and 1 are allowed, but this regex allows any digit.
     - This part of the regex matches bad binary values.

- `0[0-9]+`:
  - Matches bad octal values.
  - `0`: Matches the prefix "0" indicating an octal literal.
  - `[0-9]+`: Matches one or more digits. However, in octal literals, only digits 0-7 are allowed, but this regex allows any digit after the leading zero.
- `(0x0X)[0-9a-zA-Z]+`:
  - This part matches bad hexadecimal values.
  - `(0x0X)`—: This capturing group matches either "0x" or "0X", indicating the prefix for hexadecimal literals, regardless of case sensitivity.
  - `[0-9a-zA-Z]+`: Matches one or more characters that can be digits (0-9) or letters (a-z, A-Z). However, in hexadecimal literals, only digits 0-9 and letters a-f or A-F are allowed, but this regex allows any alphanumeric character.

4. `(ULL|LL|L|UL|F|ull|ll|l|ul|f)?`

   - This regular expression matches optional C/C++ data type suffixes for numeric literals.
   - It allows for various combinations of uppercase and lowercase letters representing data types, followed by an optional occurrence of those letters.

5. `\\d+[a-zA-Z_#$@]+\\d*|[a-zA-Z_#$@]+[#$@]+[a-zA-Z_#$@\\d]+`

   - This regular expression identifies strings that likely represent invalid or unconventional identifiers. It matches two distinct cases:

     Case 1:

     - Begins with one or more digits (‘`\d+`‘)
     - Followed by one or more letters, underscores, #, $, or symbols (‘`[a-zA-Z_#$@]+`‘)
     - May optionally end with zero or more digits (‘`\d*`‘)

     Case 2:

     - Begins with one or more letters, underscores, #, $, or symbols (‘`[a-zA-Z_#$@]+`‘)
     - Must include one or more #, $, or symbols immediately after (‘`[#$@]+`‘)
     - Followed by one or more letters, underscores, #, $, symbols, or digits (‘`[a-zA-Z_#$@\d]+`‘)

   - **Examples of matches:**
     - 123abc
     - abc123
     - _hello#
     - my_Var#1
     - hello#$world
     - var##$123
   - **Purpose:** This regular expression is likely intended for programming language syntax validation. Typical identifier conventions discourage starting with numbers or the excessive use of special characters like #, $, and .

6. `\"(?:[^\"\\\\]|\\\\.)*\"`

   - Matches double-quoted strings.
   - Allows for any character except a double quote or a backslash inside the quotes unless the backslash is escaping another character.

7. `'.'`

   - Matches a single character enclosed in single quotes.

8. `\"|'`

   - Matches a single double quote or single quote, representing bad punctuation when not closed.

9. `\\b[a-zA-Z_][a-zA-Z0-9_]*\\b(\\(`

   - Matches identifiers that are immediately followed by an opening parenthesis '(', suggesting function names.
   - Begins with a letter or underscore, followed by any number of letters, digits, or underscores.

10. `\}(?:\s*\w+)?\s*;`

    - Matches the end of a block for structures, possibly followed by a word (identifier) and then a semicolon.

11. `^(struct|typedef( )struct)\\s+(\\w+)\\s*[\\{|;]`

    - Matches the declaration of a structure, starting at the beginning of a line.
    - Followed by an optional 'typedef', then the structure name, and possibly a curly brace or semicolon.

12. `\\b[a-zA-Z_][a-zA-Z0-9_]*\\b((?!\\()|(?!\\{))`

    - Matches identifiers that are not immediately followed by an opening parenthesis '(' or a curly brace '', suggesting variable names.

13. `\\(|\\)|\\{|\\}|;|,`

    - Matches parentheses, braces, semicolons, and commas.

14. `\\+\\+|--|==|!=|<=|>=|&&`

    - Matches various comparison and logical operators:
      - `++`: Increment operator.
      - `--`: Decrement operator.
      - `==`: Equality comparison operator.
      - `!=`: Inequality comparison operator.
      - `<=`: Less than or equal to comparison operator.
      - `>=`: Greater than or equal to comparison operator.
      - `&&`: Logical AND operator.

15. `\\-\\>|\\+\\=|\\-\\=|\\*\\=|\\/\\=|\\%\\=|\\&\\=|\\|\\=|\\|\\|`

    - Matches compound assignment operators:
      - `->`: Structure pointer access operator.
      - `+=`: Addition assignment operator.
      - `-=`: Subtraction assignment operator.
      - `*=`: Multiplication assignment operator.
      - `/=`: Division assignment operator.
      - `%=`: Modulus assignment operator.
      - `&=`: Bitwise AND assignment operator.
      - `||=`: Logical OR assignment operator.

16. \\+|*|/|%|\\^\=|\\<\\<\\=|\\>\\>\\=|\\<\\<|\\>\\>|<|>

- Matches arithmetic and bitwise operators:
  - `+`: Addition operator.
  - `*`: Multiplication operator.
  - `/`: Division operator.
  - `%`: Modulus operator.
  - `^=`: Bitwise XOR assignment operator.
  - `<<=`: Left shift assignment operator.
  - `>>=`: Right shift assignment operator.
  - `<<`: Left shift operator.
  - `>>`: Right shift operator.
  - `<`: Less than operator.
  - `>`: Greater than operator.

17. \\-|\\+|*|\\=|\\&\\&|\\|\\||\\!|\\&|\\||\\^|\\~|\\?

- Matches various miscellaneous operators:
  - `-`: Subtraction operator.
  - `+`: Addition operator.
  - `*`: Multiplication operator.
  - `=`: Assignment operator.
  - `&&`: Logical AND operator.
  - `||`: Logical OR operator.
  - `!`: Logical NOT operator.
  - `&`: Bitwise AND operator.
  - `|`: Bitwise OR operator.
  - `^`: Bitwise XOR operator.
  - `~`: Bitwise complement operator.
  - `?`: Ternary conditional operator.

18. \\:|\\+\\+|\\-\\-|\\.

- Matches colons, increment, decrement, and the dot operator:
  - `::` Colon operator.
  - `++`: Increment operator.
  - `--`: Decrement operator.
  - `.`: Dot operator.

3.3 Code Explanation

Our Compiler implementation works under the work of 5 classes, 3 classes have the logic & 2 classes handles GUI.

3.3.1 C file Reader class

Initialization

The `cReader` class is initialized with two instance variables: `path` to hold the file path of the .c file, and `cLines` to store the lines of the .c file.

```
private String path;
private ArrayList<String> cLines = new ArrayList<String>();
```

Reading and Cleaning Lines

The `getClines()` method reads each line from the specified .c file, removes comments (both single-line and multi-line) and preprocessors, and stores the cleaned lines in the `cLines` ArrayList.

```
public ArrayList<String> getClines() {
    // Specify the path to your .c file
    String filePath = path;

    try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
        String line;

        // Read file line by line
        while ((line = br.readLine()) != null) {
            // Process each line as needed
            cLines.add(line);
        }
    } catch (IOException e) {
        // Handle any IO exceptions
        e.printStackTrace();
    }
    cleanClines();
    return cLines;
}
```

Cleaning Process

- Single-line comments (`//`) are detected and removed entirely from each line.
- Multi-line comments (`/* ... */`) are identified and replaced with an empty string.
- Preprocessor directives (`#`) are detected and removed from the line.

Handling Block Comments

A boolean flag (`blockComment`) is used to handle multi-line comments. It is set when encountering `/*` and unset when encountering `*/`. Lines within block comments are replaced with an empty string until the end of the comment block is reached.

```
private void cleanClines(){
    boolean blockComment = false;
    for (int i = 0; i < cLines.size(); i++) {
        String line = cLines.get(i);
        if (blockComment) {
            if (line.matches("(.)*\\*/")) {
                blockComment = false;
            }
            cLines.set(i, "");
        } else {
            if (line.matches("(.)*//(.)*")){
                cLines.set(i, cLines.get(i).replaceAll("//(.)*", ""));
            }
            else if (line.matches("(.)*/\\*(.)*\\*/")){
                cLines.set(i, cLines.get(i).replaceAll("/\\*(.)*\\*/", ""));
            }
            else if (line.matches("(.)*/\\*(.)*")) {
                blockComment = true;
                cLines.set(i, cLines.get(i).replaceAll("/\\*(.)*", ""));
            }
        }
        if (line.matches("(.)*#(.)*")){
            cLines.set(i, cLines.get(i).replaceAll("#(.)*", ""));
        }
    }
}
```

Providing Access and Clearing

The `cReader` class offers methods to access the cleaned lines (`getClines()`) and the original lines (`cLinesGetter()`). It also provides a method (`clear()`) to clear the stored lines from the `cLines` ArrayList.

```
public ArrayList<String> cLinesGetter() {
    return cLines;
}

public void clear(){
    cLines.clear();
}
```

Usage

Users can instantiate the `cReader` class with or without a file path, then call the `getClines()` method to read and prepossess the lines of the specified .c file. After prepossessing, the cleaned lines can be accessed for further processing or analysis.

3.3.2 Lexer class

Brief Overview

The `Lexer` class is responsible for tokenizing C code files. Tokenization involves breaking down the source code into smaller units called tokens, which represent the basic building blocks of the code.

Initialization

The `Lexer` class initializes a `cReader` object to read the C code file. It also defines predefined tokens for operators, punctuation, and keywords. We create 3 Array Lists to store Predefined operators, predefined & predefined keywords.

```
public class Lexer {
    private static Dictionary<String, ArrayList<String>> predefinedTokens = new Hashtable<>();
    public cReader reader;
    private ArrayList<Token> tk = new ArrayList<>();

    public Lexer(String path){
        reader = new cReader(path);
        ArrayList<String> cOperators = new ArrayList<>();
        ArrayList<String> cPunctuation = new ArrayList<>();
        ArrayList<String> cKeywords = new ArrayList<>();
                // Add C-language operators to the ArrayList
        cOperators.add("+");cOperators.add("-");
        cOperators.add("/");cOperators.add("%");
        cOperators.add("=");cOperators.add("==");
        cOperators.add(">");cOperators.add("<");
        cOperators.add(">=");cOperators.add("<=");
        cOperators.add("||");cOperators.add("!");
        cOperators.add("&");cOperators.add("|");
        cOperators.add("^");cOperators.add("~");
        cOperators.add("<<");cOperators.add("*");
        cOperators.add(">>");cOperators.add(">>>");
        cOperators.add("?:");cOperators.add("++");
        cOperators.add("--");cOperators.add("!=");
        cOperators.add("+=");cOperators.add("-=");
        cOperators.add("*=");cOperators.add("&&");
        cOperators.add("/=");cOperators.add("%=");
        cOperators.add("&=");cOperators.add("|=");
        cOperators.add("^=");cOperators.add("<<=");
        cOperators.add(">>=");cOperators.add(".");
        cOperators.add(">>>=");cOperators.add("->");
        cOperators.add("?");cOperators.add(":");

        // Add C-language punctuations to the ArrayList
        cPunctuation.add("(");cPunctuation.add(")");
        cPunctuation.add("{");cPunctuation.add("}");
        cPunctuation.add("]");cPunctuation.add(";");
        cPunctuation.add(",");cPunctuation.add("[");
        cPunctuation.add("'");cPunctuation.add("\"");

        // Add C-language keywords to the ArrayList
        cKeywords.add("auto");cKeywords.add("break");
        cKeywords.add("case");cKeywords.add("char");
        cKeywords.add("const");cKeywords.add("continue");
        cKeywords.add("default");cKeywords.add("do");
```

```
        cKeywords.add("double");cKeywords.add("else");cKeywords.add("enum");
        cKeywords.add("extern");
        cKeywords.add("float");cKeywords.add("for");cKeywords.add("goto");
        cKeywords.add("if");
        cKeywords.add("int");cKeywords.add("long");cKeywords.add("register");
        cKeywords.add("return");
        cKeywords.add("short");cKeywords.add("signed");cKeywords.add("sizeof");
        cKeywords.add("static");
        cKeywords.add("struct");cKeywords.add("switch");cKeywords.add("typedef");
        cKeywords.add("union");
        cKeywords.add("unsigned");cKeywords.add("void");cKeywords.add("volatile");
        cKeywords.add("while");
        cKeywords.add("inline");cKeywords.add("restrict");

        predefinedTokens.put("Operators", cOperators);
        predefinedTokens.put("Punctuations", cPunctuation);
        predefinedTokens.put("Keywords", cKeywords);
    }
}
```

Tokenization

The `tokenize()` method reads each line of the C code, matches tokens using regular expressions (Explained in the earlier subsection), and adds them to the token list. It works by traversing on line by line in the file then start extracting tokens from each lexeme. It compares each token with pattern that was defined by regular expressions & accordingly divides it into tokens.

```
public void tokenize() {
    tk.clear();
    Token.idCount = 1;
    ArrayList<String> id_struct = new ArrayList<>();
    ArrayList<String> lines = reader.getClines();
    boolean inBlockComment = false;
    String num_regex = "\\d+(\\.\\d+)?((e|E)(['+']|-)?\\d+)?";
    String binary_octal_hex_regex = "(((0b|0B)[0-1]+)|(0[0-7]+)|((0x|0X)[0-9a-fA-F]+))";
    String bad_binary_octal_hex_regex = "(0b|0B)[0-9]+|0[0-9]+|(0x|0X)[0-9a-zA-Z]+";
    String num_dataType_regex = "(ULL|LL|L|UL|F|ull|ll|l|ul|f)?";
    String bad_Identifiers = "\\d+[a-zA-Z_#$@]+\\d*|[a-zA-Z_#$@]+[#$@]+[a-zA-Z_#$@\\d]+";
    Pattern tokenPattern = Pattern.compile(
            "\"(?:[^\"\\\\]|\\\\.)*\"|" +
            bad_binary_octal_hex_regex + "|" +
            binary_octal_hex_regex + "|" +
            "'.'|" +
            "\"|'|" +
            bad_Identifiers + "|" +
            num_regex + num_dataType_regex + "|" +
            "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b(\\()|" +
            "\\}(?:\\s*\\w+)?\\s*;|" +
            "^(struct|typedef( )struct)\\s+(\\w+)\\s*[\\{|;]|" +
            "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b((?!\\()|(?!\\{))|" +
            "\\(|\\)|\\{|\\}|;|,|\\[|\\]|" +
            "\\+\\+|--|==|!=|<=|>=|&&|" +
            "\\-\\>|\\+\\=|\\-\\=|\\*\\=|\\/\\=|\\%\\=|\\&\\=|\\|\\=|\\|\\||" +
            "\\+|\\*|/|%|\\^\\=|\\<\\<\\=|\\>\\>\\=|\\<\\<|\\>\\>|<|>|" +
            "\\-|\\+|\\*|\\=|\\&\\&|\\|\\||\\!|\\&|\\||\\^|\\~|\\?|" +
            "\\:|\\+\\+|\\-\\-|\\."
```

```
        );
        boolean isStruct = false;
        boolean structDataType;
        int lineNum = 1;
        for (String line : lines) {
            structDataType = false;
            // Skip lines starting with "#" and comments
            if (!line.trim().startsWith("#")) {
                // Match tokens using regular expression
                Matcher matcher = tokenPattern.matcher(line);
                while (matcher.find()) {
                    String token = matcher.group();
                    if (token.matches("\\}(?:\\s*\\w+)?\\s*;") && isStruct){
                        isStruct = false;
                        String filteredStr = token.replaceAll("\\s", "").replaceAll(";", "")
                                .replaceAll("}", "");
                        tk.add(new Token("Puncatuations", "}", lineNum));
                        tk.add(new Token("Puncatuations", ";", lineNum));
                        if (!filteredStr.isEmpty()){
                            id_struct.add(filteredStr);
                            tk.add(new Token("Identifiers (Struct)", filteredStr, lineNum));
                        }
                        continue;
                    }
                    if (isPunctuation(token)) {
                        tk.add(new Token("Puncatuations", token, lineNum));
                        // Disable Struct Data Type Flag
                        if (token.equals(";"))
                            structDataType = false;
                    } else if (isOperator(token)) {
                        tk.add(new Token("Operators", token, lineNum));
                    } else if (token.matches("(0b|0B)[0-9]+")
                               && !token.replaceAll("[2-9]+", "").equals(token)
                               || token.matches("0[0-9]+")
                               && !token.replaceAll("[8-9]+", "").equals(token)
                               || token.matches("(0x|0X)[0-9a-zA-Z]+")
                               && !token.replaceFirst("(0x|0X)", "")
                               .replaceAll("[g-zG-Z]+", "")
                               .equals(token.replaceFirst("(0x|0X)", ""))
                               ) {
                        tk.add(new Token("Bad Integers", token, lineNum));
                    } else if (token.matches(binary_octal_hex_regex + "|"
                               + num_regex + num_dataType_regex)
                              ) {
                        if (token.matches(num_regex + "(ul|UL|l|L)")) {
                            tk.add(new Token("Long", token, lineNum));
                        } else if (token.matches(num_regex + "(ull|ULL|ll|LL)")) {
                            tk.add(new Token("Long Long", token, lineNum));
                        }
                        else if (token.matches(binary_octal_hex_regex) ||
                                token.matches("(0|([1-9][0-9]*))")){
                            tk.add(new Token("Integers", token, lineNum));
                        } else{
                            tk.add(new Token("Floats", token, lineNum));
                        }
                    } else if (token.startsWith("'") && token.endsWith("'")) {
```

```
                    tk.add(new Token("Characters", token, lineNum));
                } else if (token.startsWith("\"") && token.endsWith("\"")) {
                    tk.add(new Token("Strings", token, lineNum));
                } else if (isKeyword(token)) {
                    if (token.equals("struct")) {
                        isStruct = true;
                    }
                    tk.add(new Token("Keywords", token, lineNum));
                } else {
                    if (id_struct.contains(token) && !structDataType){
                        id_struct.add(token);
                        tk.add(new Token("Identifiers (Struct)", token, lineNum));
                        structDataType = true;
                    }
                    // Check if the token is a function or a variable
                    else if (isFunction(token)) {
                        token = token.replaceFirst("\\(", "");
                        tk.add(new Token("Puncatuations", "(", lineNum));
                        if (isKeyword(token)) {
                            tk.add(new Token("Keywords", token, lineNum));
                        } else {
                            tk.add(new Token("Identifiers (Function)", token, lineNum));
                        }
                    } else if (isStruct(token)) {
                        isStruct = true;
                        token = token.replaceFirst("typedef", "");
                        tk.add(new Token("Keywords", "typedef", lineNum));

                        token = token.replaceFirst("struct", "");
                        tk.add(new Token("Keywords", "struct", lineNum));

                        token = token.replaceAll("\\s", "");
                        while(!token.equals(token.replaceFirst("\\{", ""))){
                            token = token.replaceFirst("\\{", "");
                            tk.add(new Token("Puncatuations", "{", lineNum));
                        }
                        while(!token.equals(token.replaceFirst(";", ""))){
                            token = token.replaceFirst(";", "");
                            tk.add(new Token("Puncatuations", ";", lineNum));
                        }
                        id_struct.add(token);
                        tk.add(new Token("Identifiers (Struct)", token, lineNum));

                    } else {
                        if (token.contains("}")) {
                            token = token.replaceFirst("}", "");
                            tk.add(new Token("Puncatuations", "}", lineNum));
                        }
                        if (token.contains(";")) {
                            token = token.replaceFirst(";", "");
                            tk.add(new Token("Puncatuations", ";", lineNum));
                        }
                        if (token.matches(bad_Identifiers)){
                            tk.add(new Token("Bad Identifiers", token, lineNum));
                        } else if (!token.isEmpty()) {
                            tk.add(new Token("Identifiers (Variable)", token, lineNum));
```

```
                        }
                    }
                }
            }
        }
        lineNum++;
    }
}
```

Token Categories

Here we implemented some methods to check for the following while reading the C file:

- Function
- Struct
- Variable
- Operator
- Punctuation
- Keyword

The logic works by comparing & checking with predefined tokens(dictionary) or by checking if it matches the pattern of regex

```
private boolean isFunction(String token) {
        return token.contains("(");
    }

    private boolean isStruct(String token) {
        return !token.replaceFirst("struct", "").equals(token);
    }

    private boolean isVariable(String token) {
        // Check if the token is followed by an equal sign or is a standalone identifier
        return token.matches("[a-zA-Z_][a-zA-Z0-9_]*\\s*(=\\s*.*|;)");
    }

    private boolean isOperator(String token) {
        return predefinedTokens.get("Operators").contains(token);
    }

    private boolean isPunctuation(String token) {
        return predefinedTokens.get("Punctuations").contains(token);
    }

    private boolean isKeyword(String token) {
        return predefinedTokens.get("Keywords").contains(token);
    }
```

3.3.3 Token class

Brief Overview

The `Token` class represents individual tokens identified during the tokenization process performed by the `Lexer` class. It encapsulates information such as the name of the token, its attribute value, and the line number where it appears in the source code. Thus it is used in constructing **symbol table**.

