



Design of Compilers project

CSE439

Made by:

Hassan Walid Mohamed Almarsafy

Aley Amin Ahmed Shawky

Ahmed Mohamed Elsayed Abdelaziz Fadly

Seif Elden Samir Mohamed

Mazen sameh shawky elshabassy

Represented to:

Dr. Wafaa Samy

Eng. Ahmed Salama

Contents

a) Keywords	4
These are the keywords of the C programming language	4
b) Variable Identifiers	6
Allowed characters:	6
Naming Rules:	6
Case Sensitivity:	6
Good Practice:	6
Examples:	6
Valid variable names:	6
Invalid variable names:	7
c) Function Identifiers.....	7
Naming Rules:	7
Length Limit:	7
Reserved Keywords:	7
Scope:.....	8
Linkage:.....	8
Declaration:	8
Definition:	8
Use:.....	8
d) Data Types	8
Basic types:	9
Main types	9
Boolean type	11
Bit-precise integer	11
Derived Data Types:	11
Arrays	12
Pointers	13
Enumerated types:.....	13
Void:	13
e) Functions	13
They have the following components:	13
How to create a function:	14

Parameters and Arguments	15
Return Values	16
f) Statements	17
i. Assignment Statement	17
ii. Declaration Statement.....	18
iii. Return Statement.....	18
iv. Iterative Statement	19
v. Conditional Statements	19
vi. Function Call Statement.....	20
g) Expression	21
Types of Expression in C	21
I. Arithmetic Expression:.....	21
II. Relational Expression.....	22
III. Logical Expression	24
IV. Conditional Expression:.....	26
Lexical analyzer:	28
Defined tokens.....	28
Implementation details.....	30
Code Overview.....	30
Token Structure.....	30
Lexer Class	30
Tokenization Process	31
Main Function	32
The first phase lexical analyzer had the following issues	33
output.....	40
Parser:	46
The following is the grammar rules	46
Implementaiont.....	64
Output sample for the parser:	68

a) Keywords

This report delves into the fundamental building blocks of the C programming language—its keywords. Each keyword, from 'if' to 'typedef', serves a specific function in controlling program flow and data manipulation. Understanding these keywords is crucial for writing effective C code.

These are the keywords of the C programming language:

- `alignas` (C23): Specifies alignment requirements for types or objects.
- `alignof` (C23): Determines alignment requirements of types.
- `auto`: Specifies automatic storage duration for variables.
- `bool` (C23): Represents Boolean data type (true or false).
- `break`: Terminates the current loop or switch statement.
- `case`: Labels a statement within a switch statement.
- `char`: Represents a character data type.
- `const`: Indicates that an object's value cannot be modified.
- `constexpr` (C23): Specifies that an object or function is constant and can be evaluated at compile-time.
- `continue`: Jumps to the next iteration of a loop.
- `default`: Provides a default case for a switch statement.
- `do`: Initiates a do-while loop.
- `double`: Represents a double-precision floating-point data type.
- `else`: Specifies an alternative condition in an if statement.
- `enum`: Defines a set of named integer constants.
- `extern`: Declares a variable or function as defined externally.
- `false` (C23): Represents the Boolean value false.
- `float`: Represents a single-precision floating-point data type.
- `for`: Initiates a for loop.
- `goto`: Transfers control to a labeled statement.
- `if`: Evaluates a condition and executes a statement if true.
- `inline` (C99): Suggests inline expansion of a function.
- `int`: Represents an integer data type.
- `long`: Represents a long integer data type.
- `nullptr` (C23): Represents a null pointer.
- `register`: Specifies that a variable should be stored in a register.
- `restrict` (C99): Specifies that a pointer does not alias other pointers.
- `return`: Exits a function and returns a value.
- `short`: Represents a short integer data type.
- `signed`: Specifies that a data type can represent both positive and negative values.
- `sizeof`: Determines the size of a data type or object.
- `static`: Specifies internal linkage or persists value across function calls.
- `static_assert` (C23): Performs compile-time assertion checking.

- `struct`: Defines a structure type.
- `switch`: Evaluates an expression and executes code based on its value.
- `thread_local` (C23): Specifies that a variable has thread-local storage duration.
- `true` (C23): Represents the Boolean value `true`.
- `typedef`: Creates a new name for an existing data type.
- `typeof` (C23): Returns the type of an expression.
- `typeof_unqual` (C23): Returns the unqualified version of a type.
- `union`: Defines a union type.
- `unsigned`: Represents an unsigned integer data type.
- `void`: Represents an empty data type.
- `volatile`: Specifies that a variable can be modified by external factors.
- `while`: Initiates a while loop.
- `_Alignas` (C11): Specifies alignment requirements for types or objects.
- `_Alignof` (C11): Determines alignment requirements of types.
- `_Atomic` (C11): Provides atomic access to variables.
- `_BitInt` (C23): Represents a `_BitInt` data type (a fixed-size integer type).
- `_Bool` (C99): Represents the Boolean data type (`true` or `false`).
- `_Complex` (C99): Represents the complex floating-point data type.
- `_Decimal128` (C23): Represents a decimal floating-point data type with 128-bit precision.
- `_Decimal32` (C23): Represents a decimal floating-point data type with 32-bit precision.
- `_Decimal64` (C23): Represents a decimal floating-point data type with 64-bit precision.
- `_Generic` (C11): Allows selection of expressions based on the type of arguments.
- `_Imaginary` (C99): Represents the imaginary part of a complex floating-point number.
- `_Noreturn` (C11): Specifies that a function does not return.
- `_Static_assert` (C11): Performs compile-time assertion checking.
- `_Thread_local` (C11): Specifies that a variable has thread-local storage duration.
- `#if`, `#elif`, `#else`, `#endif`: Conditional compilation directives.
- `#ifdef`, `#ifndef`: Conditional compilation based on whether macros are defined.
- `#elifdef` (C23), `#elifndef` (C23): Combination of `#elif` with `#ifdef` or `#ifndef`.
- `#define`, `#undef`: Define and undefine macros.
- `#include`: Includes contents of another file.
- `#embed` (C23): Includes contents of another file directly into the executable.
- `#line`: Specifies the line number and optional file name for diagnostic messages.
- `#error`: Stops compilation with an error message.
- `#warning` (C23): Generates a warning message during compilation.
- `#pragma`: Implementation-defined directive for compiler-specific instructions.
- `defined`: Checks whether a macro is defined.
- `has_include` (C23): Checks if a header file can be included.
- `has_embed` (C23): Checks if a file can be embedded using `#embed`.
- `has_c_attribute` (C23): Checks if a compiler attribute is supported.

- Pragma (C99): Executes a pragma directive

b) Variable Identifiers

Allowed characters:-

- Variable names can contain all letters (Uppercase and Lowercase) , digits and underscores.
- Variable names cannot contain spaces or special characters other than the underscore.

Naming Rules:-

- Variables can only begin with either a letter (Uppercase or Lowercase) or underscore "_" • Variable names must not be reserved words such as int or struct.
- C doesn't have a specific length limitation for variable names so most compilers accept variable names up to 31 characters.

Case Sensitivity:-

- Variable names are case sensitive meaning that for example
- Num and num are 2 different variables.

Good Practice:-

- It's considered good practice to name variables by what they do or make them understandable "Meaningful Names" (eg:- if we want to name a variable to hold the summation of numbers, we can name it s or sum)
- Try using the Camel case (Camel case involves starting each word except the first with a capital letter eg :- mySumOfNums) or Underscore (underscores separate words with underscores eg:- my_sum_of_nums) naming conventions for variable names.