Constructor

The constructor of the `Token` class initializes its attributes based on the parameters provided: `name`, `attrVal`, and `lineNum`.

```
public class Token {
    String name;
    String attrVal;
    Integer lineNum;

    // Static variables for identifier management
    static Integer idCount = 1;
    Integer entryNum = 0;
    static Integer numObj = 0;
    String identifier = "";
    String identifierType = "";

    public Token(String name, String attrVal, Integer lineNum) {
        this.name = name;
        this.attrVal = attrVal;
        this.lineNum = lineNum;
        if (name.contains("Identifiers") && !name.contains("Bad")){
            this.identifier = "id" + idCount;
            this.entryNum = idCount;
            idCount++;
            identifierType = name.replaceFirst("Identifiers \\(", "")
                    .replaceFirst("\\)", "");
    }
}
```

Identifier Management

The `Token` class manages identifiers by assigning unique identifier names (`identifier`) and entry numbers (`entryNum`) to each identifier token encountered during tokenization. Then after assigning them we will use it in the symbol table

3.4 GUI Application

The "C Lexer & Parser" is a software application with a graphical user interface designed to analyze C language source code. It is a tool for analyzing C code, offering features to tokenize, create symbol tables, check syntax, and visualize code structure with parse trees.

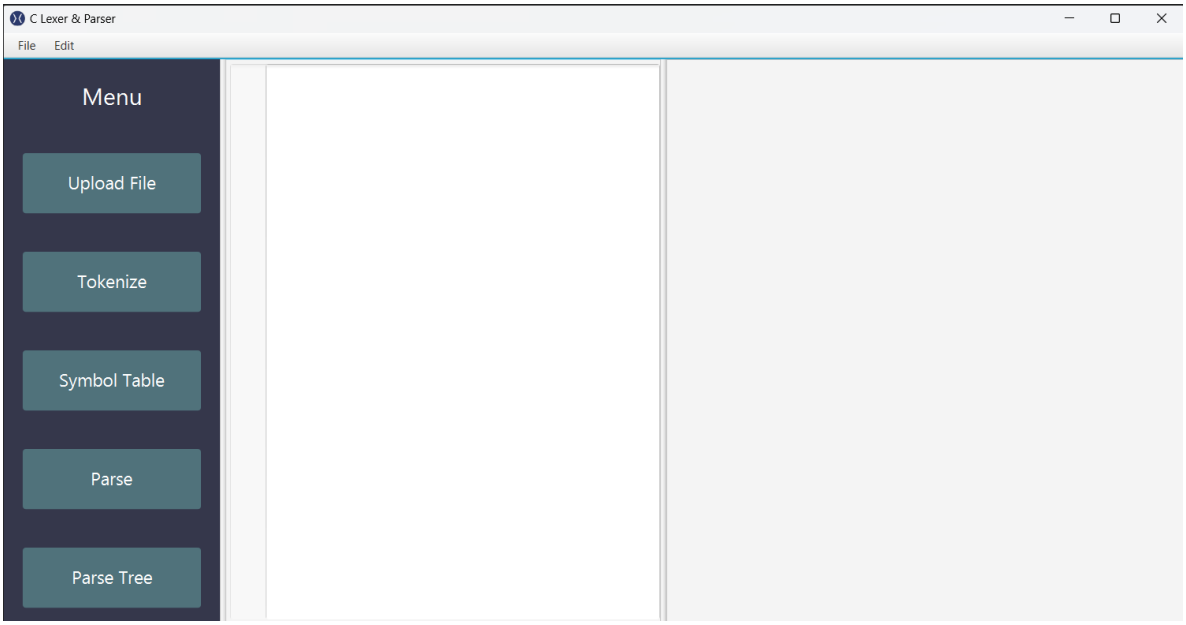


Figure 12: Main Window

1. Upload File This button is likely for uploading C source code files that you want to process with this tool.

2. Tokenize Clicking this button would typically trigger the lexer part of the tool to convert the source code into a series of tokens. Tokenization is the process of breaking up the input text into a sequence of tokens, which are the meaningful elements of the language like keywords, identifiers, symbols, etc.

3. Symbol Table This might display a table containing all the symbols (identifiers, constants, functions, etc.) used in the C source code with additional information such as their types, scope, and perhaps memory addresses.

4. Parse This would initiate the parsing process, where the sequence of tokens generated by the lexer is analyzed to make sure it follows the grammatical structure of the language. Parsing checks the syntax and builds a representation of the code that can be used for further processing, like constructing a parse tree or an abstract syntax tree (AST).

5. Parse Tree After parsing, this button might display the parse tree (or syntax tree), which is a tree representation of the syntactic structure of the source code based on the grammar of the language. This tree illustrates the hierarchy of language constructs and their relationship to each other within the code.

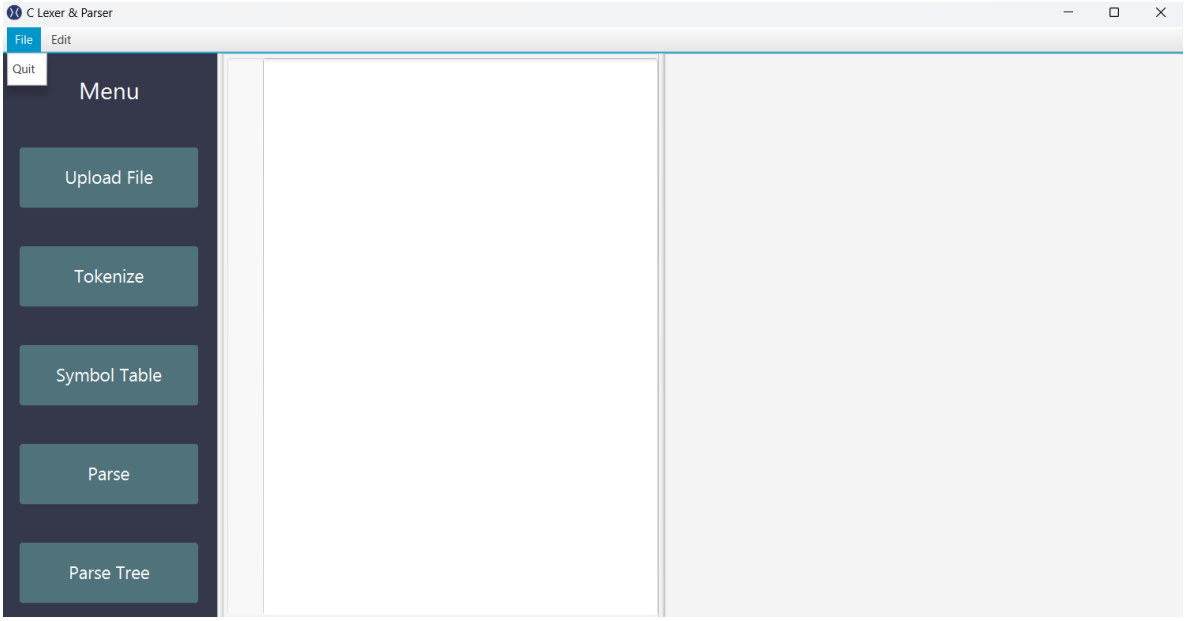


Figure 13: File Button

White Area The large white area to the right seems to be the main workspace where the output of each action (token list, symbol table, parse tree) might be displayed after the user performs an operation using the buttons in the menu. The top-left menu options labeled "File" and "Edit" are typical of application GUIs and would likely contain further options to save, load, or modify files, among other functions.

File Button In the image, the "File" menu in the upper left corner is shown with one visible option: "Quit." This option is standard in many software applications and is used to close the application. When you select "Quit," it typically prompts the application to end its processes and close the window, effectively exiting the program. Depending on the application's design, you might be prompted to save any unsaved work before the application closes to prevent loss of data.

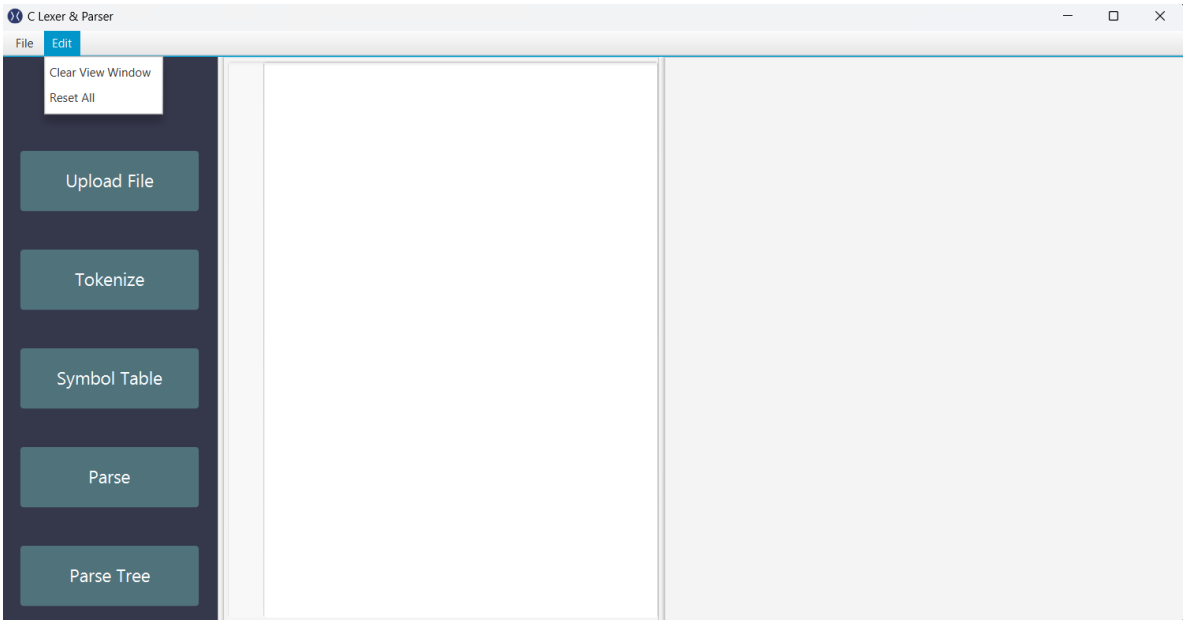


Figure 14: Edit Button

Edit Button In the upper left corner of the GUI, there's a menu labeled "Edit." This menu is typically used to house functions that allow the user to modify or manipulate the current document or files they are working on. In many applications, an "Edit" menu might include options like undo/redo, cut, copy, paste, find, and replace. However, in the context of this specific "C Lexer & Parser" application, the "Edit" menu contents are not visible, so we can only speculate that it would contain functions relevant to editing the code or data that the user is working with. It may include options to modify the input C code or adjust how the lexing and parsing operations are performed.

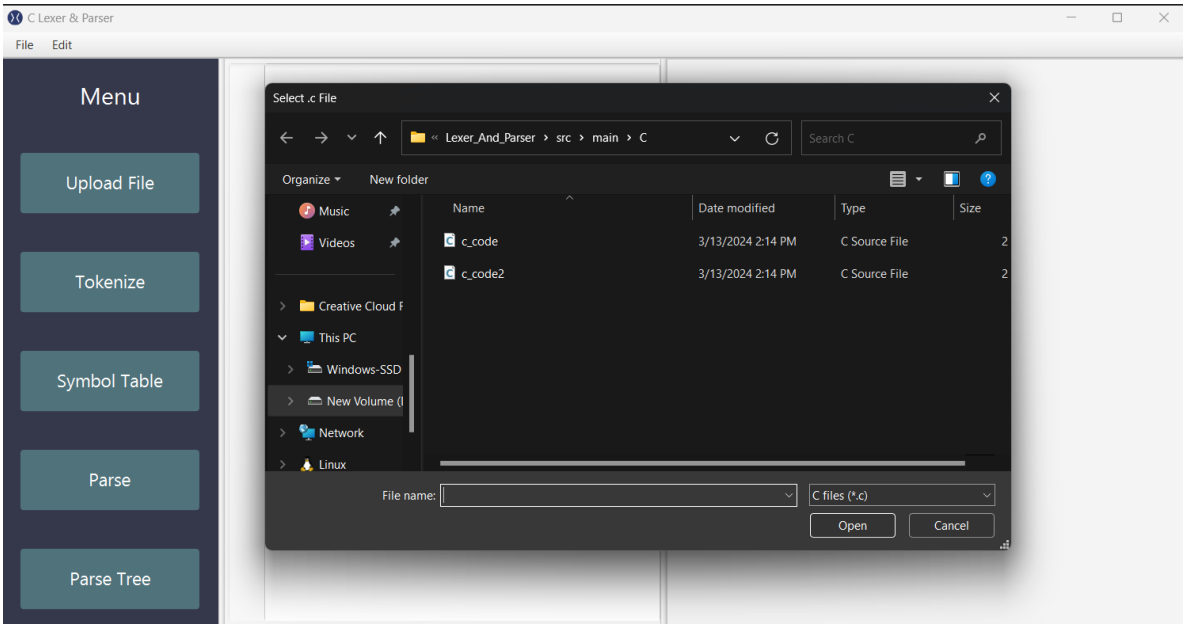


Figure 15: Upload File

The GUI shown in the image is a file dialog box, which is part of the "C Lexer & Parser" software application. This dialog box has been opened by clicking the "Upload File" button in the application's main window.

- **Dialog Box Title**: "Select .c File" indicates the purpose of the dialog is to select a C language source file.
- **Breadcrumb Navigation**: At the top, there's a path (Lexer_And_Parser ¿ src ¿ main ¿ C), showing the current directory, which allows the user to navigate the file system hierarchy.
- **Back and Forward Buttons**: These can be used to navigate to previously visited directories.
- **Main Area**: Displays files and folders from the current directory. In this case, two C source files named "c_code" and "c_code2" are visible, along with details such as modification date, type, and size.
- **Search Bar**: It allows the user to search for files within the current directory.
- **File Name Field**: Here, the user can type the name of a file they wish to open.
- **File Type Dropdown**: It allows the user to filter the files shown in the dialog box. It's currently set to "C files (*.c)" which will only show files with the .c extension.
- **Open and Cancel Buttons**: "Open" would confirm the selection and presumably load the file into the "C Lexer & Parser" application for processing. "Cancel" would close the dialog box without making a selection.

This dialog box is a common component in GUI applications, facilitating file opening and uploading operations.

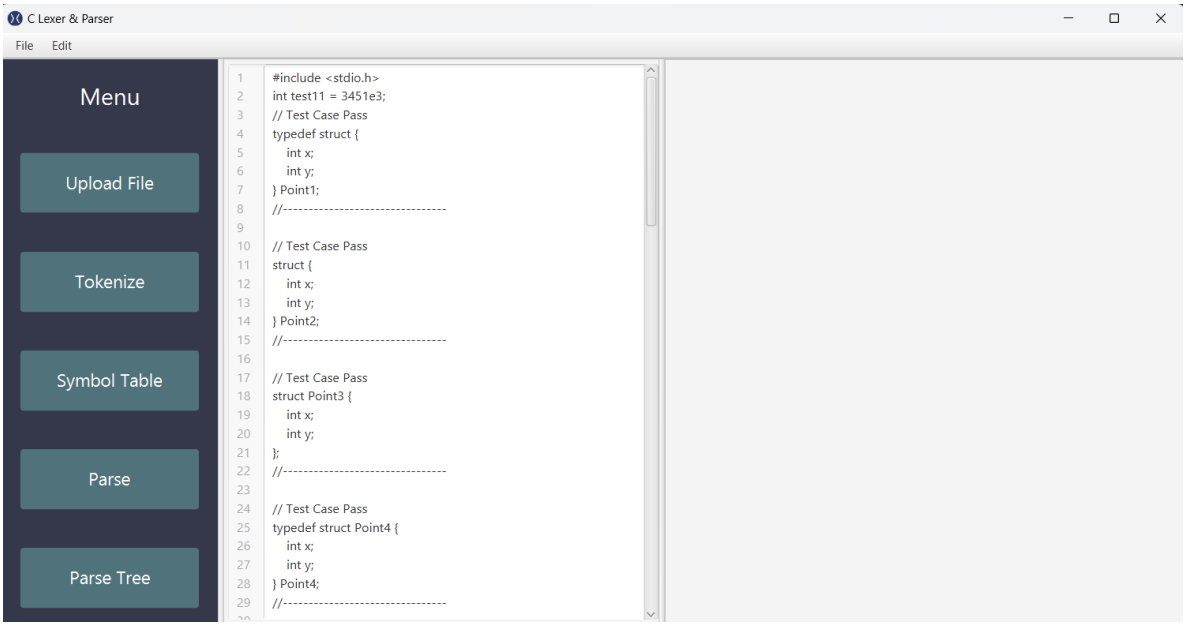


Figure 16: C File Code View

The GUI shown in the image is a menu bar from the software application "C Lexer Parser". The menu bar provides options for interacting with the application and potentially managing C source code files.