Examples:-

Valid variable names:-

- int
- class_average; ○
- float temperature;

- char
firstName[20];
- int
numberOfStudent
s;
- float
total_salary;

Invalid variable names:-

- int 2much; (Variables can't start with a number)
- int my-Sum; ("-" is a special character we can use)
- float double; (double is a reserved word in C)

○ More about variables in Statements -> Declaration Statement

c) Function Identifiers

In C programming, function identifiers are essential for defining, declaring, and using functions. They follow specific rules and conventions to ensure proper naming, scoping, and linkage within a program. Understanding these constructs is fundamental for writing efficient and maintainable C code.

Naming Rules:-

Function identifiers in C must follow the same rules as other identifiers. They can consist of letters, digits, and underscores, but must start with a letter or underscore. The remaining characters can be letters, digits, or underscores. C is case-sensitive, so uppercase and lowercase letters are considered different. It's good practice to choose descriptive and meaningful names for functions to indicate their purpose or functionality.

Length Limit:-

Function identifiers can be of any length, but only the first 63 characters are significant. This means that identifiers longer than 63 characters may be treated as different identifiers if only the first 63 characters are the same.

Reserved Keywords:-

Function identifiers cannot be the same as reserved keywords in C. Keywords are words that have special meaning in the C language and are used to define the syntax and structure of the program. Examples of keywords in C include int, float, for, while, if, else, etc.

Scope:-

By default, function identifiers in C have file scope, which means they are visible only within the file where they are declared or defined. To make a function identifier visible outside the file, you can use the extern keyword in the declaration.

Linkage:-

Function identifiers in C have external linkage by default, which means they can be accessed from other files if declared with the extern keyword. External linkage allows functions to be shared between different parts of a program.

Declaration:-

Before you can use a function in C, you must declare it. A function declaration tells the compiler about the function's name, return type, and parameters, but does not provide the function's implementation. Function declarations are typically placed in header files.

Definition:-

The definition of a function in C provides the actual implementation of the function. It includes the function's name, return type, parameters, and the code that defines what the function does. A function must be defined before it is called in the program.

Use:-

Function identifiers are used in C to call functions, pass arguments to functions, and return values from functions. To call a function, you simply use its name followed by parentheses containing any arguments the function requires.

d) Data Types

In the C programming language, data types constitute the semantics and characteristics of storage of data elements. They are expressed in the language syntax in form of declarations for memory locations or variables. Data types also determine the types of operations or methods of processing of data elements.

The C language provides basic types, such as integer and real number types. Derived data types such as Array, Pointer, Structure and Union types. Enumerated types which are defined with enum to assign names to integral constants. Void which The type specifier void indicates that no value is available.

Basic types:-

Main types

The C language provides the four basic arithmetic type specifiers char, int, float and double, and the modifiers signed, unsigned, short, and long as following.

- Char
The char datatype is used to represent a single character. It is the smallest addressable unit of the machine that can contain the basic character set. The size of a char is always 8 bits (1 byte). char can be either signed or unsigned, depending on the compiler and platform, but it is often treated as a signed type by default with Format specifier: %c.
- Signed char
the signed char datatype is similar to the char datatype, but it is explicitly specified as signed. It is guaranteed to be able to represent both positive and negative values. The size of a signed char is always 8 bits (1 byte), just like the regular char. The range of values that can be represented by a signed char is typically from -128 to 127. Format specifier is %c or %hhi for numerical output.
- Unsigned char
It is similar to char datatype, but it is explicitly specified as unsigned. It is used to represent only non-negative values, ranging from 0 to 255. The size of an unsigned char is always 8 bits (1 byte), just like the regular char with the same Format specifier.
- Short, short int, signed short, signed short int
These types are all synonymous and refer to the same integer data type. These types represent a short signed integer, typically with a size of 16 bits (2 bytes). The range of values that can be represented by a short is often from -32,768 to 32,767. Format specifiers: %hi or %hd for printf/scanf functions.
- unsigned short, unsigned short int unsigned short and unsigned short int are synonymous and refer to the same unsigned integer data type. These types represent a short unsigned integer, typically with a size of 16 bits (2 bytes). The range of values that can be represented by an unsigned short is from 0 to 65,535.
- int, Signed, signed int
These are all synonymous and refer to the same signed integer data type. These types represent a basic signed integer with a size of at least 16 bits (2 bytes). The range of values that can be represented by an int is often from -32,768 to 32,767. Format specifier: %i or %d for printf/scanf functions.
- unsigned, unsigned int

These are synonymous and refer to the same unsigned integer data type. These types represent a basic unsigned integer with a size of at least 16 bits (2 bytes). The range of values that can be represented by an unsigned or unsigned int is from 0 to 65,535. Format specifier: %u for printf/scanf functions

- long, long int, signed long, signed long int
These are all synonymous and refer to the same signed integer data type. These types represent a signed long integer, typically with a size of at least 32 bits (4 bytes). The range of values that can be represented by a long is often from - 2,147,483,648 to 2,147,483,647.
- unsigned long, unsigned long int
These are synonymous and refer to the same unsigned long integer data type. These types represent an unsigned long integer, typically with a size of at least 32 bits (4 bytes). The range of values that can be represented by an unsigned long is from 0 to 4,294,967,295. Format specifier: %lu for printf/scanf functions
- long long, long long int, signed long long, signed long long int
These are all synonymous and refer to the same signed long long integer data type. These types represent a signed long long integer, typically with a size of at least 64 bits (8 bytes). The range of values that can be represented by a long long is often from - 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Format specifiers: %lli or %lld for printf/scanf functions
- unsigned long long, unsigned long long int are synonymous and refer to the same unsigned long long integer data type. These types represent an unsigned long long integer, typically with a size of at least 64 bits (8 bytes). The range of values that can be represented by an unsigned long long is from 0 to 18,446,744,073,709,551,615. Format specifier: %llu for printf/scanf functions
- Float
float is a datatype used to represent single-precision floating-point numbers. It is typically used to store real numbers (those with decimal points) and has a size of 4 bytes. The float type provides a compromise between precision and storage space. Format specifier: %f for printf/scanf functions
- Double
double is a datatype used to represent double-precision floating-point numbers. It is typically used to store real numbers (those with decimal points) and provides higher precision compared to the float type. The size of a double is typically 8 bytes. Format specifier: %lf for printf/scanf functions
- long double long double is a datatype used to represent extended-precision floating-point numbers. It provides a higher precision than both float and double. The size of a long double is implementation-dependent and may vary across different systems and compilers. Format specifier: %Lf for printf/scanf functions

Boolean type

In C programming, there is no dedicated boolean data type. Instead, boolean functionality is often implemented using integers. A common convention is to use integers, where 0 represents false, and any non-zero value represents true.

To enhance readability and convenience, some programmers use macros like `#define true 1` and `#define false 0`. This provides a more expressive way to work with boolean values.

Additionally, C provides the `_Bool` type, which functions similarly to a normal integer type but is commonly used to represent boolean values. The `<stdbool.h>` header can be included to define the `bool` alias for `_Bool` and macros for `true` and `false`.

Bit-precise integer

In C programming, bit-precise integer types allow for precise control over the number of bits used to represent integer values. This is particularly useful in situations where the size of the integer matters, such as in embedded systems or when dealing with hardware-level programming. C, by default, provides basic integer types like `int` and `long`, but their sizes are implementation-dependent.

For bit-precise control, C does not have a native syntax for specifying a fixed number of bits directly for integer types. However, developers often use bit-field structures and bitwise operations to achieve similar effects.

Derived Data Types:-

derived data types are types that are created using the basic data types (`int`, `float`, `char`, etc.) to define more complex data structures. There are several types of derived data types in C including arrays, structures, pointers and Union.

Union

A union type is a special construct that permits access to the same memory block by using a choice of differing type descriptions. For example, a union of data types may be declared to permit reading the same data either as an integer, a float, or any other user declared type:

```

union {
    int i; float f; struct {
        unsigned int u;
        double d;
    } s; }
u;

```

The total size of `u` is the size of `u.s` – which happens to be the sum of the sizes of `u.s.u` and `u.s.d` – since `s` is larger than both `i` and `f`. When assigning `u.i` of `u.f` something to `u.i` some parts may be preserved if `i` is smaller than `u.f`.

Reading from a union member is not the same as casting since the value of the member is not converted, but merely read.

Arrays

An array is a derived data type in C that allows you to group a collection of elements of the same data type under a single name. Arrays are used to store multiple values of the same data type in contiguous memory locations. The general syntax for declaring an array in C is:
`data_type array_name[array_size];`

Here, `data_type` specifies the type of elements that the array will hold, `array_name` is the name of the array, and `array_size` is the number of elements in the array. For example `int numbers[5];`

Structure

Structure is a derived data type that allows you to group different data types together under a single name. A structure is a way to define a record that can contain various types of data members, each with its own data type. The general syntax for declaring a structure in C is as follows:

```

struct
    structure_name
    {
        data_type
        member1;
        data_type
        member2;
        // ... (more members)};

```

Pointers

Pointers are a powerful feature that allows you to work with memory addresses directly. A pointer is a variable that stores the memory address of another variable. The data type of a pointer is based on the type of variable it points to. Here's the general syntax for declaring a pointer: `data_type *pointer_name;` .

Enumerated types:-

Enumeration or Enum in C is a special kind of data type defined by the user. It consists of constant integrals or integers that are given names by a user. The use of enum in C to name the integer values makes the entire program easy to learn, understand, and maintain by the same or even different programmer.

An enum is defined by using the 'enum' keyword in C, and the use of a comma separates the constants within. The basic syntax of defining an enum is: `enum enum_name{int_const1, int_const2, int_const3, int_constN};`

Void:-

Void data type is a special type that indicates the absence of a specific data type. It is often used as a placeholder to signify that a function does not return any value or that a pointer does not have a specific type.

When used as the return type of a function, void indicates that the function doesn't return any value to the calling code. For example, a function with void return type: `void printMessage() { printf("Hello, World!\n");}` void can be used as a pointer type, representing a generic or untyped pointer. For instance, in the context of dynamically allocated memory, a void pointer can be used to point to memory of any data type: `void* allocateMemory(size_t size) {return malloc(size);}` .

e) Functions

Functions in (C) are blocks of code that perform a specific task. They allow for modular programming where complex tasks are divided into smaller, manageable units.

They have the following components:-

- Function name: A unique identifier for the function.
- Return Type: Specifies the type of value of the function returns. It can be (void) if the

function doesn't return any value.

- Arguments: Inputs to the function that are passed when the function is called
- Function Body: The block of the code enclosed within curly braces `{}` that defines what the function does.
- Function Declaration: A prototype that declares the function's name, return type and parameters. It appears before the function's first use in the code.
- Function Definition: The actual implementation of the function.

How to create a function:-

- To create a function, we need to specify the name of the function followed by parentheses () and curly brackets {}:

Syntax

```
void myFunction() {  
    // code to be executed  
}
```

- To Call a Function, write the function's name followed by two parentheses () and a semicolon ;

Example

Inside `main`, call `myFunction()` :

```
// Create a function
void myFunction() {
    printf("I just got executed!");
}

int main() {
    myFunction(); // call the function
    return 0;
}

// Outputs "I just got executed!"
```

- Note: we can call functions multiple times.

Example

```
void myFunction() {
    printf("I just got executed!");
}

int main() {
    myFunction();
    myFunction();
    myFunction();
    return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

- Now example on some of the components:

let's take an

Parameters and Arguments:

Syntax

```
returnType functionName(parameter1, parameter2, parameter3)
{
    // code to be executed
}
```

- This Function takes a string of characters with name as a parameter. When function is called, we pass along a name.

Example

```
void myFunction(char name[]) {
    printf("Hello %s\n", name);
}

int main() {
    myFunction("Liam");
    myFunction("Jenny");
    myFunction("Anja");
    return 0;
}

// Hello Liam
// Hello Jenny
// Hello Anja
```

Return Values:

Example

```
int myFunction(int x) {  
    return 5 + x;  
}  
  
int main() {  
    printf("Result is: %d", myFunction(3));  
    return 0;  
}  
  
// Outputs 8 (5 + 3)
```

- This example returns the sum of a function with two parameters.

f) Statements

i. Assignment Statement:

- An assignment statement is fundamental in C programming, used to assign a value to a variable.
- Syntax: variable = value;
- Example: x = 10;
- Here, x is a variable, and 10 is the value being assigned to it.
- The assignment operator = is used to assign the value on the right side to the variable on the left side.
- Variables must be declared before they can be assigned values. For example:
int x; // Declaration statement x = 10; // Assignment statement
- Assignment statements can involve various types of expressions on both sides of the assignment operator. For instance:
int a, b, result; a = 5; b = 3; result = a + b; // Expression on the right side
- The value assigned to a variable can be a constant, a variable, or the result of an expression.
- Assignment statements are crucial for storing and manipulating data within a program, allowing variables to hold different values throughout program execution.

ii. Declaration Statement:

- This statement is used to declare variables or functions before they are used in the program.
- Syntax: type identifier;
- Example: `int x;`
- In this example, `int` is the data type, and `x` is the variable name.
- Declaration statements inform the compiler about the type and name of the variable.
- It allocates memory for the variable if it's a data type like `int`, `char`, `float`, etc.
- Declarations may also include initialization, where a value is assigned to the variable at the time of declaration. For example: `int x = 10;`
- Multiple variables of the same type can be declared in a single declaration statement, separated by commas. For example: `int a, b, c;`

Declaration statements are crucial for ensuring that variables are properly defined before they are used in the program, preventing compilation errors and ensuring proper memory allocation.

iii. Return Statement:

- The return statement is used to exit a function and return a value to the caller.
- Syntax: `return expression;`
- Example: `return x + y;`
- When a return statement is encountered in a function, the function stops executing and returns the value of the expression to the caller.
- If the function has a return type of `void`, the return statement can be used without an expression to simply exit the function.
- The expression can be a constant, variable, or more complex expression.
- The return statement is optional in functions with a return type of `void`, but it's mandatory for functions with non-void return types.
- It's important to note that only one return statement is executed in a function, and once it's executed, the control returns to the caller.

Return statements are fundamental in functions as they allow the function to communicate its result back to the calling code. They are essential for building modular and reusable code.

iv. Iterative Statement:

- Iterative statements, also known as loops, allow a block of code to be executed repeatedly as long as a specified condition is true.
- There are three main types of iterative statements in C: for, while, and do-while.
- Each type of loop has its own syntax and use cases, but they all serve the same purpose of repetition.
- for loop: It consists of an initialization expression, a loop condition, and an update expression. For example:

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

- while loop: It repeatedly executes a target statement as long as the given condition is true. For example:

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

- do-while loop: It's similar to the while loop but guarantees that the code block is executed at least once before checking the loop condition.

For example:

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

- Loops provide a powerful mechanism for iterating over data structures, performing repetitive tasks, and controlling program flow.
- It's important to ensure that the loop condition is properly defined to prevent infinite loops.

Iterative statements are essential for implementing repetitive tasks efficiently in C programs. They provide a structured way to handle tasks that require repeated execution.

v. Conditional Statements:

- Conditional statements allow the execution of different blocks of code based on certain conditions.
- The main conditional statements in C are if, else if, else, and switch.
- if statement: It executes a block of code if a specified condition is true.