- **File Menu:** This menu likely offers functionalities related to file management, such as opening existing C source files, saving the currently edited file, or creating new files.
- **Edit Menu:** This menu might provide options for editing the C code within the application, such as copy, paste, undo, and redo functionalities.
- **Menu:** The presence of a menu named simply "Menu" suggests it might be a placeholder or a work-in-progress feature. Its functionalities are unclear from this screenshot.
- **Tokenize Menu:** This option likely initiates the process of tokenizing the loaded C source code file. Tokenization is the process of breaking the code down into smaller meaningful units (tokens) for further analysis by the application.
- **Parse Menu:** This option likely triggers the parsing stage, where the tokenized code is analyzed according to the grammar rules of the C language. This allows the application to understand the structure and meaning of the code.
- **Parse Tree Menu:** This option might be used to display a visual representation of the parsed code structure, often referred to as a parse tree. This can be helpful for debugging and understanding the code's syntax.
- **Symbol Table Menu:** This option likely provides access to the symbol table, a data structure that stores information about variables and other symbols used within the C code.

The "C Lexer Parser" software seems to be designed for lexical and syntactical analysis of C code. The menu bar offers functionalities for loading, manipulating, and analyzing C source code files.

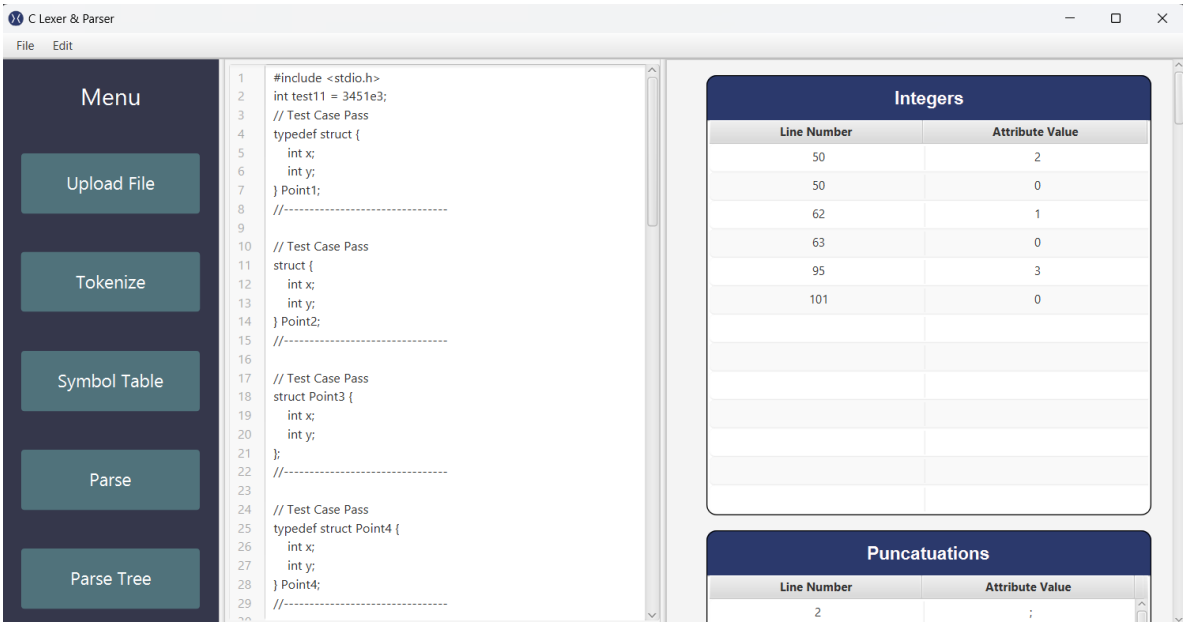
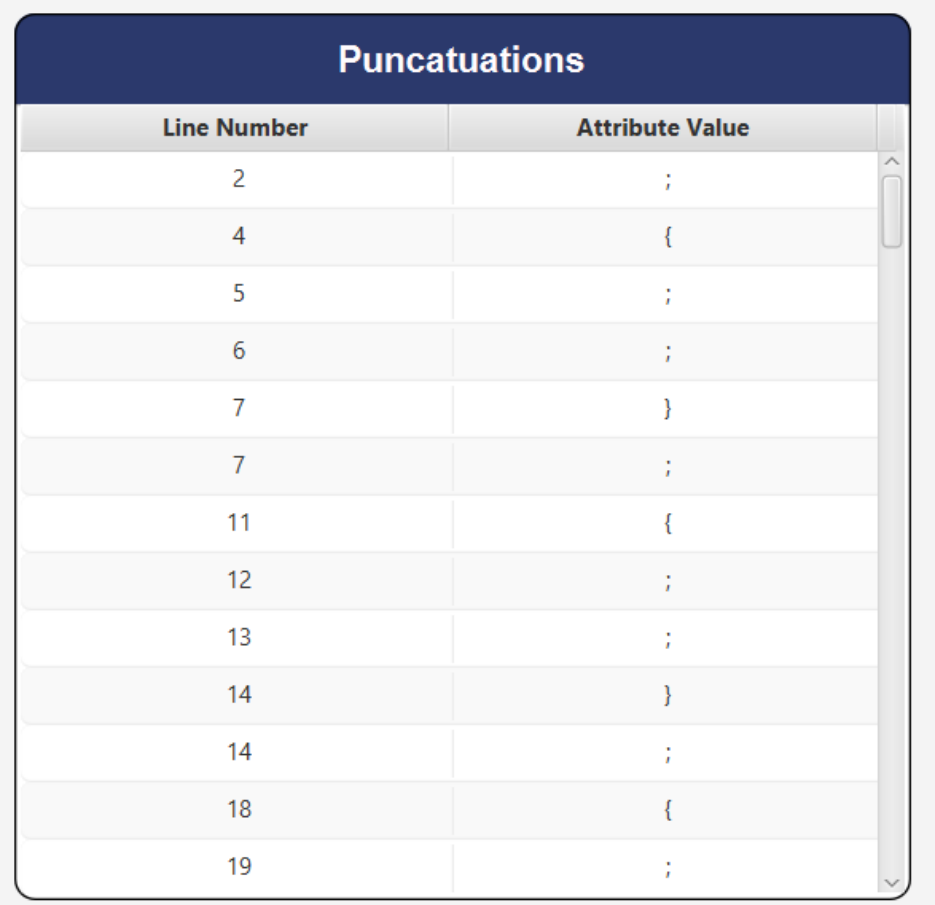


Figure 17: Tokenize Button

The screenshot displays a tokenization results window within the "C Lexer & Parser" software application. This window presents the results of tokenizing a C source code file, providing a structured view for analysis.

- **Table Header:** The row at the top likely indicates the column names within the tokenization results table. Typical headers could be "Token Type", "Token Value", "Line Number", and potentially others.
- **Tokenized Data:** The subsequent rows contain the actual tokenized data. Each row likely represents a single token identified within the C code during the tokenization process.
- **Token Types:** This column likely displays the type of token. Examples would include keywords (like 'int', 'if', 'while'), identifiers (variable names), operators, and punctuation symbols.
- **Token Values:** This column displays the actual value of the token within the source code. For example, a variable name would be listed here.
- **Line Numbers:** This column might indicate the line number in the original source code where the corresponding token was found.
- **Navigation/Search (Possibly):** The window might include navigation buttons or a search bar to allow users to quickly find specific tokens within the table.

The "C Lexer & Parser" software appears to feature a dedicated window for inspecting the results of tokenization. This table-based format provides a structured representation of the C code, likely used for further analysis or debugging within the application.



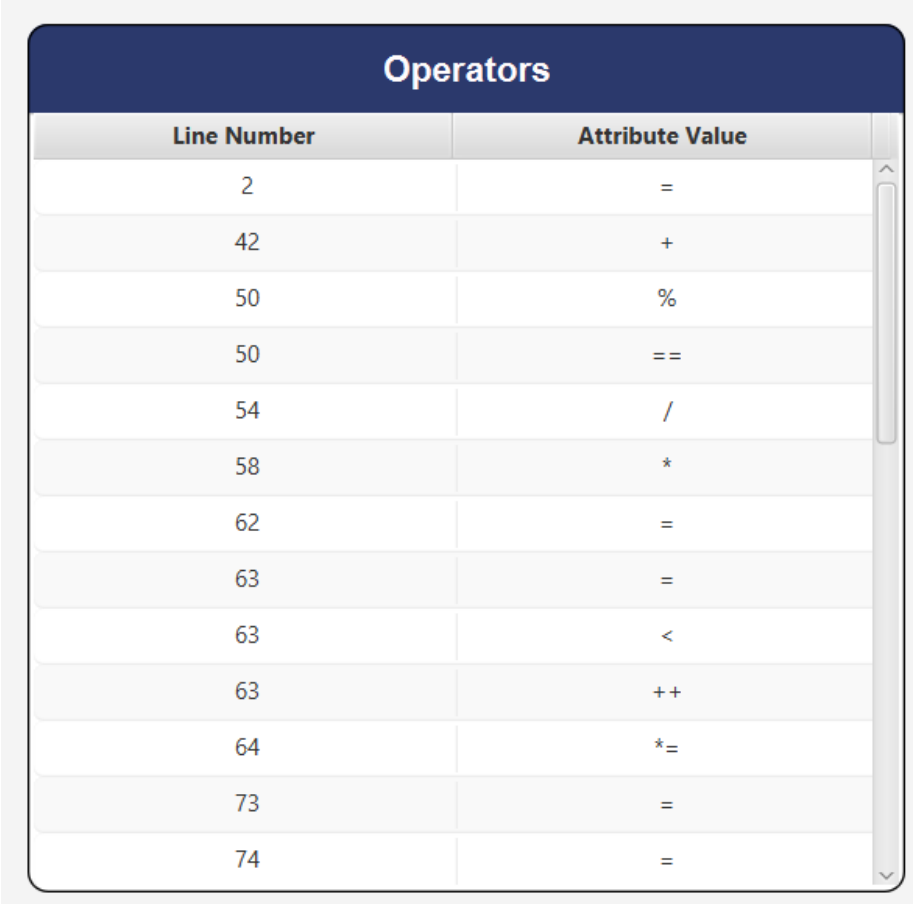
Puncatuations

Line Number	Attribute Value
2	;
4	{
5	;
6	;
7	}
7	;
11	{
12	;
13	;
14	}
14	;
18	{
19	;

Figure 18: Punctuations Tokens Table

Punctuation marks are essential for structuring code and separating different elements within the program's syntax. The lexer identifies punctuation tokens to aid in understanding the structural aspects of the code.

- **Table Headers**: "Punctuation Symbol", "Line Number", (potentially) "Column Number"
- **Punctuation Symbols**: This column lists the various punctuation symbols found in the code, such as periods (.), commas (,), semicolons (;), colons (:), braces (), etc.
- **Line Numbers**: Indicates the line where each punctuation symbol appears in the source code.
- **Column Numbers (Optional)**: Provides the column position of the punctuation symbol within its line.



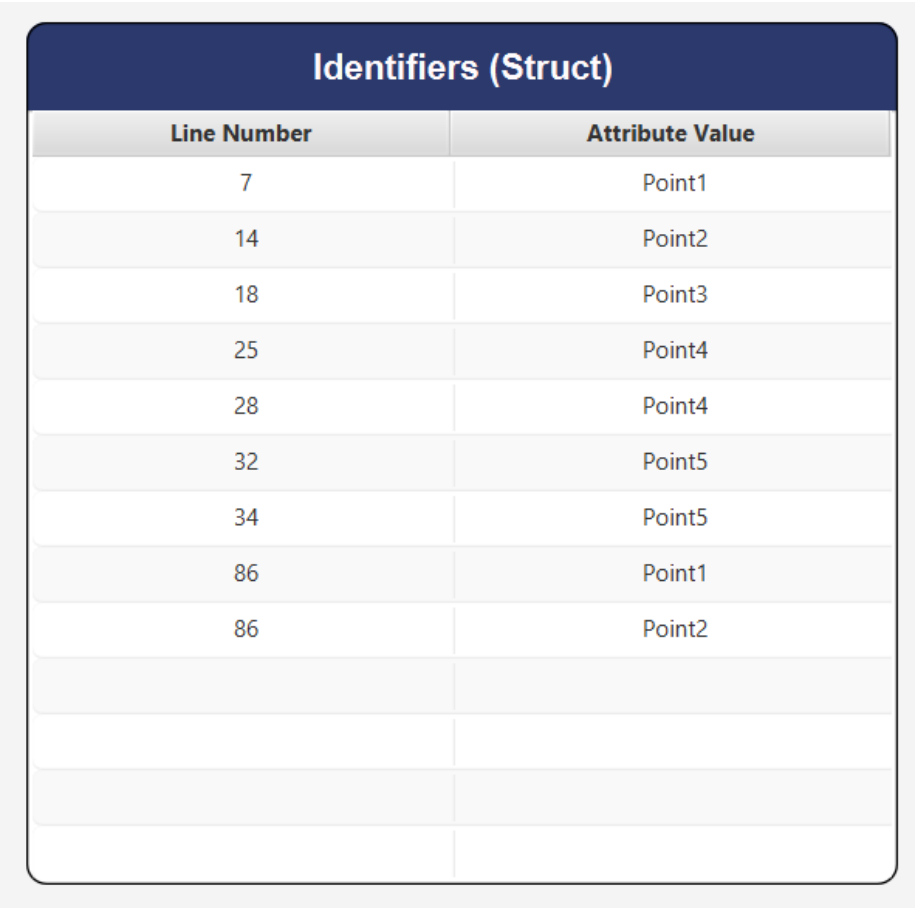
Operators

Line Number	Attribute Value
2	=
42	+
50	%
50	==
54	/
58	*
62	=
63	=
63	<
63	++
64	*=
73	=
74	=

Figure 19: Operators Tokens Table

Operators are crucial for performing calculations, comparisons, logical operations, and assignments within the code. Analyzing operator tokens is vital for understanding the code's behavior.

- **Table Headers**: "Token Type", "Token Value", "Line Number", (potentially) "Column Number"
- **Token Types**: Classifications like arithmetic, comparison, logical, assignment, etc.
- **Token Values**: The actual operator symbols (+, -, ¡, , etc.).
- **Line and Column Numbers**: Location of the operator within the source code.



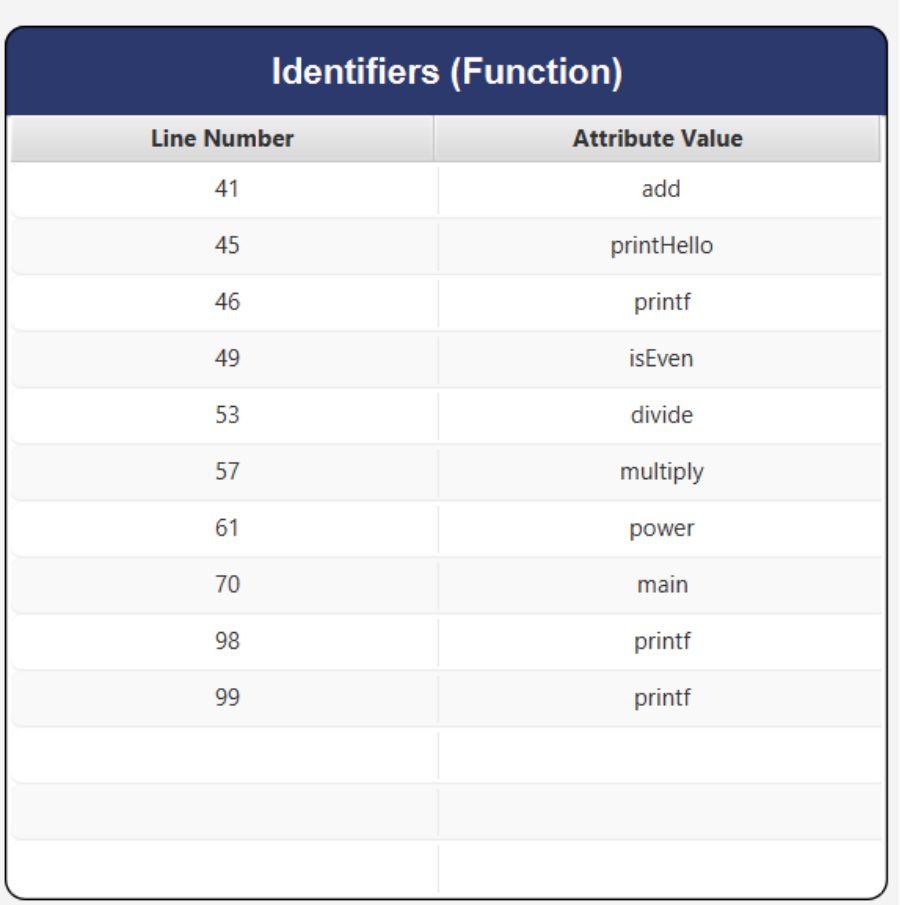
Identifiers (Struct)

Line Number	Attribute Value
7	Point1
14	Point2
18	Point3
25	Point4
28	Point4
32	Point5
34	Point5
86	Point1
86	Point2

Figure 20: Identifiers (Struct) Tokens Table

The Identifiers (Struct) Tokens Table lists the structure name identifiers detected by the lexer within the source code. Structures are user-defined data types that group variables of different types under a single name. This table likely provides details about what structures are defined, where they appear in the code, and the members contained within these structures.