```
if (condition) { // code to execute if the condition is true }
```
- else if statement: It allows for multiple conditions to be checked one after another if the previous conditions were not met. For example:

```
if (condition1) { // code to execute if condition1 is true } else if (condition2) { // code to execute if condition2 is true } else { // code to execute if none of the conditions are true }
```

- else statement: It executes a block of code if the preceding if or else if statements were not true. For example:

```
if (condition) { // code to execute if the condition is true } else { // code to execute if the condition is false }
```

- switch statement: It allows the program to evaluate an expression and execute code blocks based on matching cases. For example:

```
switch (expression) { case value1: // code to execute if expression equals value1 break; case value2: // code to execute if expression equals value2 break; default: // code to execute if none of the cases match break; }
```

- Conditional statements provide the ability to make decisions and control the flow of the program based on various conditions.
- They are fundamental for implementing logic that responds to different scenarios within a program.

Conditional statements are crucial for implementing branching logic in C programs, allowing for dynamic and flexible execution paths.

vi. Function Call Statement:

- Function call statements are used to invoke functions in C, allowing the execution of the code inside the function's body.
- Syntax: `function_name(arguments);`
- Example: `printf("Hello, world!\n");`
- In the example, `printf` is the function name, and `"Hello, world!\n"` is the argument passed to the function.
- Function calls can include zero or more arguments, depending on the function's definition.
- The function call statement causes the program execution to transfer to the function being called.
- Arguments are expressions or values passed to the function, which can be used within the function's body.
- The function may return a value to the caller, which can be used in further computations or assignments.
- Functions allow for modularization and reuse of code, making programs more organized and easier to maintain.

Function call statements are fundamental in C programming for executing modularized code and performing various tasks within a program. They facilitate code reuse and organization by allowing the encapsulation of functionality into callable units.

g) Expression

Expression C is a combination of symbols, numbers, and/or text that produces a particular result. 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.

Types of Expression in C:

- 1- Arithmetic
- 2- Relational
- 3- Logical
- 4- Conditional

I. Arithmetic Expression:-

It performs computations on the int, float, double type values.

They can be performed in a single line of code or multiple lines combined with arithmetic operations such as addition, subtraction, multiplication, and division.

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.
$6 * 2 / (2 + 0 + 6) + 8 * (8 / 4)$	2 is divided by 3, giving a value 0
$6 * 2 / 8 + 8 * (8 / 4)$	2 is added to 6, giving a value 8.
$6 * 2 / 8 + 8 * 2$	8 is divided by 4, giving a value 2.
$12 / 8 + 8 * 2$	6 is multiplied by 2, giving a value 12.
$1 + 8 * 2$	12 is divided by 8, giving a value 1.
$1 + 16$	8 is multiplied by 2, giving a value 16.
17	1 is added to 16, giving a value 17.

II. Relational Expression:

Relational expressions use comparison operator such as '>' (greater than) and '<' (less than) to compare two operands. The result of the comparison is a boolean value i.e. 0(false) or non-zero (true).

Example:

Relational Expression	Description
$x \% 2 == 0$	The given condition checks whether the x is an even number or not. This relational expression shows the result as the value 1 if x is an even number otherwise as the value 0
$a != b$	This relational expression is used to check if a is not equal to b and it results in 1 if a is not equal to b otherwise 0.
$a + b == x + y$	It is used to check if this particular expression "a+b" is equal to the expression "x+y"
$a >= 9$	It is used to check if the value of a is greater than or equal to 9.

Code
Example:

○

```
#include <stdio.h>

int main()
{
    int x = 5, y = 10;
    if (x == y) {
        printf("x is equal to y\n");
    } else {
        printf("x is not equal to y\n");
    }
    return 0;
}
```

○

Output

```
x is not equal to y
```


III. Logical Expression:

- Logical expressions in C are a powerful tool for controlling the logic of the flow of the program. They are made by combining as many relational expressions as the programmer wants.

Example:

Logical Expressions	Description
<code>(x > 4) && (x < 6)</code>	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.
<code>x > 10 y < 11</code>	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.
<code>! (x > 10) && (y == 2)</code>	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

Code
Example:



```
#include <stdio.h>
int main()
{
    int x = 10;
    int y = 2;
    if ( (x > 10) || (y < 5) )
    {
        printf("Condition is true");
    }
    else
    {
        printf("Condition is false");
    }
    return 0;
}
```

Output

```
Condition is true
```

IV. Conditional Expression:-

The conditional expression consists of three operands. It returns 1 if the condition is true otherwise 0.

Example:



```
#include <stdio.h>
#include <string.h>
int main()
{
    int age = 29;
    char status;
    status = (age>22) ? 'M': 'U';
    if(status == 'M')
        printf("Married");
    else
        printf("Unmarried");
    return 0; }
```

Lexical analyzer:

Defined tokens

```
"INT_KEYWORD",
"FLOAT_KEYWORD",
"CHAR_KEYWORD",
"VOID_KEYWORD",
"DOUBLE_KEYWORD",
"LONG_KEYWORD",
"SHORT_KEYWORD",
"UNSIGNED_KEYWORD",
"SIGNED_KEYWORD",
"BOOL_KEYWORD",
"COMPLEX_KEYWORD",
"IMAGINARY_KEYWORD",
"ALIGNAS_KEYWORD",
"ALIGNOF_KEYWORD",
"ASM_KEYWORD",
"AUTO_KEYWORD",
"BREAK_KEYWORD",
"CASE_KEYWORD",
"CONST_KEYWORD",
"CONTINUE_KEYWORD",
"DEFAULT_KEYWORD",
"DO_KEYWORD",
"ELSE_KEYWORD",
"ENUM_KEYWORD",
"EXTERN_KEYWORD",
"FOR_KEYWORD",
"GOTO_KEYWORD",
"IF_KEYWORD",
"REGISTER_KEYWORD",
"RETURN_KEYWORD",
"SIZEOF_KEYWORD",
"STATIC_KEYWORD",
"STRUCT_KEYWORD",
"SWITCH_KEYWORD",
"TYEDEF_KEYWORD",
"UNION_KEYWORD",
"VARIABLE_KEYWORD",
"WHILE_KEYWORD",
"ALIGNAS_KEYWORD",
"ALIGNOF_KEYWORD",
"ATOMIC_KEYWORD",
"GENERIC_KEYWORD",
"NORETURN_KEYWORD",
"STATIC_ASSERT_KEYWORD",
"THREAD_LOCAL_KEYWORD",

"IDENTIFIER",
"FLOATING_POINT_LITERAL",
"INTEGER_LITERAL",
"CHAR_LITERAL",
"STRING_LITERAL",

"ILLEGAL_ASSIGNMENT",
```

"MINUS_ASSIGNMENT",
"MULTIPLY_ASSIGNMENT",
"DIVIDE_ASSIGNMENT",
"MODULUS_ASSIGNMENT",
"BITWISE_AND_ASSIGNMENT",
"BITWISE_OR_ASSIGNMENT",
"BITWISE_XOR_ASSIGNMENT",
"SHIFT_LEFT_ASSIGNMENT",
"SHIFT_RIGHT_ASSIGNMENT",
"SHIFT_LEFT",
"SHIFT_RIGHT",
"EQUAL",
"NOT_EQUAL",
"LESS_THAN_OR_EQUAL",
"GREATER_THAN_OR_EQUAL",
"LOGICAL_AND",
"LOGICAL_OR",
"INCREMENT",
"DECREMENT",
"PLUS",
"MINUS",
"MULTIPLY",
"DIVIDE",
"MODULUS",
"BITWISE_AND",
"BITWISE_OR",
"BITWISE_XOR",
"LOGICAL_NOT",
"BITWISE_NOT",
"LESS_THAN",
"GREATER_THAN",
"CONDITIONAL",
"ASSIGNMENT",
"COLON",
"COMMA",
"DOT",
"OFIEN_fIAREN",
"CLOSE_fIAREN",
"OFIEN_BRACE",
"CLOSE_BRACE",
"OFIEN_SQUARE_BRACKET",
"CLOSE_SQUARE_BRACKET",
"HASH",
"AT",
"DOLLAR",
"SEMI_COLON"

Implementation details

Code Overview

The program begins by including the necessary libraries. `iostream` is used for input/output operations, `fstream` for file handling, `regex` for regular expressions, `string` for string manipulation, and `vector` for dynamic arrays.