- **Table Headers**: "Struct Name", "Line Number", (potentially) "Column Number", (potentially) "Member Variables"
- **Struct Names**: The names used to define structures in the code.
- **Line and Column Numbers**: The location of the struct definition.
- **Member Variables (Optional)**: A list of variable names and types declared within the struct.



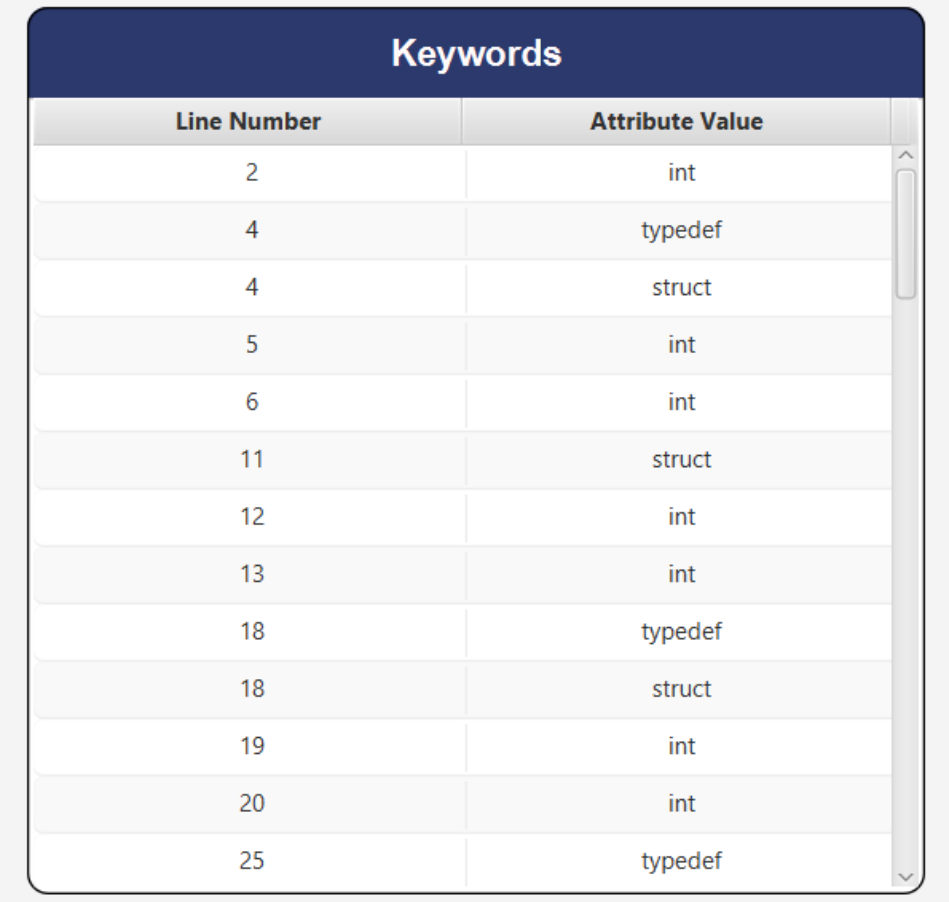
Identifiers (Function)

Line Number	Attribute Value
41	add
45	printHello
46	printf
49	isEven
53	divide
57	multiply
61	power
70	main
98	printf
99	printf

Figure 21: Identifiers (Function) Tokens Table

The Identifiers (Function) Tokens Table lists function names detected within the source code. Functions are blocks of code designed to perform specific tasks. This table provides information about the defined functions, helping understand where they are declared and used in the code.

- **Table Headers**: "Function Name", "Line Number", (potentially) "Column Number", (potentially) "Parameter Types"
- **Function Names**: The names used to define and call functions.
- **Line and Column Numbers**: The location of the function definition.
- **Parameter Types (Optional)**: A list of data types for the arguments the function expects.



Keywords

Line Number	Attribute Value
2	int
4	typedef
4	struct
5	int
6	int
11	struct
12	int
13	int
18	typedef
18	struct
19	int
20	int
25	typedef

Figure 22: Keywords Tokens Table

The Keywords Tokens Table lists keywords recognized by the programming language. Keywords have specific meanings and cannot be used as variable names. This table helps identify the core language constructs used in the code.

- **Table Headers**: "Keyword", "Line Number", (potentially) "Column Number"
- **Keywords**: Reserved words like "if", "else", "for", "while", "int", "float", etc.
- **Line and Column Numbers**: Where the keyword appears in the code.

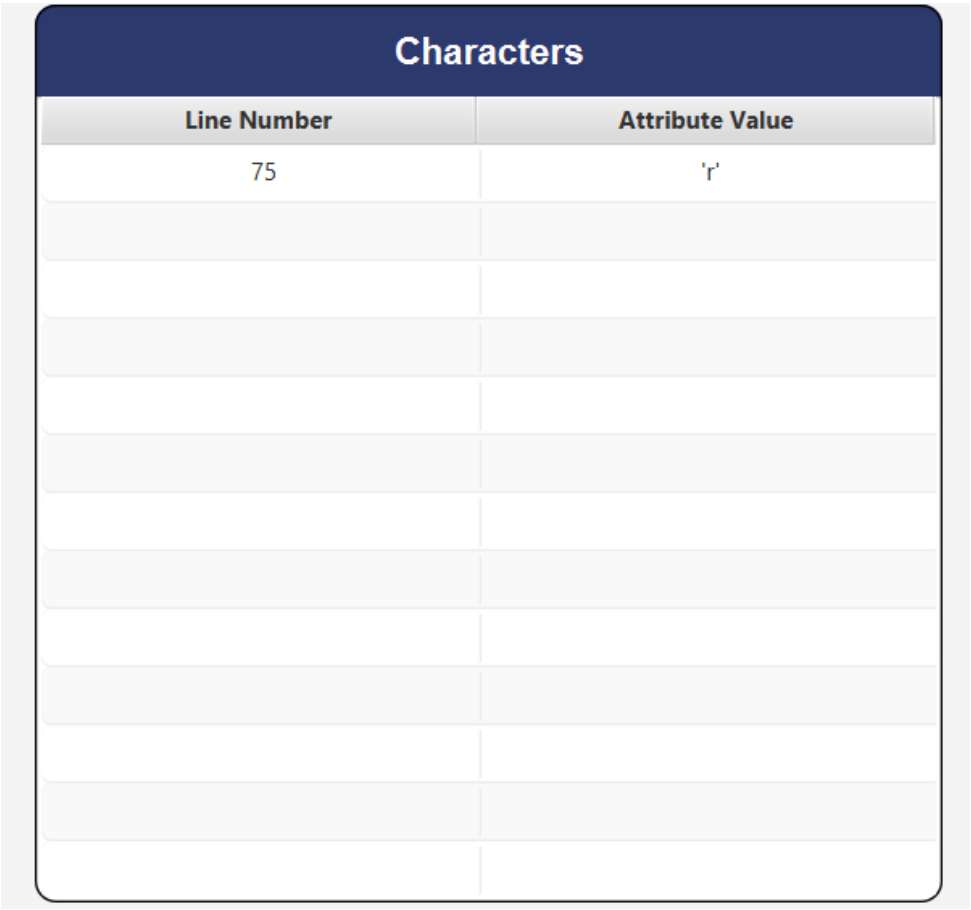
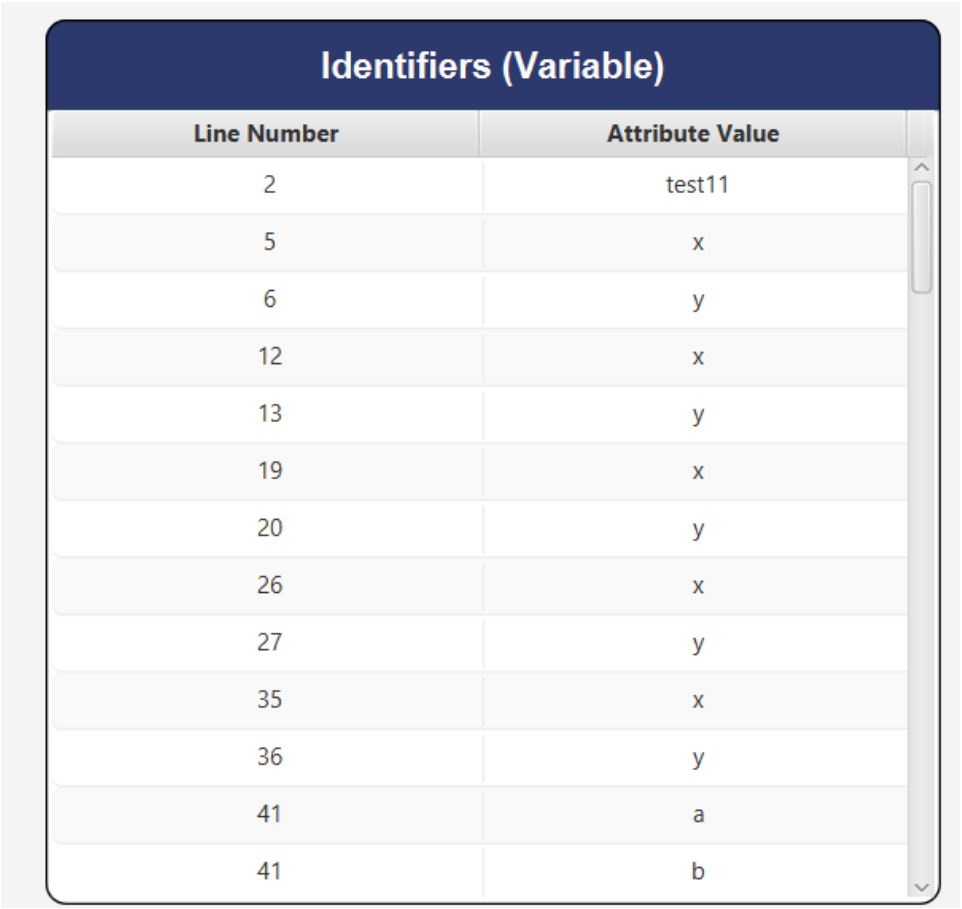


Figure 23: Characters Tokens Table

The Characters Tokens Table lists individual characters detected in the code. Characters are enclosed within single quote marks and typically represent single elements of textual data. This table aids in understanding strings and other character manipulations used within the code.

- **Table Headers**: "Character", "Line Number", (potentially) "Column Number"
- **Character**: Individual letters, numbers, symbols, or special characters found within single quotes.
- **Line and Column Numbers**: Location of the character within the source code.



Identifiers (Variable)

Line Number	Attribute Value
2	test11
5	x
6	y
12	x
13	y
19	x
20	y
26	x
27	y
35	x
36	y
41	a
41	b

Figure 24: Identifiers (Variables) Tokens Table

The Identifiers (Variables) Tokens Table lists variable names used within the source code. Variables are used to store and manipulate data during program execution. This table likely identifies the declared variables, their data types (optional), and where they appear within the code.

- **Table Headers**: "Variable Name", "Line Number", (potentially) "Column Number", (potentially) "Data Type"
- **Variable Names**: The names used to reference data.
- **Line and Column Numbers**: The location in the code where the variable is declared or used.
- **Data Type (Optional)**: Indicates whether the variable stores integers, floating-point numbers, characters, etc.

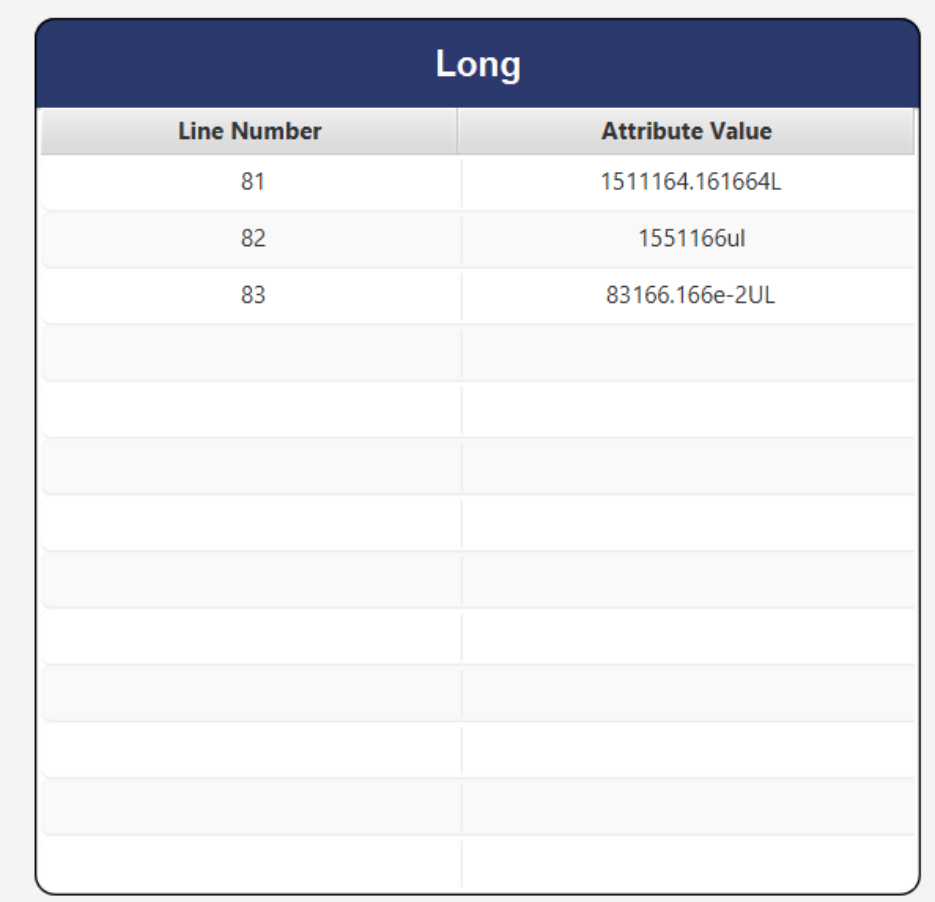
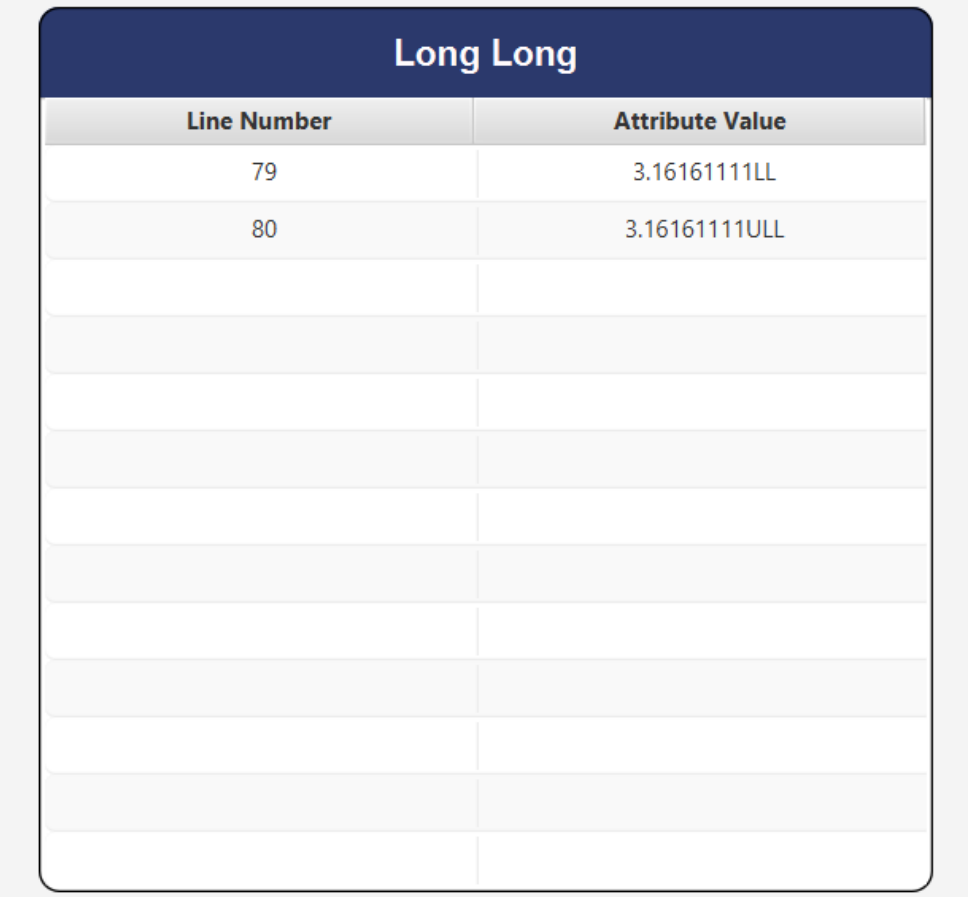


Figure 25: Long Tokens Table

The Long Tokens Table lists 'long' integer values found in the code. Long integers provide a larger range of whole numbers compared to regular integers, especially in older systems. This table helps identify where these larger numerical values are used within the code.

- **Table Headers**: "Long Value", "Line Number", (potentially) "Column Number"
- **Long Values**: Whole numbers with the 'long' specifier (e.g., 123456789L).
- **Line and Column Numbers**: Location of the long value in the code.



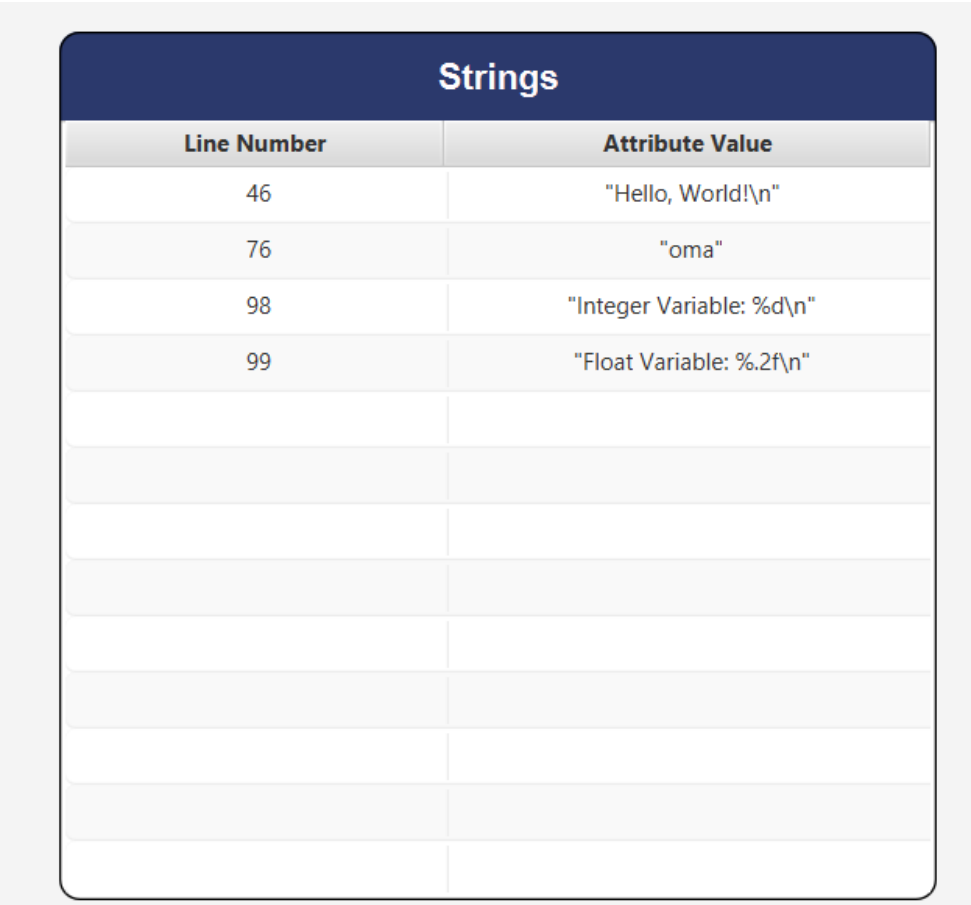
Long Long

Line Number	Attribute Value
79	3.16161111LL
80	3.16161111ULL

Figure 26: Long Long Tokens Table

The Long Long Tokens Table lists 'long long' integer values in the code. Long long integers offer an even wider range of whole numbers and are common in modern systems. This table helps identify where these very large numeric values are used for calculations or data storage.

- **Table Headers**: "Long Long Value", "Line Number", (potentially) "Column Number"
- **Long Long Values**: Whole numbers with the 'long long' specifier (e.g., 123456789012345LL).
- **Line and Column Numbers**: Location of the long long value within the code.



Strings

Line Number	Attribute Value
46	"Hello, World!\n"
76	"oma"
98	"Integer Variable: %d\n"
99	"Float Variable: %.2f\n"

Figure 27: Strings Tokens Table

String tokens represent textual data within the code. Understanding strings is essential for interpreting output messages, user prompts, or textual elements within the program.

- **Table Headers**: "String Value", "Line Number", (potentially) "Column Number"
- **String Values**: The actual text content enclosed within quotation marks in the source code.
- **Line and Column Numbers**: The location of the string within the source code.

Floats

Line Number	Attribute Value
2	3451e3
73	42.3
74	3451e3
77	3.14
93	4.5578441e-7
94	3.146416e-7

Figure 28: Floats Tokens Table

Floating-point tokens represent real numbers, allowing the code to handle more precise calculations compared to integer values alone.

- **Table Headers**: "Floating-Point Value", "Line Number", (potentially) "Column Number"
- **Floating-Point Values**: Decimal numbers present in the code (e.g., 3.14159, 12.5, -0.003).
- **Line and Column Numbers**: The location of the floating-point value within the source code.

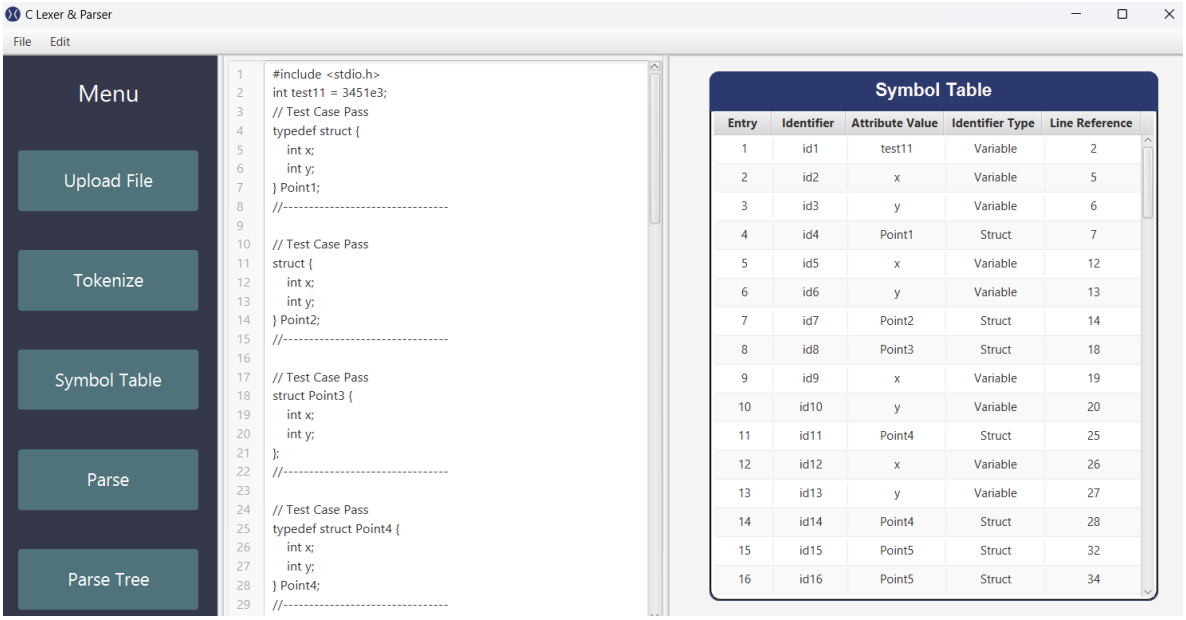
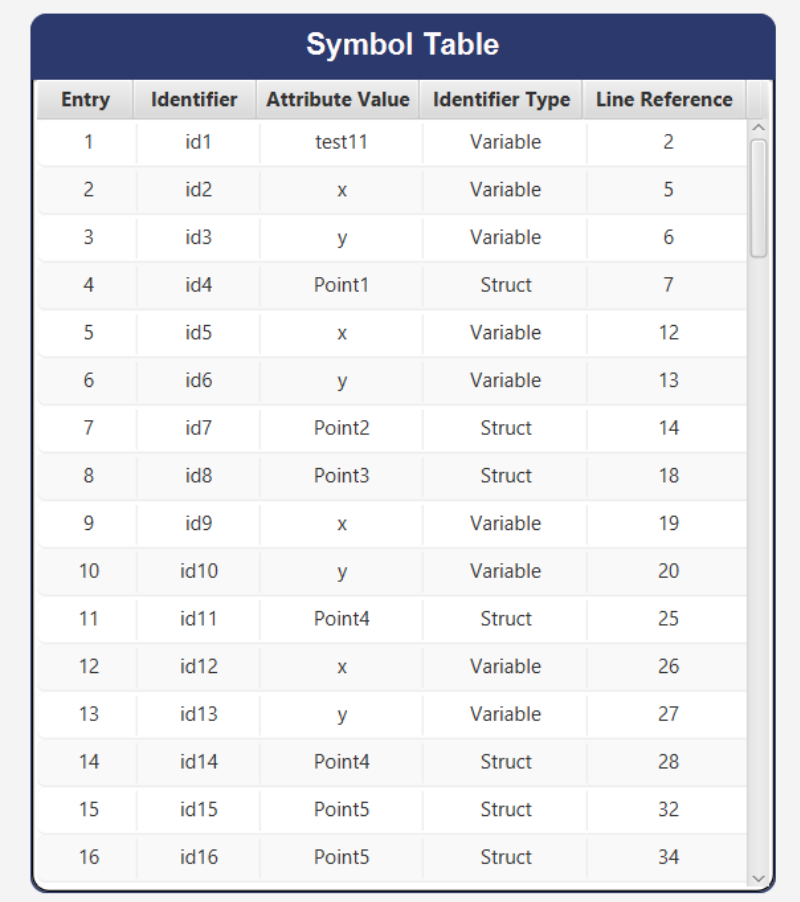


Figure 29: Symbol Table Button

The screenshot displays the symbol table window within the "C Lexer & Parser" software application. This window presents information about variables and other symbols identified during the lexical analysis stage from a C source code file.

- **Table Headers**: The top row shows the column names, which typically include "Entry", "Identifier", "Attribute Value", "Identifier Type", and "Line Reference".
- **Entry**: An identifier or sequential number assigned to each symbol within the symbol table.
- **Identifier**: The name of the variable, function, struct, or other symbol found in the code.
- **Attribute Value (Optional)**: This column might contain additional details specific to the symbol type.
- **Identifier Type**: The category or classification of the symbol, such as "Variable", "Struct", or "Function".
- **Line Reference**: The line number in the source code where the symbol was first encountered.
- **Symbol Table Entries**: The subsequent rows list the symbols identified by the lexer, along with their corresponding details.
- **Variable Information**: For variables, the "Attribute Value" column might show the variable's data type (e.g., "int", "float").
- **Struct Information**: For structs, the "Attribute Value" column might contain details about member variables within the struct.

The symbol table serves as a reference for the compiler or interpreter during subsequent stages of code processing. It provides a structured view of the symbols used within the code, aiding in memory allocation, variable lookup, and overall program analysis.



Symbol Table

Entry	Identifier	Attribute Value	Identifier Type	Line Reference
1	id1	test11	Variable	2
2	id2	x	Variable	5
3	id3	y	Variable	6
4	id4	Point1	Struct	7
5	id5	x	Variable	12
6	id6	y	Variable	13
7	id7	Point2	Struct	14
8	id8	Point3	Struct	18
9	id9	x	Variable	19
10	id10	y	Variable	20
11	id11	Point4	Struct	25
12	id12	x	Variable	26
13	id13	y	Variable	27
14	id14	Point4	Struct	28
15	id15	Point5	Struct	32
16	id16	Point5	Struct	34

Figure 30: Symbol Table

The screenshot showcases the symbol table window of the "C Lexer & Parser" application. The symbol table acts as a central repository of information about variables, functions, and other symbolic elements defined within the analyzed C source code.

- **Symbol Table Structure**: The table is organized into columns, providing specific details about each symbol:
- **Entry:** A unique identifier or index for referencing symbols.
- **Identifier:** The actual name used in the code to represent a variable, function, or other entity.
- **Attribute Value:** Additional information tied to the symbol, potentially including its data type, structure members, or other properties.
- **Identifier Type:** Classifies the symbol, indicating whether it's a variable, function, structure, etc.
- **Line Reference:** Provides the line number in the original code where the symbol was first declared or encountered.

Purpose of the Symbol Table: During compilation or interpretation, the symbol table is consulted to resolve references to variables, check for correct usage, and allocate memory. This structured representation of symbols ensures consistent interpretation of the code.

4 SYNTAX ANALYSIS

4.1 Introduction

Syntax Analysis is a second phase of the compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal grammar. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

Syntax Analysis in Compiler Design process comes after the Lexical analysis phase. It is also known as the Parse Tree or Syntax Tree. The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyser also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program. Otherwise, it will display error messages.

4.2 Parsing Techniques

Parsing techniques are divided into two different groups:

Top-Down Parsing, Bottom-Up Parsing Top-Down Parsing In the top-down parsing construction of the parse tree starts at the root and then proceeds towards the leaves.

Two types of Top-down parsing are:

Predictive Parsing: Predictive parse can predict which production should be used to replace the specific input string. The predictive parser uses look-ahead point, which points towards next input symbols. Backtracking is not an issue with this parsing technique. It is known as LL(1) Parser

Recursive Descent Parsing: This parsing technique recursively parses the input to make a prase tree. It consists of several small functions, one for each nonterminal in the grammar.

We used Top Down Parsing

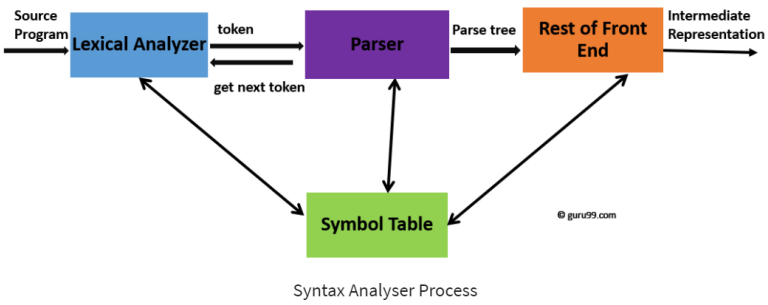


Figure 31: Syntax Analysis process

4.3 Context free Grammar CFG

Context Free Grammar (CFG) is a type of formal grammar that is used to generate every conceivable string in a given formal language. It's like a recipe that tells you how to create different strings in a language.
A context free grammar G is defined by four tuples:

G = (V, T, P, S)

Here's what they mean:

1. G refers to a grammar that is made up of sets of different production rules. These rules are used to generate the strings of a language.
2. T represents the final set of terminal symbols, which are usually denoted by lower case letters.
3. V represents the final set of nonterminal symbols, which are usually denoted by capital letters.
4. P is a set of production rules. These rules are used to replace the nonterminal symbols (on the left side of a production) with other terminals (on the right side of a production).
5. S is the start symbol. This symbol is used to derive the string.

The string in a CFG is derived using the start symbol. This string can be derived by continuously replacing a nonterminal with the right-hand side of a production, until all the nonterminals are replaced by terminal symbols.

4.3.1 Grammar

A grammar is a set of structural rules which describe a language. Grammars assign structure to any sentence.left side of a production consists of non-terminal.Right Side of a production is a combination between terminals and non-terminals.

4.3.2 Left Recursion

A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with
Why we eliminated left recursion from grammar rules? Left recursion is a common problem that

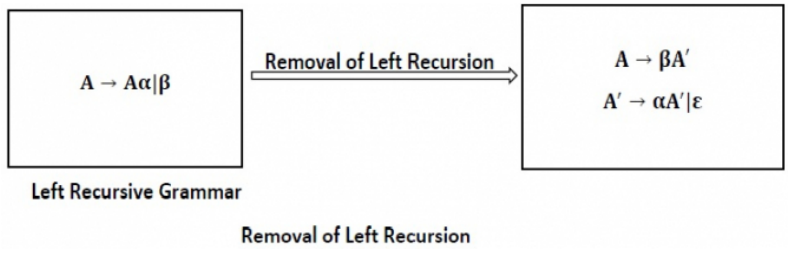


Figure 32: Left Recursion

occurs in grammar during parsing in the syntax analysis part of compilation. It is important to remove left recursion from grammar because it can create an infinite loop, leading to errors and a significant decrease in performance.

4.3.3 Left Factoring

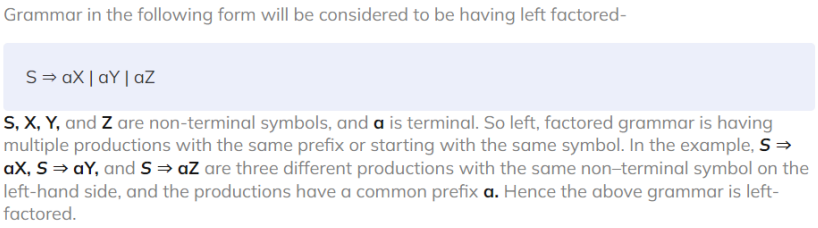


Figure 33: Left Factoring

Problems with Left Factored Grammar: Left-factored grammar creates an ambiguous situation for top-down parsers. Top-down parsers cannot choose the production to derive the given string since multiple productions will have a common prefix. This creates an ambiguous situation for the top-down parser.

To tackle this situation, we need to convert the left-factored grammar into an equivalent grammar without any productions having a common prefix. This is done using left factoring in Compiler design.