```
✓ #include <iostream>
  #include <fstream>
  #include <regex>
  #include <string>
  #include <vector>

  using namespace std;
```

Token Structure

The `Token` structure is used to hold the type and value of a token. The `type` is a string that represents the category of the token (e.g., "KEYWORD", "IDENTIFIER", etc.), and the `value` is the actual text of the token.

```
✓ struct Token {
  |     string type;
  |     string value;
  | };
```

Lexer Class

The `Lexer` class contains the `tokenize` function, which takes a string of code as input and returns a vector of `Token` objects.

```
✓ class Lexer {
  | public:
  |     // Function to tokenize the input C code
  |     vector<Token> tokenize(string code) { ... }
  | };
```

Tokenization Process

The `tokenize` function first removes all comments from the code. Then, it uses regular expressions to match different types of tokens, including keywords, identifiers, literals, operators, punctuators, and special characters. For each match, a new `Token` object is created and added to the `tokens` vector.

```
vector<Token> tokenize(string code) {
    vector<Token> tokens;

    // Regular expression to match C-style comments
    regex comment_regex("(//.*|/\\*.*?\\*/)");

    // Remove all comments from the code
    code = regex_replace(code, comment_regex, "");

    // Regular expressions for various token types
    regex keyword("\\b(?:int|float|char|void|double|long|short|unsigned|signed|_Bool|_Complex|_Imaginary|alignas|alignof|asm|a
    regex identifier("\\b[a-zA-Z_][a-zA-Z0-9_]*\\b");
    regex literal("\\b(\\d*\\.\\d+|\\d+\\.\\d*|\\d+)([eE][+-]?\\d+)?\\b'|\"(\\\\\\\\.|[^\"])*\"\\b");
    regex operators("\\+\\+?|\\-\\-?|\\*=?|\\/=?|\\&\\&?|\\|=|<=?|>=?|!=|\\^=?|\\|\\|?|\\%|~|\\|?|:");
    regex punctuators("\\[\\[\\[\\{\\}\\}\\];;:;#%&|^~!=<>+\\-\\/\\\\\\.\\.");
    regex specials("\\#|\\$");

    vector<regex> regexes = { keyword, identifier, literal, operators, punctuators, specials };
    vector<string> types = { "KEYWORD", "IDENTIFIER", "LITERAL", "OPERATOR", "PUNCTUATOR", "SPECIAL_CHARACTER" };
}
```

The function uses a `while` loop to iterate over the code. For each character in the code, it checks if it matches any of the regular expressions. If a match is found, a new `Token` is created with the corresponding type and value, and the character is removed from the code. If no match is found, the character is skipped.

```
string::const_iterator searchStart(code.cbegin());
while (searchStart != code.cend())
{
    bool found = false;
    for (int i = 0; i < regexes.size(); ++i)
    {
        smatch match;
        if (regex_search(searchStart, code.cend(), match, regexes[i]) && match.prefix().length() == 0)
        {
            tokens.push_back({ types[i], match.str() });
            searchStart += match.length();
            found = true;
            break; // we found a match so no need to check other patterns
        }
    }

    // if no match we skip one character and continue to search
    if (!found) ++searchStart;
}

return tokens;
```

This process continues until all characters in the code have been checked. The function then returns the vector of `Token` objects.

Main Function

The `main` function is the entry point of the program. It starts by reading the C code from a file. If the file cannot be opened, an error message is printed and the program exits. The used file is normal text file with C code written inside.

```
main() {  
    // Read input C code from file  
    ifstream inputFile("C:\\hassan\\cess\\semester 6\\design of compilers\\project\\  
    if (!inputFile.is_open()) {  
        cerr << "Unable to open input file." << endl;  
        return 1;  
    }  
}
```

Next, a `Lexer` object is created. The `tokenize` function of the `Lexer` object is called with the read code as an argument. The returned vector of `Token` objects is then outputted.

```
string code((istreambuf_iterator<char>(inputFile)), (istreambuf_iterator<char>()));  
  
// Create a Lexer object  
Lexer lexer;  
  
// Tokenize the code  
vector<Token> tokens = lexer.tokenize(code);  
  
// Output the tokens  
cout << "Tokens:" << endl;
```

The function uses a `while` loop to iterate over the vector of `Token` objects. For each `Token`, it prints the type and value and removes the `Token` from the vector. This continues until all `Token` objects have been printed and removed.

```
while (!tokens.empty()) {  
    Token token = tokens.front();  
    cout << token.type << ": " << token.value << endl;  
    tokens.erase(tokens.begin());  
}  
  
return 0;
```

The function then returns 0 to indicate successful execution of the program.

The first phase lexical analyzer had the following issues:

- 1- The literals were not separated by type.
- 2- Erroneous tokens were not handled.

In the new code for the lexer these problems were fixed.
each keyword, operator and literal has its own regex

```
✓ #include <iostream>
#include <fstream>
#include <regex>
#include <string>
#include <vector>

using namespace std;

// Token structure to hold token type and value

✓ struct Token {
    string type;
    string value;
};

✓ class Lexer {
public:
    // Function to tokenize the input C code
    ✓ vector<Token> tokenize(string code) {
        vector<Token> tokens;

        // Regular expression to match C-style comments
        regex comment_regex("(//.*|/\\*.*?\\*/)");

        // Remove all comments from the code
        code = regex_replace(code, comment_regex, "");
        // Regular expressions for various token types
        regex int_keyword("\\bint\\b");
        regex float_keyword("\\bfloat\\b");
        regex char_keyword("\\bchar\\b");
        regex void_keyword("\\bvoid\\b");
        regex double_keyword("\\bdouble\\b");
        regex long_keyword("\\blong\\b");
        regex short_keyword("\\bshort\\b");
        regex unsigned_keyword("\\bunsigned\\b");
        regex signed_keyword("\\bsigned\\b");
        regex bool_keyword("\\bBool\\b");
        regex complex_keyword("\\b_Complex\\b");
        regex imaginary_keyword("\\b_Imaginary\\b");
        regex alignas_keyword("\\balignas\\b");
        regex alignof_keyword("\\balignof\\b");
        regex asm_keyword("\\bas\\b");
```

```
regex auto_keyword("\\bauto\\b");
regex break_keyword("\\bbreak\\b");
regex case_keyword("\\bcase\\b");
regex const_keyword("\\bconst\\b");
regex continue_keyword("\\bcontinue\\b");
regex default_keyword("\\bdefault\\b");
regex do_keyword("\\bdo\\b");
regex else_keyword("\\belse\\b");
regex enum_keyword("\\benum\\b");
regex extern_keyword("\\bextern\\b");
regex for_keyword("\\bfor\\b");
regex goto_keyword("\\bgoto\\b");
regex if_keyword("\\bif\\b");
regex register_keyword("\\bregister\\b");
regex return_keyword("\\breturn\\b");
regex sizeof_keyword("\\bsizeof\\b");
regex static_keyword("\\bstatic\\b");
regex struct_keyword("\\bstruct\\b");
regex switch_keyword("\\bswitch\\b");
regex typedef_keyword("\\btypedef\\b");
regex union_keyword("\\bunion\\b");
regex volatile_keyword("\\bvolatile\\b");
regex while_keyword("\\bwhile\\b");
regex alignas_keyword("\\b_Alignas\\b");
regex alignof_keyword("\\b_Alignof\\b");
regex atomic_keyword("\\b_Atomic\\b");
regex generic_keyword("\\b_Generic\\b");
regex noreturn_keyword("\\b_Noreturn\\b");
regex static_assert_keyword("\\b_Static_assert\\b");
regex thread_local_keyword("\\b_Thread_local\\b");
```