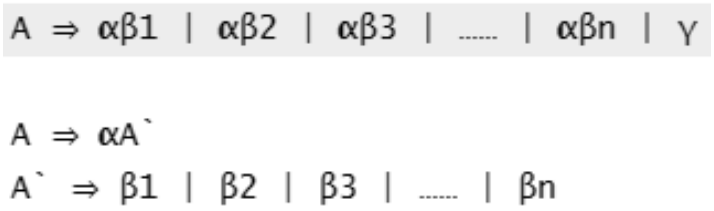


Figure 34: Elimination Of Left Factor

4.3.4 Grammar Rules

```
translation-unit → statement-list
translation-unitFactor → translation-unit' | epsilon
translation-unit' → external-declaration translation-unit' | epsilon
external-declaration → external-declarationFactor
external-declarationFactor → init-declarator-list
constant_opt → const | epsilon

function-definition → Identifiers_Function ( parameter-list ) function-prototype
function-prototype → compound-statement | ;
parameter-list → type-specifier Identifiers_Variable assignment-expression_opt parameter-list'
    | epsilon
assignment-expression_opt → assignment-expression | epsilon
parameter-list' → , parameter-listFact | epsilon
parameter-listFact → type-specifier Identifiers_Variable | . . .

function-call → Identifiers_Function ( argument-expression-list )

declaration-specifiers_opt → declaration-specifiers | epsilon
declaration → ( declaration_opt ) ;
declaration_opt → init-declarator-list | epsilon
declaration-list → declaration declaration-list'
declaration-list' → epsilon | declaration declaration-list'
```

```
declaration-specifiers → storage-class-specifier declaration-specifiers
    | type-specifier declaration-specifiers
    | type-qualifier declaration-specifiers
    | epsilon
storage-class-specifier → auto | register | static | extern | typedef
type-specifier → void | char pointerLoop
    | short pointerLoop | int pointerLoop
    | long long_opt pointerLoop
    | float pointerLoop | double pointerLoop
    | signed | unsigned
    | struct-or-union-specifier | enum-specifier
long_opt → epsilon | long

specifier-qualifier-list → specifier-qualifier-listFactor specifier-qualifier-list | epsilon
specifier-qualifier-listFactor → type-specifier | type-qualifier
type-name → specifier-qualifier-list type-name_opt
type-name_opt → abstract-declarator | epsilon

abstract-declarator → pointer | pointer_opt abstract-declaratorFactor
pointerLoop → epsilon | * pointerLoop
pointer_opt → * | epsilon
abstract-declaratorFactor → direct-abstract-declarator | epsilon
direct-abstract-declarator → ( abstract-declarator ) direct-abstract-declarator'
    | direct-abstract-declarator''
direct-abstract-declarator' → [ declaratorConst ] direct-abstract-declarator'
    | ( init-declarator-list ) direct-abstract-declarator'
    | epsilon
direct-abstract-declarator'' → [ declaratorConst]
   ] | ( init-declarator-list )
declaratorConst → values | identifier

init-declarator-list → init-declarator init-declarator-list'
init-declarator-list' → epsilon | , type-specifier_opt init-declarator init-declarator-list'
type-specifier_opt → type-specifier | epsilon
init-declarator → Identifiers_Variable init
init → [ Integers ] init | = initializer | epsilon

struct-or-union-specifier → struct-or-union identifierStruct_opt { struct-declaration-list } ;
identifierStruct_opt → Identifiers_Struct | epsilon
struct-or-union → struct | union
struct-declaration-list → struct-declaration struct-declaration-list' | epsilon
struct-declaration-list' → struct-declaration struct-declaration-list' | epsilon
struct-declaration → specifier-qualifier-list struct-declarator-list ;
struct-declarator-list → struct-declarator struct-declarator-list' | epsilon
struct-declarator-list' → , struct-declarator struct-declarator-list' | epsilon
struct-declarator → Identifiers_Variable struct-declarator-detail
struct-declarator-detail → : structConst | epsilon
structConst → values | identifier

enum-specifier → typdef_opt enum enum-specifierFactor
typdef_opt → typedef | epsilon
enum-specifierFactor → identifier enum-body_opt ; | { enumerator-list } ; identifier_opt
identifier_opt → epsilon | identifier
enum-body_opt → { enumerator-list } | epsilon
enumerator-list → enumerator enumerator-list' | epsilon
enumerator-list' → , enumerator enumerator-list' | epsilon
```

```
enumerator → identifier enumerator'
enumerator' → epsilon | = constExpr
constExpr → values | identifier

statement → labeled-statement | expression-statement
    | compound-statement | selection-statement
    | selection-switch | iteration-statement
    | do statement doWhile_Body | jump-statement
labeled-statement → identifier : | case caseBody : | default :
caseBody → values | identifier

statement-list → statement statement-list'
statement-list' → statement statement-list' | epsilon
expression-statement → expression ; | constant_opt type-specifier functionOrVarDef
    | enum-specifier | struct-or-union-specifier
functionOrVarDef → external-declaration ; | function-definition
expression-statement_opt → expression-statement | epsilon
semiOpt → ; | epsilon

compound-statement → { declaration-list_opt statement-list_opt }
declaration-list_opt → declaration-list | epsilon
statement-list_opt → statement-list | epsilon
selection-statement → if ( expression ) compound-statement elseBody
elseBody → else | epsilon
selection-switch → switch ( expression )
iteration-statement → while ( expressionOrBool)
   ) | for ( expression1_opt ; expression2_opt ; expression3_opt )
expression1_opt → type-specifier_opt init-declarator | epsilon
expression2_opt → expression | epsilon
expression3_opt → expression | epsilon
doWhile_Body → while ( expressionOrBool ) ;

expressionOrBool → expression | true | false
jump-statement → goto identifier ;
    | continue ; | break ; | return valueOrExpression ;
valueOrExpression → values | expression_opt
expression_opt → expression | epsilon

expression → function-call postOrAssignmentOpt | postOrAssignment
postOrAssignmentOpt → epsilon | postOrAssignmentWithoutId
postOrAssignmentWithoutId → identifier_opt postOrAssignmentFactor
postOrAssignment → identifier postOrAssignmentFactor
postOrAssignmentFactor →  epsilon
    | assignment-expression postOrAssignmentFactor
    | postfix-expression
expression' → , assignment-expression expression' | epsilon

value-expression → epsilon | unary-expression value-expression
assignment-expression → identifier isTernary | functionCallOrConstant
isTernary → epsilon | conditional-expression
functionCallOrConstant → values | assignment-operator assignmentRHS
    | unary-expression unary-expressionFactor
unary-expressionFactor → values | identifier
    | assignment-operator assignmentRHS
    | function-call
assignmentRHS → values | identifier | function-call
```

```
assignment-operator_opt → epsilon | assignment-operator
assignment-operator → = | *= | /= | %= | += | -= | <<= | >>= | &= | ^= | |=

conditional-expression → comparison-expression comparator-RHS ? comparator-RHS : comparator-RHS
cast-expression → unary-expression cast-expression | ( type-name ) cast-expression
unary-expression → postfix-expression | unary-operator
    | sizeof unary-expression | sizeof ( type-name )
unary-operator → & | * | + | - | ~ | ! | / | %
postfix-expression → comparison-expression comparator-RHS
    | ++ | --
comparator-RHS → values | primary-expression
postfix-expression' → [ expression ] postfix-expression'
    | ( postfix-expression'-opt ) postfix-expression'
    | . identifier postfix-expression'
    | -> identifier postfix-expression'
    | epsilon
comparison-expression → < | > |  |  | == | !=
postfix-expression'-opt → argument-expression-list | epsilon
primary-expression → constant | string | ( expression )

argument-expression-list → values argument-expression-list'
    | postOrAssignment argument-expression-list'
argument-expression-list' → , values argExpFact argument-expression-list'
    | epsilon
argExpFact → epsilon | postOrAssignment | values
initializer → values | assignment-expression | { initializer-list }
values → sign values_opt | Characters | Strings
values_opt → Integers | Floats | Long Long_Num_opt
Long_Num_opt → Long | epsilon
sign → + | - | epsilon

initializer-list → initializer initializer-list'
initializer-list' → , initializer initializer-list' | epsilon
constant → integer-constant | character-constant
    | floating-constant | enumeration-constant
identifier → Identifiers_Variable
type-qualifier → const | volatile
```

4.3.5 Description of Grammar Rules

We have **121** Grammar Rule :

```
1. translation-unit:
   - `translation-unit → statement-list`
     This defines the `translation-unit` as a `statement-list`. A translation unit is usually the
entire source code to be compiled.

2. translation-unitFactor:
   - `translation-unitFactor → translation-unit' | epsilon`
     Allows for an optional component following the `translation-unit`.

3. translation-unit':
   - `translation-unit' → external-declaration translation-unit' | epsilon`
     Describes a sequence of `external-declaration` followed by more `external-declaration` or
nothing (`epsilon` means empty).
```

4. external-declaration:
   - `external-declaration → external-declarationFactor`
     References another rule for further processing of external declarations.

5. external-declarationFactor:
   - `external-declarationFactor → init-declarator-list`
     Specifies that an external declaration can be a list of initializations.

6. constant_opt:
   - `constant_opt → const | epsilon`
     Defines an optional `const` keyword or nothing.

7. function-definition:
   - `function-definition → Identifiers_Function ( parameter-list ) function-prototype`
     A function definition includes a function identifier, parameters, and a body or prototype.

8. function-prototype:
   - `function-prototype → compound-statement | ;`
     The function body can either be a compound statement (actual implementation) or a semicolon (`;`), indicating a prototype.

9. parameter-list:
   - `parameter-list → type-specifier Identifiers_Variable assignment-expression_opt parameter-list'`
     Specifies the list of parameters, which includes types and optionally assignment expressions.
   - `parameter-list → epsilon`
     Allows for an empty parameter list.

10. assignment-expression_opt:
    - `assignment-expression_opt → assignment-expression | epsilon`
      An optional assignment expression or nothing.

11. parameter-list':
    - `parameter-list' → , parameter-listFact | epsilon`
      Continuation of a parameter list after a comma.

12. parameter-listFact:
    - `parameter-listFact → type-specifier Identifiers_Variable | . . .`
      Specifies a parameter list fact or indicates variadic arguments (`...`).

13. function-call:
    - `function-call → Identifiers_Function ( argument-expression-list )`
      Defines a function call that includes the function name and a list of arguments.

14. declaration-specifiers_opt:
    - `declaration-specifiers_opt → declaration-specifiers | epsilon`
      An optional declaration specifier or nothing.

15. declaration:
    - `declaration → ( declaration_opt ) ;`
      A declaration is optionally enclosed in parentheses and ends with a semicolon.

16. declaration_opt:
    - `declaration_opt → init-declarator-list | epsilon`
      Optional initialization or an empty declaration.

17. declaration-list:
    - `declaration-list → declaration declaration-list'`
      A list of declarations.

18. declaration-list':
    - `declaration-list' → epsilon | declaration declaration-list'`
      Optional continuation of the declaration list.

19. declaration-specifiers:
    - `declaration-specifiers → storage-class-specifier declaration-specifiers`
    - `declaration-specifiers → type-specifier declaration-specifiers`
    - `declaration-specifiers → type-qualifier declaration-specifiers`
    - `declaration-specifiers → epsilon`
      Specifies the different declaration specifiers, such as storage class, types, and qualifiers, or nothing.

20. storage-class-specifier:
    - `storage-class-specifier → auto | register | static | extern | typedef`
      Defines the storage class specifier.

21. type-specifier:
    - `type-specifier → void | char pointerLoop`
    - `type-specifier → short pointerLoop | int pointerLoop`
    - `type-specifier → long long_opt pointerLoop`
    - `type-specifier → float pointerLoop | double pointerLoop`
    - `type-specifier → signed | unsigned`
    - `type-specifier → struct-or-union-specifier | enum-specifier`
      Enumerates different possible type specifiers, including built-in types and user-defined types like `struct` and `enum`.

22. long_opt:
    - `long_opt → epsilon | long`
      An optional `long` keyword or nothing.

23. specifier-qualifier-list:
    - `specifier-qualifier-list → specifier-qualifier-listFactor specifier-qualifier-list | epsilon`
      A combination of specifier and qualifier list items.

24. specifier-qualifier-listFactor:
    - `specifier-qualifier-listFactor → type-specifier | type-qualifier`
      Either a type specifier or type qualifier.

25. type-name:
    - `type-name → specifier-qualifier-list type-name_opt`
      A type name includes a specifier-qualifier list.

26. type-name_opt:
    - `type-name_opt → abstract-declarator | epsilon`
      An optional abstract declarator or nothing.

27. abstract-declarator:
    - `abstract-declarator → pointer | pointer_opt abstract-declaratorFactor`
      Defines an abstract declarator involving pointers.

28. pointerLoop:
    - `pointerLoop → epsilon | * pointerLoop`
      An optional loop of asterisks to represent pointer levels.

29. pointer_opt:
    - `pointer_opt → * | epsilon`
      An optional pointer indicator or nothing.

30. abstract-declaratorFactor:
    - `abstract-declaratorFactor → direct-abstract-declarator | epsilon`
      Further detailing of the abstract declarator or nothing.

31. direct-abstract-declarator:
    - `direct-abstract-declarator → ( abstract-declarator ) direct-abstract-declarator'`
    - `direct-abstract-declarator'' → [ declaratorConst] ] | ( init-declarator-list )`
      Specifies a detailed form of an abstract declarator.

32. direct-abstract-declarator':
    - `direct-abstract-declarator' → [ declaratorConst ] direct-abstract-declarator'`
    - `direct-abstract-declarator' → ( init-declarator-list ) direct-abstract-declarator'`
    - `direct-abstract-declarator' → epsilon`
      Continuation and detailing of the direct abstract declarator.

33. declaratorConst:
    - `declaratorConst → values | identifier`
      A constant declarator or an identifier.

34. init-declarator-list:
    - `init-declarator-list → init-declarator init-declarator-list'`
      A list of declarators with initial values.

35. init-declarator-list':
    - `init-declarator-list' → epsilon | , type-specifier_opt init-declarator
init-declarator-list'`
      An optional continuation of the initialization declarator list.

36. type-specifier_opt:
    - `type-specifier_opt → type-specifier | epsilon`
      An optional type specifier or nothing.

37. init-declarator:
    - `init-declarator → Identifiers_Variable init`
      Defines a variable identifier followed by an initialization.

38. init:
    - `init → [ Integers ] init | = initializer | epsilon`
      The actual initialization, which could include arrays or assignment expressions.

39. struct-or-union-specifier:
    - `struct-or-union-specifier → struct-or-union identifierStruct_opt { struct-declaration-list }
;`
      Specifies a structure or union, optionally named.

40. identifierStruct_opt:
    - `identifierStruct_opt → Identifiers_Struct | epsilon`
      An optional struct identifier.

41. struct-or-union:
    - `struct-or-union → struct | union`
      Defines either a `struct` or `union`.

42. struct-declaration-list:
    - `struct-declaration-list → struct-declaration struct-declaration-list' | epsilon`
      A list of declarations within a structure.

43. struct-declaration-list':
    - `struct-declaration-list' → struct-declaration struct-declaration-list' | epsilon`
      Continuation or nothing.

44. struct-declaration:
    - `struct-declaration → specifier-qualifier-list struct-declarator-list ;`
      A single declaration inside a structure, with a type and a list of declarators.

45. struct-declarator-list:
    - `struct-declarator-list → struct-declarator struct-declarator-list' | epsilon`
      A list of declarators inside a structure.

46. struct-declarator-list':
    - `struct-declarator-list' → , struct-declarator struct-declarator-list' | epsilon`
      Continuation or nothing.

47. struct-declarator

:
    - `struct-declarator → Identifiers_Variable struct-declarator-detail`
      A variable declaration with additional details.

48. struct-declarator-detail:
    - `struct-declarator-detail → : structConst | epsilon`
      Specifies further details for a structure declarator, such as bit fields.

49. structConst:
    - `structConst → values | identifier`
      Constant values for structure declarators.

50. enum-specifier:
    - `enum-specifier → typdef_opt enum enum-specifierFactor`
      Specifies an enumeration.

51. typdef_opt:
    - `typdef_opt → typedef | epsilon`
      An optional `typedef` keyword.

52. enum-specifierFactor:
    - `enum-specifierFactor → identifier enum-body_opt ;`
    - `enum-specifierFactor → { enumerator-list } ; identifier_opt`
      Defines the enumeration, either by name or with enumerators.

53. identifier_opt:
    - `identifier_opt → epsilon | identifier`
      An optional identifier.

54. enum-body_opt:
    - `enum-body_opt → { enumerator-list } | epsilon`
      An optional enumeration body.

55. enumerator-list:
    - `enumerator-list → enumerator enumerator-list' | epsilon`
      A list of enumerators.

56. enumerator-list':
    - `enumerator-list' → , enumerator enumerator-list' | epsilon`
      Continuation or nothing.

57. enumerator:
    - `enumerator → identifier enumerator'`
      Defines an enumerator.

58. enumerator':
    - `enumerator' → epsilon | = constExpr`
      Optionally assigns a value to the enumerator.

59. constExpr:
    - `constExpr → values | identifier`
      Defines a constant expression.

60. statement:
    - `statement → labeled-statement | expression-statement`
    - `statement → compound-statement | selection-statement`
    - `statement → selection-switch | iteration-statement`
    - `statement → do statement doWhile_Body | jump-statement`
      Enumerates the types of statements possible, including loops, jumps, expressions, and selections.

61. labeled-statement:
    - `labeled-statement → identifier : | case caseBody : | default :`
      Defines a statement with a label or a case in a switch statement.

62. caseBody:
    - `caseBody → values | identifier`
      A value or identifier for a case statement.

63. statement-list:
    - `statement-list → statement statement-list'`
      A list of statements.

64. statement-list':
    - `statement-list' → statement statement-list' | epsilon`
      Continuation or nothing.

65. expression-statement:
    - `expression-statement → expression ; | constant_opt type-specifier functionOrVarDef`
    - `expression-statement → enum-specifier | struct-or-union-specifier`
      An expression or declaration statement.

66. functionOrVarDef:
    - `functionOrVarDef → external-declaration ; | function-definition`
      Indicates either a function or variable definition.

67. expression-statement_opt:
    - `expression-statement_opt → expression-statement | epsilon`
      An optional expression statement.

68. semiOpt:
    - `semiOpt → ; | epsilon`
      An optional semicolon.

69. compound-statement:
    - `compound-statement → { declaration-list_opt statement-list_opt }`
      A block enclosed by braces, containing declarations and statements.

70. declaration-list_opt:
    - `declaration-list_opt → declaration-list | epsilon`
      An optional declaration list.

71. statement-list_opt:
    - `statement-list_opt → statement-list | epsilon`
      An optional statement list.

72. selection-statement:
    - `selection-statement → if ( expression ) compound-statement elseBody`
      An `if-else` selection statement.

73. elseBody:
    - `elseBody → else | epsilon`
      An optional `else` clause.

74. selection-switch:
    - `selection-switch → switch ( expression )`
      Defines a `switch` statement.

75. iteration-statement:
    - `iteration-statement → while ( expressionOrBool)`
    - `iteration-statement → for ( expression1_opt ; expression2_opt ; expression3_opt )`
      Describes looping statements.

76. expression1_opt:
    - `expression1_opt → type-specifier_opt init-declarator | epsilon`
      An optional first expression in a `for` loop.

77. expression2_opt:
    - `expression2_opt → expression | epsilon`
      An optional second expression in a `for` loop.

78. expression3_opt:
    - `expression3_opt → expression | epsilon`
      An optional third expression in a `for` loop.

79. doWhile_Body:
    - `doWhile_Body → while ( expressionOrBool ) ;`
      Defines a `do-while` loop body.

80. expressionOrBool:
    - `expressionOrBool → expression | true | false`
      Specifies an expression or boolean values.

81. jump-statement:
    - `jump-statement → goto identifier ;`
    - `jump-statement → continue ; | break ; | return valueOrExpression ;`
      Jump statements, including `goto`, `break`, `continue`, and `return`.

82. valueOrExpression:
    - `valueOrExpression → values | expression_opt`
      A value or expression.

83. expression_opt:
    - `expression_opt → expression | epsilon`
      An optional expression.

84. expression:
    - `expression → function-call postOrAssignmentOpt | postOrAssignment`
      An expression involves a function call or a series of assignments.

85. postOrAssignmentOpt:
    - `postOrAssignmentOpt → epsilon | postOrAssignmentWithoutId`
      An optional post or assignment.

86. postOrAssignmentWithoutId:
    - `postOrAssignmentWithoutId → identifier_opt postOrAssignmentFactor`
      Assignment operations without an identifier.

87. postOrAssignment:
    - `postOrAssignment → identifier postOrAssignmentFactor`
      A post or assignment operation.

88. postOrAssignmentFactor:
    - `postOrAssignmentFactor → epsilon`
    - `postOrAssignmentFactor → assignment-expression postOrAssignmentFactor`
    - `postOrAssignmentFactor → postfix-expression`
      Specifies how assignments and postfix expressions follow the assignment identifier.

89. expression':
    - `expression' → , assignment-expression expression' | epsilon`
      Optional continuation of assignment expressions.

90. value-expression:
    - `value-expression → epsilon | unary-expression value-expression`
      Expression involving values and unary operators.

91. assignment-expression:
    - `assignment-expression → identifier isTernary | functionCallOrConstant`
      An assignment expression involving identifiers or constants.

92. isTernary:
    - `isTernary → epsilon | conditional-expression`
      A ternary (conditional) expression or nothing.

93. functionCallOrConstant:
    - `functionCallOrConstant → values | assignment-operator assignmentRHS`
    - `functionCallOrConstant → unary-expression unary-expressionFactor`
      A function call or constant involving values and assignment operators.

94. unary-expressionFactor:
    - `unary-expressionFactor → values | identifier`
    - `unary-expressionFactor → assignment-operator assignmentRHS`
    - `unary-expressionFactor → function-call`
      Factors in unary expressions.

95. assignmentRHS:
    - `assignmentRHS → values | identifier | function-call`
      Specifies the right-hand side of an assignment.

96. assignment-operator_opt:
    - `assignment-operator_opt → epsilon | assignment-operator`
      Optional assignment operator.

97. assignment-operator:
    - `assignment-operator → = | *= | /= | %= | += | -= | <<= | >>= | &= | ^= | |=`
      Various assignment operators.

98. conditional-expression:
    - `conditional-expression → comparison-expression comparator-RHS ? comparator-RHS :
comparator-RHS`
      Ternary conditional expressions.

99. cast-expression:
    - `cast-expression → unary-expression cast-expression | ( type-name ) cast-expression`
      Type-casting expressions.

100. unary-expression:
    - `unary-expression → postfix-expression | unary-operator`
    - `unary-expression → sizeof unary-expression | sizeof ( type-name )`
      Unary expressions, including `sizeof` operations.

101. unary-operator:
    - `unary-operator → & | * | + | - | ~ | ! | / | %`
      Various unary operators.

102. postfix-expression:
    - `postfix-expression → comparison-expression comparator-RHS`
    - `postfix-expression

 → ++ | --`
      Postfix expressions involving increments/decrements.

103. comparator-RHS:
    - `comparator-RHS → values | primary-expression`
      Right-hand side of comparisons.

104. postfix-expression':
    - `postfix-expression' → [ expression ] postfix-expression'`
    - `postfix-expression' → ( postfix-expression'-opt ) postfix-expression'`
    - `postfix-expression' → . identifier postfix-expression'`
    - `postfix-expression' → -> identifier postfix-expression'`
    - `postfix-expression' → epsilon`
      Postfix expression operations, like indexing and member access.

105. comparison-expression:
    - `comparison-expression → < | > |  |  | == | !=`
      Various comparison operators.

106. postfix-expression'-opt:
    - `postfix-expression'-opt → argument-expression-list | epsilon`
      Optional postfix expression continuation.

107. primary-expression:
    - `primary-expression → constant | string | ( expression )`
      A primary expression could be a constant, string, or parenthesized expression.

108. argument-expression-list:
    - `argument-expression-list → values argument-expression-list'`
    - `argument-expression-list → postOrAssignment argument-expression-list'`
      A list of arguments for function calls.

109. argument-expression-list':
    - `argument-expression-list' → , values argExpFact argument-expression-list'`
    - `argument-expression-list' → epsilon`
      Continuation or nothing.

110. argExpFact:
    - `argExpFact → epsilon | postOrAssignment | values`
      Factors involving argument expressions.

111. initializer:
    - `initializer → values | assignment-expression | { initializer-list }`
      An initializer for variable declarations.

112. values:
    - `values → sign values_opt | Characters | Strings`
    - `values_opt → Integers | Floats | Long Long_Num_opt`
    - `Long_Num_opt → Long | epsilon`
    - `sign → + | - | epsilon`
      Specifies numerical values with signs and their various types.

113. initializer-list:
    - `initializer-list → initializer initializer-list'`
      A list of initializers.

114. initializer-list':
    - `initializer-list' → , initializer initializer-list' | epsilon`
      Continuation or nothing.

115. constant:
    - `constant → integer-constant | character-constant`
    - `constant → floating-constant | enumeration-constant`
      Various types of constant values.

116. identifier:
    - `identifier → Identifiers_Variable`
      An identifier for a variable.

117. type-qualifier:
    - `type-qualifier → const | volatile`
      Specifies the type qualifiers.

4.4 Test Cases

4.4.1 Sniped C-code

```c
 switch (number) {
     case 1:
         printf("You entered one.\n");
         break;
     case 2:
         printf("You entered two.\n");
         break;
     case 3:
         printf("You entered three.\n");
         break;
     case 4:
         printf("You entered four.\n");
         break;
     case 5:
         printf("You entered five.\n");
         break;
     default:
         printf("Invalid number! Please enter a number between 1 and 5.\n");
 }
 int x = y>0 ? 1 : 0;
 int arr[3] = {5};
char z = 'c';
 float h = 3.4f;
 long tt = 864l;
 x = y-3;
 x = z + h;
 y = 5;
 int* x = 3;
 int add(int a, char* b);
 int add(int a, char* b) {
     return a + add(4);
 }
 int y=0;
 int z = 17;
 if (x<5) {
     y = 5;
 }
 else {
     y = 6;
```

```c
}
for (int i=0; i<3; i++) {
    x = x+i;
}
while (y==3) {y+=3;}
while (true) {
    x++;
    if (x == 10) {
        break;
    }
}
do{
   z = x + y;
} while(x==3);
int main() {
    int x = 3;
    int y = 4;
}
int add2(int x, ...) {
    return a + add2(x+2);
}
add(3,"4");
x = add(3,4);
enum color;
enum Color {
    Red,    // 0
    Green,  // 1
    Blue    // 2
};
typedef enum {
    On,
    Off
} SwitchState;
enum {
    WindowWidth = 800,
    WindowHeight = 600
};
struct ComplexNumber {
    double real;
    double imaginary;
};
double 27961broccoliVar1616;
```

4.4.2 Analysis C-code

The following analysis applies to the C code snippet:

1. **Switch Statement**:

```c
    switch (number) {
        case 1:
            printf("You entered one.\n");
            break;
        case 2:
            printf("You entered two.\n");
            break;
```

```
        case 3:
            printf("You entered three.\n");
            break;
        case 4:
            printf("You entered four.\n");
            break;
        case 5:
            printf("You entered five.\n");
            break;
        default:
            printf("Invalid number! Please enter a number between 1 and 5.\n");
    }
```

Rule: ‘selection-switch‘ utilizing the ‘case‘ keyword for conditional execution and ‘default‘ as a fallback.

2. **Conditional Operator**:

```
    int x = y > 0 ? 1 : 0;
```

Rule: ‘conditional-expression‘ used to assign a value based on the condition ‘y ¿ 0‘.

3. **Array Initialization**:

```
    int arr[3] = {5};
```

Rule: ‘init-declarator‘ with ‘initializer-list‘ indicating array initialization.

4. **Variable Declarations and Initializations**:

```
    char z = 'c';
    float h = 3.4f;
    long tt = 8641;
```

Rule: Basic type variable declarations using ‘type-specifier‘ and ‘init-declarator‘.

5. **Pointer and Function Declarations**:

```
    int* x = 3; // Incorrect, it should probably be &some_variable or nullptr
    int add(int a, char* b);
```

Rule: Utilizes ‘pointer‘ and ‘function-definition‘. Note, the pointer assignment to ‘3‘ is semantically incorrect in C.

6. **Function Definition**:

```
    int add(int a, char* b) {
        return a + add(4);
        /* Note: 'add(4)' is incorrect as it does not match the function
                                signature of two arguments *\
    }
```

Rule: An instance of ‘function-definition‘ with parameters and a body.

7. **Loop Structures**:

```
    for (int i = 0; i < 3; i++) {
        x = x + i;
    }
    while (y == 3) {
        y += 3;
    }
    do {
        z = x + y;
    } while (x == 3);
```

Rule: Examples of ‘iteration-statement‘ using ‘for‘, ‘while‘, and ‘do-while‘ loops.

8. **Enumeration and Structure Definition**:

```
    enum color;
    typedef enum {
        On,
        Off
    } SwitchState;
    struct ComplexNumber {
        double real;
        double imaginary;
    };
```

Rule: ‘enum-specifier‘ and ‘struct-or-union-specifier‘ define enumeration and structure types, respectively.

9. **Incorrect Identifiers and Usage**:

```
    double 27961broccoliVar1616; // Incorrect identifier: cannot start with a digit
```

This code will result in a syntax error as per standard C rules.

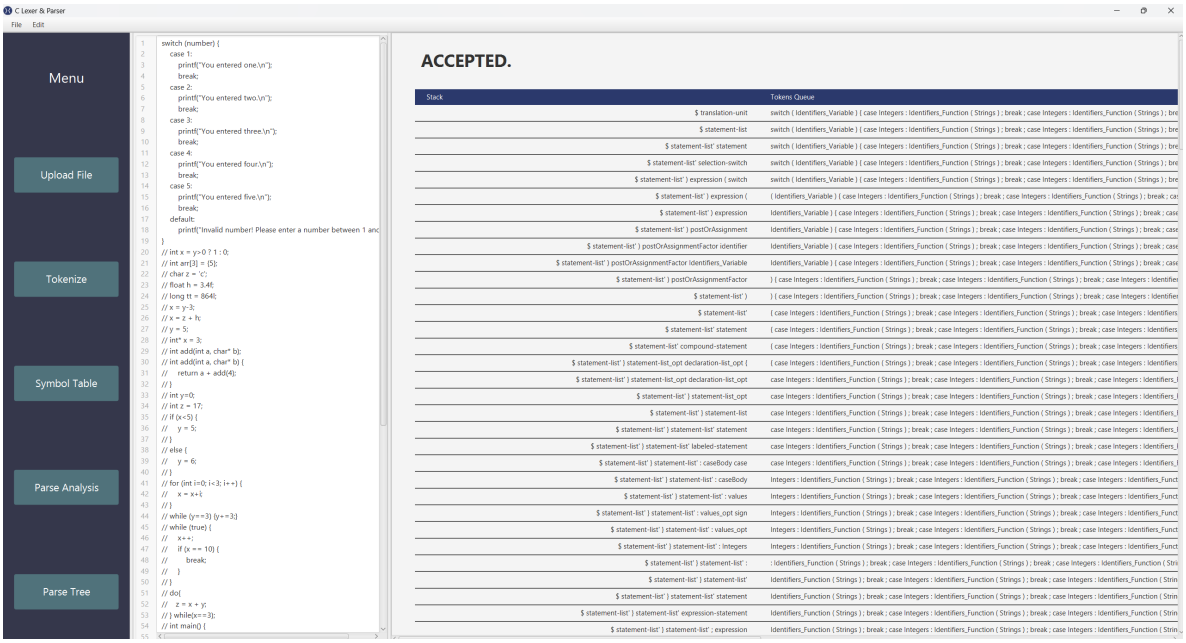
10. **Function Overloading and Variadic Functions**:

```
    int add2(int x, ...) {
        return a + add2(x + 2);
    }
```

Rule: This resembles ‘function-definition‘, but it uses variadic arguments denoted by ‘...‘.

The grammar rules cover all of these constructs.

4.5 GUI Application



8

Figure 35: Parse Analysis Table1

1. Left Panel - Menu and Code Input:
This panel includes menu options like “File,” “Edit,” and specific functionality such as “Upload File,” “Tokenize,” “Symbol Table,” “Parse Analysis,” and “Parse Tree.” Below the menu, there is a text area where code is displayed. In this instance, it shows a series of switch-case statements in a programming language, along with other commented and uncommented code snippets which include variable declarations, arithmetic operations, and conditional statements.

2. Middle Panel - Analysis Process and Stack:
This section provides detailed information about the parsing process. It displays a hierarchical structure of parsing states and operations, showing how the input code is being analyzed and broken down into manageable components. This “Stack” section helps users understand the step-by-step processing of their code by the software.

3. Right Panel - Tokens Queue and Analysis Results:
This displays the “Tokens Queue,” which lists all tokens extracted from the input code, classified by their types such as identifiers, integers, strings, etc. This section provides a comprehensive overview of the elements recognized in the code and their respective classifications. Additionally, the top of this panel prominently displays the word “ACCEPTED,” indicating that the input code has been successfully parsed and analyzed without errors.

This interface is typical of development environments and tools used for programming education, code analysis, and debugging, providing a comprehensive toolset for examining and understanding code structure and syntax.

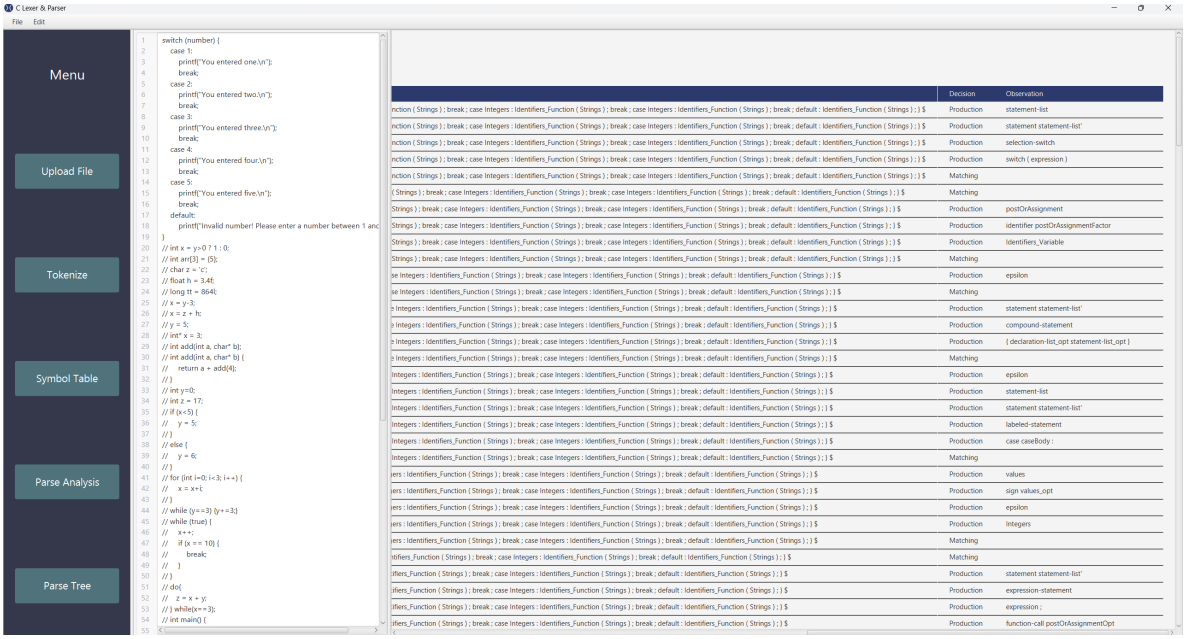


Figure 36: Parse Analysis Table2

1. Middle Panel - Tokens Queue and Syntax Tree : This section lists the sequence of tokens recognized from the input code. Each token is displayed with a corresponding syntactic category, such as identifiers, integers, strings, and their contextual usage within the code (e.g., function, expression). This organized view helps users track how the code is tokenized and interpreted by the software.

2. Right Panel - Production Rules and Observations
The far right section details the production rules applied during the parsing process, which are categorized under "Production," "Matching," and "Observation." These entries provide insights into the parsing decisions and grammatical constructions recognized in the code. For instance, it lists productions like "statement-list," "compound-statement," and various matching cases based on syntax elements identified in the tokens queue.

This comprehensive interface offers a robust toolset for code analysis, useful in academic settings, or for developers seeking a deeper understanding of code parsing and debugging. The design facilitates a clear and methodical examination of programming constructs, syntactic validity, and logical structure.

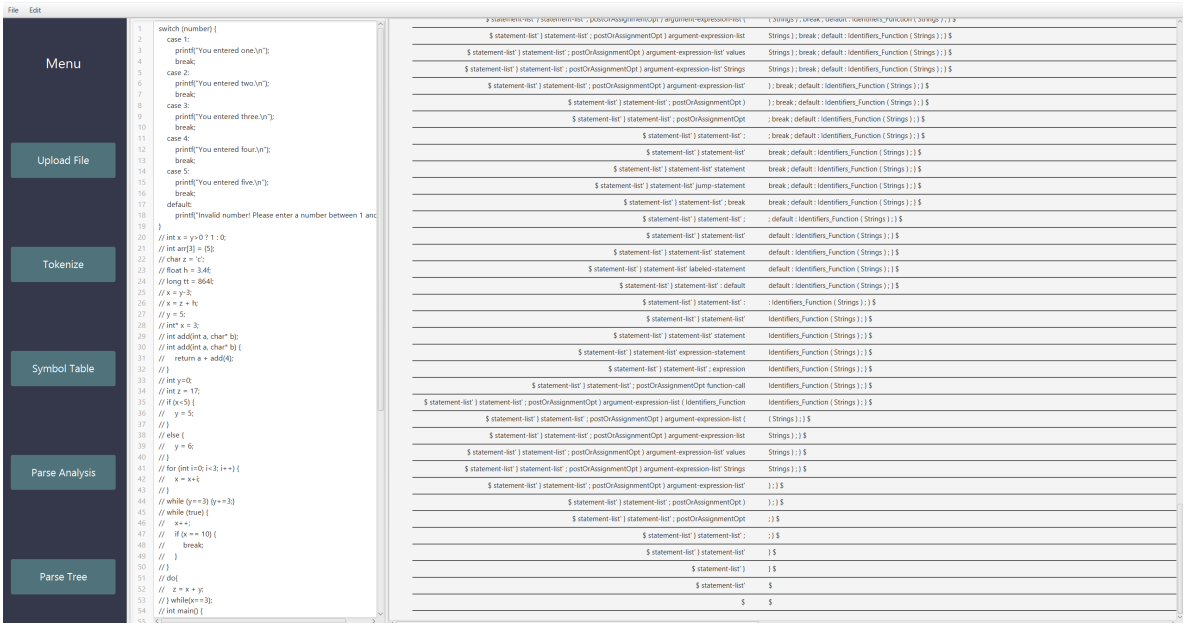


Figure 37: Parse Analysis Table3

1. **Central Panel - Parse Analysis:** This expanded view shows the structure and parsing steps applied to a given segment of code. It lists various syntactic constructs such as `statement-list`, `postOrAssignmentOpt`, `argument-expression-list`, and others. Each entry in this list corresponds to a particular action in the parsing process, helping to break down complex code structures into more manageable elements. The detailed breakdown shows how each part of the input code correlates to specific grammatical rules, illustrating the parsing strategy used by the software.