```
regex identifier("\\b[a-zA-Z_][a-zA-Z0-9_]*\\b");
regex floating_point_literal("\\b\\d+\\.\\d+\\b");
regex integer_literal("\\b\\d+\\b");
regex char_literal("'([^'\\\\\\\\]|\\\\\\\\.)*'");
regex string_literal("\"([^\"]\\\\\\\\|\\\\\\\\.)*\"");
regex plus_assignment("\\+=");
regex minus_assignment("-=");
regex multiply_assignment("\\*=");
regex divide_assignment("/=");
regex modulus_assignment("%=");
regex bitwise_and_assignment("&=");
regex bitwise_or_assignment("\\|=");
regex bitwise_xor_assignment("\\^=");
regex shift_left_assignment("<<=");
regex shift_right_assignment(">>=");
regex shift_left("<<");
regex shift_right(">>");
regex equal("==");
regex not_equal("!=");
regex less_than_or_equal("<=");
regex greater_than_or_equal(">=");
regex logical_and("\\&\\&");
regex logical_or("\\|\\|");
regex increment("\\++");
regex decrement("--");
regex plus("\\+");
regex minus("-");
regex multiply("\\*");
regex divide("/");
regex modulus("%");
regex bitwise_and("&");
regex bitwise_or("\\|");
regex bitwise_xor("\\^");
regex logical_not("!");
regex bitwise_not("~");
regex less_than("<");
regex greater_than(">");
regex conditional("\\?");
regex assignment("=");
regex colon(":");
regex comma(",");
```

```

regex dot("\\.");
regex open_paren("\\(");
regex close_paren("\\)");
regex open_brace("\\{");
regex close_brace("\\}");
regex open_square_bracket("\\[");
regex close_square_bracket("\\]");
regex hash("#");
regex at("@");
regex dollar("\\$");
regex semicolon(";");

```

```

vector<regex> regexes = {
    int_keyword, float_keyword, char_keyword, void_keyword, double_keyword, long_keyword, short_keyword, unsigned_keyword, signed_keyword, bool_keyword, complex_keyword, imaginary_keyword, balignas_keyword,
    balignof_keyword, asm_keyword, auto_keyword, break_keyword, case_keyword, const_keyword, continue_keyword, default_keyword, do_keyword, else_keyword, enum_keyword, extern_keyword, for_keyword, goto_keyword,
    if_keyword, register_keyword, return_keyword, sizeof_keyword, static_keyword, struct_keyword, switch_keyword, typedef_keyword, union_keyword, volatile_keyword, while_keyword, alignas_keyword, alignof_keyword,
    atomic_keyword, generic_keyword, noreturn_keyword, static_assert_keyword, thread_local_keyword,
    identifier, floating_point_literal, integer_literal, char_literal, string_literal,
    plus_assignment, minus_assignment, multiply_assignment, divide_assignment, modulus_assignment, bitwise_and_assignment, bitwise_or_assignment, bitwise_xor_assignment, shift_left_assignment, shift_right_assignment,
    shift_left, shift_right, equal, not_equal, less_than_or_equal, greater_than_or_equal, logical_and, logical_or, increment, decrement, plus, minus, multiply, divide, modulus, bitwise_and, bitwise_or, bitwise_xor,
    logical_not, bitwise_not, less_than, greater_than, conditional, assignment, colon, comma, dot, open_paren, close_paren, open_brace, close_brace, open_square_bracket, close_square_bracket, hash, at, dollar, semicolon };

```

```
vector<string> types = {  
    "INT_KEYWORD",  
    "FLOAT_KEYWORD",  
    "CHAR_KEYWORD",  
    "VOID_KEYWORD",  
    "DOUBLE_KEYWORD",  
    "LONG_KEYWORD",  
    "SHORT_KEYWORD",  
    "UNSIGNED_KEYWORD",  
    "SIGNED_KEYWORD",  
    "BOOL_KEYWORD",  
    "COMPLEX_KEYWORD",  
    "IMAGINARY_KEYWORD",  
    "ALIGNAS_KEYWORD",  
    "ALIGNOF_KEYWORD",  
    "ASM_KEYWORD",  
    "AUTO_KEYWORD",  
    "BREAK_KEYWORD",  
    "CASE_KEYWORD",  
    "CONST_KEYWORD",  
    "CONTINUE_KEYWORD",  
    "DEFAULT_KEYWORD",  
    "DO_KEYWORD",  
    "ELSE_KEYWORD",  
    "ENUM_KEYWORD",  
    "EXTERN_KEYWORD",  
    "FOR_KEYWORD",  
    "GOTO_KEYWORD",  
    "IF_KEYWORD",  
    "REGISTER_KEYWORD",  
    "RETURN_KEYWORD",  
    "SIZEOF_KEYWORD",  
    "STATIC_KEYWORD",  
    "STRUCT_KEYWORD",  
    "SWITCH_KEYWORD",  
    "TYPEDEF_KEYWORD",  
    "UNION_KEYWORD",  
    "VOLATILE_KEYWORD",  
    "WHILE_KEYWORD",  
    "ALIGNAS_KEYWORD",  
    "ALIGNOF_KEYWORD",  
    "ATOMIC_KEYWORD",  
    "GENERIC_KEYWORD",  
    "NORETURN_KEYWORD",  
    "STATIC_ASSERT_KEYWORD",  
    "THREAD_LOCAL_KEYWORD",  
}
```

```
    "IDENTIFIER",  
    "FLOATING_POINT_LITERAL",  
    "INTEGER_LITERAL",  
    "CHAR_LITERAL",  
    "STRING_LITERAL",  
}
```



```
"PLUS_ASSIGNMENT",
"MINUS_ASSIGNMENT",
"MULTIPLY_ASSIGNMENT",
"DIVIDE_ASSIGNMENT",
"MODULUS_ASSIGNMENT",
"BITWISE_AND_ASSIGNMENT",
"BITWISE_OR_ASSIGNMENT",
"BITWISE_XOR_ASSIGNMENT",
"SHIFT_LEFT_ASSIGNMENT",
"SHIFT_RIGHT_ASSIGNMENT",
"SHIFT_LEFT",
"SHIFT_RIGHT",
"EQUAL",
"NOT_EQUAL",
"LESS_THAN_OR_EQUAL",
"GREATER_THAN_OR_EQUAL",
"LOGICAL_AND",
"LOGICAL_OR",
"INCREMENT",
"DECREMENT",
"PLUS",
"MINUS",
"MULTIPLY",
"DIVIDE",
"MODULUS",
"BITWISE_AND",
"BITWISE_OR",
"BITWISE_XOR",
"LOGICAL_NOT",
"BITWISE_NOT",
"LESS_THAN",
"GREATER_THAN",
"CONDITIONAL",
"ASSIGNMENT",
"COLON",
"COMMA",
"DOT",
"OPEN_PAREN",
"CLOSE_PAREN",
"OPEN_BRACE",
"CLOSE_BRACE",
"OPEN_SQUARE_BRACKET",
"CLOSE_SQUARE_BRACKET",
"HASH",
"AT",
"DOLLAR",
"SEMI_COLON" };
```

```

string::const_iterator searchStart(code.cbegin());
while (searchStart != code.cend())
{
    // Skip over spaces
    while (searchStart != code.cend() && isspace(*searchStart)) {
        ++searchStart;
    }

    // Check if searchStart reached the end of code
    if (searchStart == code.cend()) {
        break; // Exit the loop if end of code is reached
    }

    bool found = false;
    for (int i = 0; i < regexes.size(); ++i)
    {
        smatch match;
        if (regex_search(searchStart, code.cend(), match, regexes[i]) && match.prefix().length() == 0)
        {
            tokens.push_back({ types[i], match.str() });
            searchStart += match.length();
            found = true;
            break; // we found a match so no need to check other patterns
        }
    }
    if (!found) {
        string invalidToken = "";
        invalidToken += *searchStart;
        tokens.push_back({ "ERROR", invalidToken });
        ++searchStart;
    }
}

return tokens;
}
};

```

output:

```
Tokens:
IF_KEYWORD: if
OPEN_PAREN: (
INTEGER_LITERAL: 1
GREATER_THAN_OR_EQUAL: >=
INTEGER_LITERAL: 0
CLOSE_PAREN: )
INT_KEYWORD: int
IDENTIFIER: number
ASSIGNMENT: =
ERROR: 1
ERROR: 0
IDENTIFIER: r
SEMI_COLON: ;
FLOAT_KEYWORD: float
IDENTIFIER: decimal
ASSIGNMENT: =
FLOATING_POINT_LITERAL: 0.5
SEMI_COLON: ;
CHAR_KEYWORD: char
IDENTIFIER: character
ASSIGNMENT: =
CHAR_LITERAL: 'a'
SEMI_COLON: ;
CHAR_KEYWORD: char
MULTIPLY: *
IDENTIFIER: string
ASSIGNMENT: =
STRING_LITERAL: "Hello, world!"
SEMI_COLON: ;
LONG_KEYWORD: long
LONG_KEYWORD: long
IDENTIFIER: bigNumber
ASSIGNMENT: =
INTEGER_LITERAL: 1234567890123456789
SEMI_COLON: ;
UNSIGNED_KEYWORD: unsigned
INT_KEYWORD: int
IDENTIFIER: unsignedNumber
ASSIGNMENT: =
INTEGER_LITERAL: 500
SEMI_COLON: ;
DOUBLE_KEYWORD: double
IDENTIFIER: doubleNumber
ASSIGNMENT: =
FLOATING_POINT_LITERAL: 0.123456789
SEMI_COLON: ;
BOOL_KEYWORD: _Bool
IDENTIFIER: boolean
ASSIGNMENT: =
INTEGER_LITERAL: 1
```



```
SEMI_COLON: ;
VOID_KEYWORD: void
IDENTIFIER: printMessage
OPEN_PAREN: (
CONST_KEYWORD: const
CHAR_KEYWORD: char
MULTIPLY: *
IDENTIFIER: message
CLOSE_PAREN: )
SEMI_COLON: ;
ENUM_KEYWORD: enum
IDENTIFIER: Days
OPEN_BRACE: {
IDENTIFIER: SUN
COMMA: ,
IDENTIFIER: MON
COMMA: ,
IDENTIFIER: TUE
COMMA: ,
IDENTIFIER: WED
COMMA: ,
IDENTIFIER: THU
COMMA: ,
IDENTIFIER: FRI
COMMA: ,
IDENTIFIER: SAT
CLOSE_BRACE: }
SEMI_COLON: ;
STRUCT_KEYWORD: struct
IDENTIFIER: Point
OPEN_BRACE: {
INT_KEYWORD: int
IDENTIFIER: x
SEMI_COLON: ;
INT_KEYWORD: int
IDENTIFIER: y
SEMI_COLON: ;
CLOSE_BRACE: }
SEMI_COLON: ;
INT_KEYWORD: int
IDENTIFIER: main
OPEN_PAREN: (
CLOSE_PAREN: )
OPEN_BRACE: {
IF_KEYWORD: if
OPEN_PAREN: (
IDENTIFIER: number
GREATER_THAN: >
```

```
INTEGER_LITERAL: 5
CLOSE_PAREN: )
OPEN_BRACE: {
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "Number is greater than 5\n"
CLOSE_PAREN: )
SEMI_COLON: ;
CLOSE_BRACE: }
ELSE_KEYWORD: else
OPEN_BRACE: {
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "Number is not greater than 5\n"
CLOSE_PAREN: )
SEMI_COLON: ;
CLOSE_BRACE: }
INT_KEYWORD: int
IDENTIFIER: k
ASSIGNMENT: =
INTEGER_LITERAL: 0
SEMI_COLON: ;
IF_KEYWORD: if
OPEN_PAREN: (
IDENTIFIER: k
EQUAL: ==
INTEGER_LITERAL: 0
CLOSE_PAREN: )
IDENTIFIER: k
PLUS_ASSIGNMENT: +=
INTEGER_LITERAL: 1
SEMI_COLON: ;
FOR_KEYWORD: for
OPEN_PAREN: (
INT_KEYWORD: int
IDENTIFIER: i
ASSIGNMENT: =
INTEGER_LITERAL: 0
SEMI_COLON: ;
IDENTIFIER: i
LESS_THAN: <
IDENTIFIER: MAX
SEMI_COLON: ;
IDENTIFIER: i
INCREMENT: ++
CLOSE_PAREN: )
```

```
OPEN_BRACE: {
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "%d\n"
COMMA: ,
IDENTIFIER: i
CLOSE_PAREN: )
SEMI_COLON: ;
CLOSE_BRACE: }
INT_KEYWORD: int
IDENTIFIER: j
ASSIGNMENT: =
INTEGER_LITERAL: 0
SEMI_COLON: ;
WHILE_KEYWORD: while
OPEN_PAREN: (
IDENTIFIER: j
LESS_THAN: <
IDENTIFIER: MAX
CLOSE_PAREN: )
OPEN_BRACE: {
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "%d\n"
COMMA: ,
IDENTIFIER: j
CLOSE_PAREN: )
SEMI_COLON: ;
IDENTIFIER: j
INCREMENT: ++
SEMI_COLON: ;
CLOSE_BRACE: }
INT_KEYWORD: int
IDENTIFIER: k
ASSIGNMENT: =
INTEGER_LITERAL: 0
SEMI_COLON: ;
DO_KEYWORD: do
OPEN_BRACE: {
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "%d\n"
COMMA: ,
IDENTIFIER: k
CLOSE_PAREN: )
SEMI_COLON: ;
IDENTIFIER: k
INCREMENT: ++
SEMI_COLON: ;
CLOSE_BRACE: }
```

```
WHILE_KEYWORD: while
OPEN_PAREN: (
IDENTIFIER: k
LESS_THAN: <
IDENTIFIER: MAX
CLOSE_PAREN: )
SEMI_COLON: ;
SWITCH_KEYWORD: switch
OPEN_PAREN: (
IDENTIFIER: number
CLOSE_PAREN: )
OPEN_BRACE: {
CASE_KEYWORD: case
INTEGER_LITERAL: 1
COLON: :
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "Number is 1\n"
CLOSE_PAREN: )
SEMI_COLON: ;
BREAK_KEYWORD: break
SEMI_COLON: ;
CASE_KEYWORD: case
INTEGER_LITERAL: 2
COLON: :
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "Number is 2\n"
CLOSE_PAREN: )
SEMI_COLON: ;
BREAK_KEYWORD: break
SEMI_COLON: ;
DEFAULT_KEYWORD: default
COLON: :
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "Number is not 1 or 2\n"
CLOSE_PAREN: )
SEMI_COLON: ;
BREAK_KEYWORD: break
SEMI_COLON: ;
CLOSE_BRACE: }
IDENTIFIER: printMessage
OPEN_PAREN: (
STRING_LITERAL: "This is a message"
CLOSE_PAREN: )
SEMI_COLON: ;
RETURN_KEYWORD: return
```