2. **Right Panel - Syntax and Token Analysis:**
   The right section of the interface categorizes elements of the code into tokens and associates them with their syntactic functions such as `identifiers`, `functions`, `strings`, and control structures (`break`, `default`). It provides a token queue that systematically lists how each piece of the code is identified and categorized, enhancing the understanding of how the parser interprets each component.

   Together, these panels are instrumental for users who need to analyze, debug, or learn about the syntactic structure of programming languages through visual breakdowns of code. This setup is particularly beneficial in educational environments or for developers seeking to understand or improve parsing algorithms.

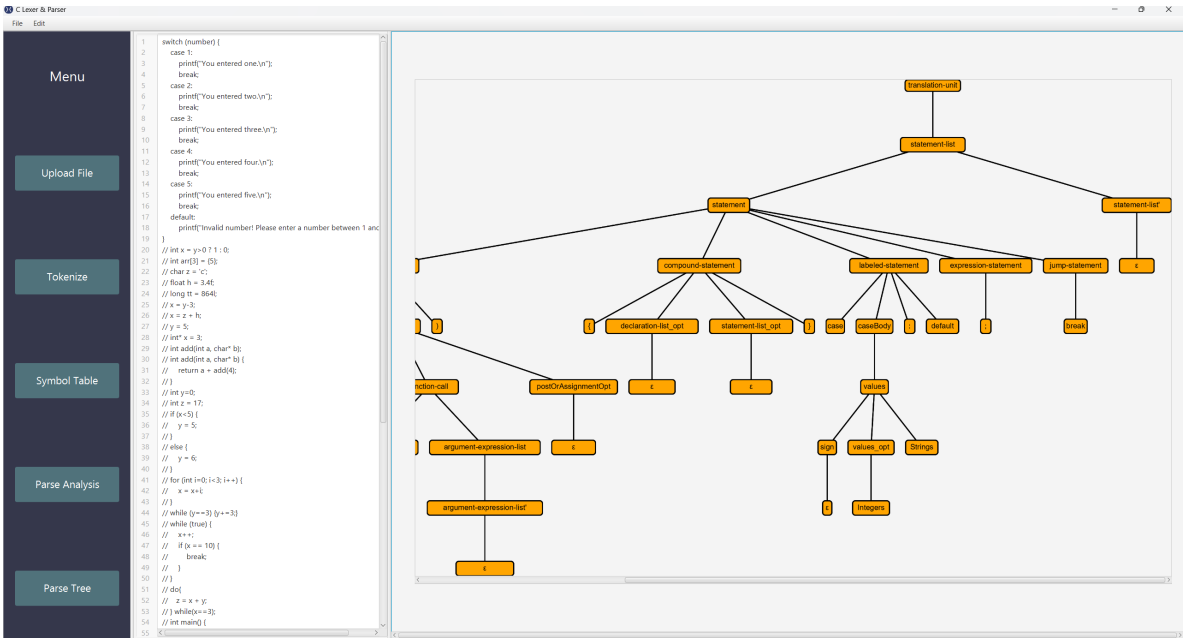


Figure 38: Parse Tree1

1. **Left Panel - Menu and Code Input**: This area contains the raw code and various functional options related to file management and code analysis.

2. **Central Panel - Parse Tree Visualization**: This panel features a graphical representation of the parse tree. It starts from the "translation-unit" at the top, branching out into various categories such as "statement-list," which further divides into "compound-statement," "labeled-statement," "expression-statement," and "jump-statement." Each node represents a syntactic category with connections showing their relationships, such as "declaration-list$_opt$" *and* "*statement*−*list*$_o$*pt*" *under* "*compound*−*statement*." *Specificsyntaxconstructionslike*" *caseBody*, "" *default*, " *and*" *break*" *arealsodetailed*, *illustratingthestructureofcontrolflowstatementswithinthecode*.

This visualization is crucial for users seeking to analyze or debug the parsing process of their code, providing a clear and organized view of how each piece of the code is syntactically connected. The parse tree is a vital tool in compiler design and code analysis, helping to identify errors in code structure or to understand deeper aspects of language syntax for educational purposes.

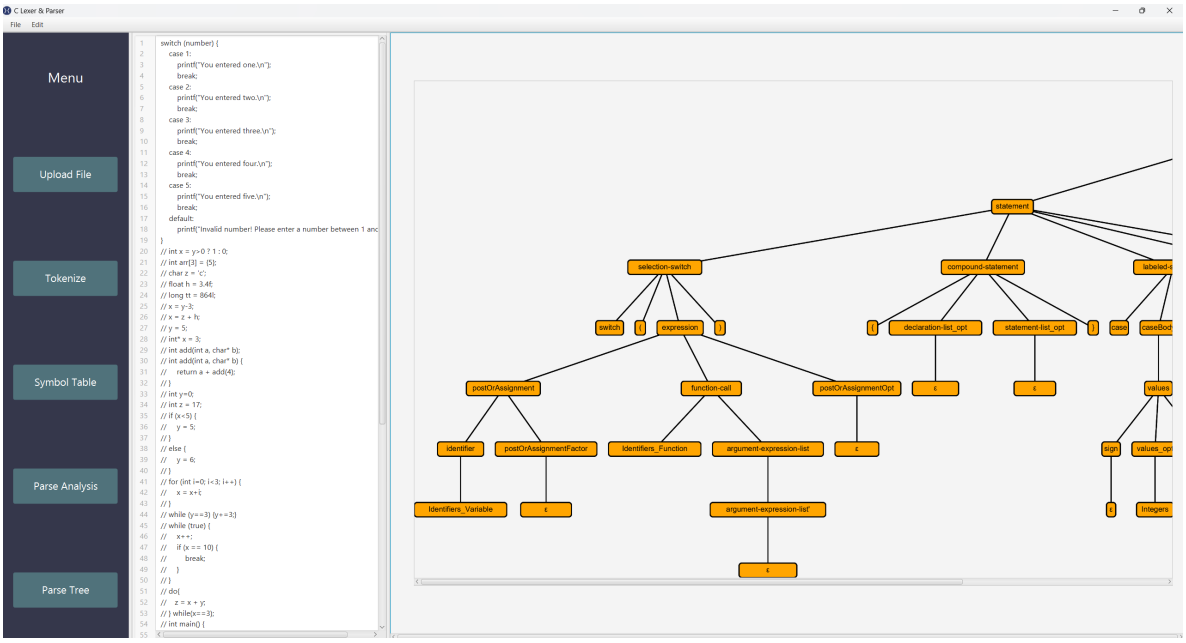


Figure 39: Parse Tree2

1.Central Panel - Parse Tree Visualization: The panel features a graphical representation of the parse tree for a segment of code involving a switch statement. At the top of the tree, there is a node labeled "selection-switch," which branches out into two sub-nodes: "switch" and "expression." This highlights the structure of the switch statement, where the "expression" node likely represents the condition evaluated in the switch.
Further down, the tree expands into more specific categories:

- **"compound-statement"** branches into "declaration-list_opt" and "statement-list_opt," indicating optional lists of declarations and statements within the compound structure of the code.
- **"caseBody"** represents the body of a case within the switch, further breaking down into nodes like "values," where specific values handled by cases are detailed.

2.Detailed Nodes for Expression and Assignments:

- The "postOrAssignment" and "function-call" nodes suggest the presence of assignments or function calls within the switch cases.
- Nodes like "identifier," "postOrAssignmentFactor," and "argument-expression-list" point to specific identifiers used in the code and how expressions are structured and evaluated.

This visualization is crucial for understanding the detailed syntactic structure of the switch statement within the code. It helps in visualizing how different components of the code are hierarchically related and managed by the parser. Such tools are immensely useful in educational contexts for teaching compiler design or in professional settings for debugging complex code structures.

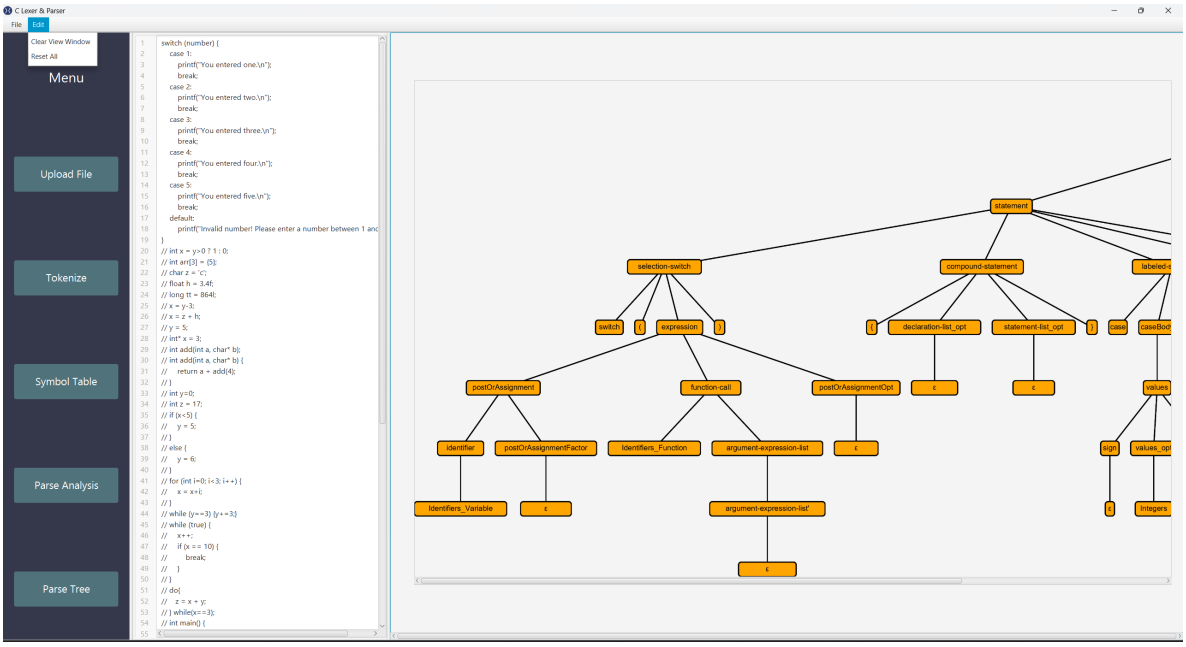


Figure 40: Parse Tree3

1.Menu and Control Options: On the left, the interface provides a vertical menu with functionalities such as "Upload File," "Tokenize," "Symbol Table," "Parse Analysis," and "Parse Tree." Additional options like "Clear View Window" and "Reset All" are available to manage the workspace and reset the analysis.

2.Central Panel - Parse Tree Visualization: This panel prominently features the parse tree for a switch-case statement within a programming code. The tree is rooted at a node labeled "selection-switch," which bifurcates into "switch" and "expression," indicating the main elements of the switch statement. **The tree further expands to nodes like:**

- **"compound-statement":** This node divides into "declaration-list_opt" and "statement-list_opt," illustrating optional components within the compound statement structure.
- **"caseBody":** Shows the structure within a case of the switch, breaking down into "values," indicating the values handled by the cases.

3.Detailed Nodes for Syntax and Operations:

- The nodes "postOrAssignment" and "function-call" suggest the operations and functions performed within the cases.
- "identifier" and "postOrAssignmentFactor" detail the variables and assignments involved, while "argument-expression-list" represents lists of arguments used in functions.

This interface is instrumental for users involved in software development or education, providing a graphical representation of how code is parsed and interpreted. The parse tree visualization is a crucial tool for debugging, learning, and enhancing the understanding of programming syntax and structure.

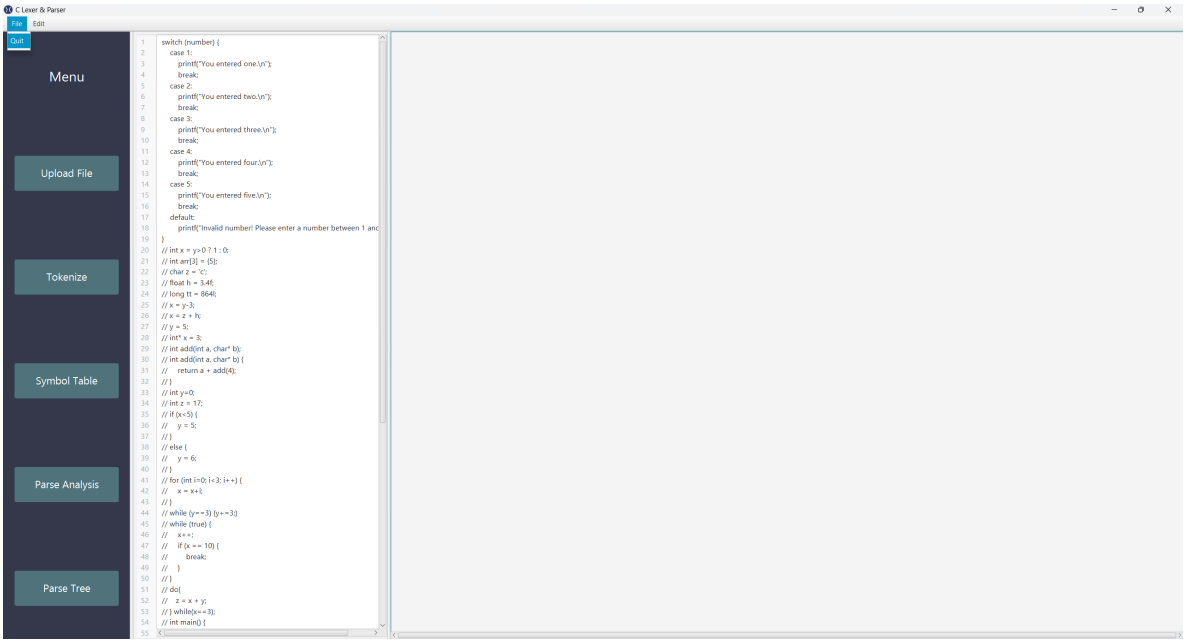


Figure 41: Test Cases

1. **Menu and Basic Functionalities:** On the left side of the interface, there is a vertical menu with options such as "Upload File," "Tokenize," "Symbol Table," "Parse Analysis," and "Parse Tree." The menu also includes basic file management options like "File" and "Edit" at the top, and a "Quit" option at the bottom, which is likely used to exit the application.

2. **Main Workspace:** The rest of the interface is a large, empty white space, indicating a primary workspace. This area would typically display the results of the selected functions from the menu, such as the output of tokenization, symbol tables, parse analysis, or the parse tree visualization of the uploaded code. However, in this snapshot, no such analysis or data is visible, suggesting that the interface is in a default or initial state

This minimalistic setup is indicative of the application's readiness to receive user input or commands, providing a straightforward environment for users to begin working with their code. This design is beneficial for focusing on specific tasks without the distraction of pre-loaded data or panels.

5 CONCLUSION

5.1 Syntax Analyser vs Lexical Analyser

Syntax vs. Lexical Analyser

Syntax Analyser	Lexical Analyser
The syntax analyser mainly deals with recursive constructs of the language.	The lexical analyser eases the task of the syntax analyser.
The syntax analyser works on tokens in a source program to recognize meaningful structures in the programming language.	The lexical analyser recognizes the token in a source program.
It receives inputs, in the form of tokens, from lexical analysers.	It is responsible for the validity of a token supplied by the syntax analyser

Figure 42: Syntax vs Lexical Analyser

5.2 Disadvantages of using Syntax Analyser

It will never determine if a token is valid or not Not helps you to determine if an operation performed on a token type is valid or not You can't decide that token is declared & initialized before it is being used.

5.3 Summary

- Syntax analysis is a second phase of the compiler design process that comes after lexical analysis.
- The syntactical analyser helps you to apply rules to the code.
- Parse checks that the input string is well-formed, and if not, reject it.
- Parsing techniques are divided into two different groups: Top-Down Parsing, Bottom-Up Parsing.
- A grammar is a set of structural rules which describe a language.

References

[1] C Reference. *C Language Reference*. https://en.cppreference.com/w/c/language.
Accessed: February 12, 2024.

[2] GeeksforGeeks. *C Keywords*. https://www.geeksforgeeks.org/c-keywords/.
Accessed: February 14, 2024.

[3] Tutorialspoint. *C Variables*. https://www.tutorialspoint.com/cprogramming/c_variables.htm.
Accessed: February 14, 2024.

[4] Programiz. *C Function Identifier*. https://www.programiz.com/c-programming/c-identifiers.
Accessed: February 28, 2024.

[5] Tutorialspoint. *C Functions*. https://www.tutorialspoint.com/cprogramming/c_functions.htm.
Accessed: March 01, 2024.

[6] University of Idaho. *ANSI C Grammar*. http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf.
Accessed: April 25, 2024.

[7] Quut. *ANSI C Grammar*. https://www.quut.com/c/ANSI-C-grammar-y.html.
Accessed: May 02, 2024.

[8] Guru99. *Syntax Analysis Types*. https://www.guru99.com/syntax-analysis-parsing-types.html.
Accessed: May 04, 2024.

[9] Testbook. *Context Free Grammar*. https://testbook.com/gate/context-free-grammar-notes.
Accessed: May 08, 2024.

[10] Naukri Learning. *Left Factoring in Compiler Design*. https://www.naukri.com/code360/library/left-factoring-in-compiler-design.
Accessed: May 08, 2024.

[11] TutorialsPoint. *What is Left Recursion and How it is Eliminated*. https://www.tutorialspoint.com/what-is-left-recursion-and-how-it-is-eliminated.
Accessed: May 08, 2024.