```
INTEGER_LITERAL: 0
SEMI_COLON: ;
CLOSE_BRACE: }
VOID_KEYWORD: void
IDENTIFIER: printMessage
OPEN_PAREN: (
CONST_KEYWORD: const
CHAR_KEYWORD: char
MULTIPLY: *
IDENTIFIER: message
CLOSE_PAREN: )
OPEN_BRACE: {
IDENTIFIER: printf
OPEN_PAREN: (
STRING_LITERAL: "%s\n"
COMMA: ,
IDENTIFIER: message
CLOSE_PAREN: )
SEMI_COLON: ;
CLOSE_BRACE: }
```

Parser:

Many challenges were faced in the making of the parser.

These are the steps followed to make the parser:

- 1- Find and write the grammar rules to be followed.
- 2- Implement each rule in a function.
- 3- Link the parser and lexer.
- 4- Rewrite the functions to return nodes for the parse tree.
- 5- Implement the code for the parse tree.

As mentioned, many challenges were faced therefore only the first three steps were completed.

The following is the grammar rules

primary_expression

: IDENTIFIER

| constant

| string

| '(' expression ')'

| generic_selection

;

constant

: I_CONSTANT/* includes character_constant */

| F_CONSTANT

| ENUMERATION_CONSTANT/* after it has been defined as such */

;

enumeration_constant/* before it has been defined as such */

: IDENTIFIER

;

string

: STRING_LITERAL

| FUNC_NAME

;

generic_selection

: GENERIC '(' assignment_expression ',' generic_assoc_list ')'

;

generic_assoc_list

: generic_association

| generic_assoc_list ',' generic_association

;

generic_association

: type_name ':' assignment_expression

| DEFAULT ':' assignment_expression

;

postfix_expression

: primary_expression

| postfix_expression '[' expression ']'

| postfix_expression '(' ')'

| postfix_expression '(' argument_expression_list ')'

| postfix_expression '.' IDENTIFIER

| postfix_expression PTR_OP IDENTIFIER

| postfix_expression INC_OP

| postfix_expression DEC_OP

| '(' type_name ')' '{' initializer_list '}'

| '(' type_name ')' '{' initializer_list ',' '}'

;

argument_expression_list

: assignment_expression

| argument_expression_list ',' assignment_expression

;

unary_expression

: postfix_expression

| INC_OP unary_expression

| DEC_OP unary_expression

| unary_operator cast_expression

| SIZEOF unary_expression

| SIZEOF '(' type_name ')'

| ALIGNOF '(' type_name ')'

;

unary_operator

: '&'

| '*'

| '+'

| '-'

| '~'

| '!'

;

cast_expression

: unary_expression

| '(' type_name ')' cast_expression

;

multiplicative_expression

: cast_expression

| multiplicative_expression '*' cast_expression

| multiplicative_expression '/' cast_expression

| multiplicative_expression '%' cast_expression

;

additive_expression

: multiplicative_expression

| additive_expression '+' multiplicative_expression

| additive_expression '-' multiplicative_expression

;

shift_expression

: additive_expression

| shift_expression LEFT_OP additive_expression

| shift_expression RIGHT_OP additive_expression

;

relational_expression

: shift_expression

| relational_expression '<' shift_expression

| relational_expression '>' shift_expression

| relational_expression LE_OP shift_expression

| relational_expression GE_OP shift_expression

;

equality_expression
: relational_expression
| equality_expression EQ_OP relational_expression
| equality_expression NE_OP relational_expression
;

and_expression
: equality_expression
| and_expression '&' equality_expression
;

exclusive_or_expression
: and_expression
| exclusive_or_expression '^' and_expression
;

inclusive_or_expression
: exclusive_or_expression
| inclusive_or_expression '|' exclusive_or_expression
;

logical_and_expression
: inclusive_or_expression
| logical_and_expression AND_OP inclusive_or_expression
;

logical_or_expression
: logical_and_expression
| logical_or_expression OR_OP logical_and_expression

;

conditional_expression

: logical_or_expression

| logical_or_expression '?' expression ':' conditional_expression

;

assignment_expression

: conditional_expression

| unary_expression assignment_operator assignment_expression

;

assignment_operator

: '='

| MUL_ASSIGN

| DIV_ASSIGN

| MOD_ASSIGN

| ADD_ASSIGN

| SUB_ASSIGN

| LEFT_ASSIGN

| RIGHT_ASSIGN

| AND_ASSIGN

| XOR_ASSIGN

| OR_ASSIGN

;

expression

: assignment_expression

| expression ',' assignment_expression

;

constant_expression

: conditional_expression/* with constraints */

;

declaration

: declaration_specifiers ';'

| declaration_specifiers init_declarator_list ';'

| static_assert_declaration

;

declaration_specifiers

: storage_class_specifier declaration_specifiers

| storage_class_specifier

| type_specifier declaration_specifiers

| type_specifier

| type_qualifier declaration_specifiers

| type_qualifier

| function_specifier declaration_specifiers

| function_specifier

| alignment_specifier declaration_specifiers

| alignment_specifier

;

init_declarator_list

: init_declarator

| init_declarator_list ',' init_declarator

;

init_declarator

: declarator '=' initializer

| declarator

;

storage_class_specifier

: TYPEDEF /* identifiers must be flagged as TYPEDEF_NAME */

| EXTERN

| STATIC

| THREAD_LOCAL

| AUTO

| REGISTER

;

type_specifier

: VOID

| CHAR

| SHORT

| INT

| LONG

| FLOAT

| DOUBLE

| SIGNED

| UNSIGNED

| BOOL

| COMPLEX

| IMAGINARY /* non-mandated extension */

| atomic_type_specifier

```
| struct_or_union_specifier  
| enum_specifier  
| TYPEDEF_NAME/* after it has been defined as such */  
;
```

```
struct_or_union_specifier  
: struct_or_union '{' struct_declaration_list '}'  
| struct_or_union IDENTIFIER '{' struct_declaration_list '}'  
| struct_or_union IDENTIFIER  
;
```

```
struct_or_union  
: STRUCT  
| UNION  
;
```

```
struct_declaration_list  
: struct_declaration  
| struct_declaration_list struct_declaration  
;
```

```
struct_declaration  
: specifier_qualifier_list ';'/* for anonymous struct/union */  
| specifier_qualifier_list struct_declarator_list ';'   
| static_assert_declaration  
;
```

```
specifier_qualifier_list  
: type_specifier specifier_qualifier_list
```

| type_specifier
| type_qualifier specifier_qualifier_list
| type_qualifier
;

struct_declarator_list
: struct_declarator
| struct_declarator_list ',' struct_declarator
;

struct_declarator
: ':' constant_expression
| declarator ':' constant_expression
| declarator
;

enum_specifier
: ENUM '{' enumerator_list '}'
| ENUM '{' enumerator_list ',' '}'
| ENUM IDENTIFIER '{' enumerator_list '}'
| ENUM IDENTIFIER '{' enumerator_list ',' '}'
| ENUM IDENTIFIER
;

enumerator_list
: enumerator
| enumerator_list ',' enumerator
;

enumerator/* identifiers must be flagged as ENUMERATION_CONSTANT */
: enumeration_constant '=' constant_expression
| enumeration_constant
;

atomic_type_specifier
: ATOMIC '(' type_name ')'
;

type_qualifier
: CONST
| RESTRICT
| VOLATILE
| ATOMIC
;

function_specifier
: INLINE
| NORETURN
;

alignment_specifier
: ALIGNAS '(' type_name ')'
| ALIGNAS '(' constant_expression ')'
;

declarator
: pointer direct_declarator
| direct_declarator

;

direct_declarator

: IDENTIFIER

| '(' declarator ')'

| direct_declarator '[' ']'

| direct_declarator '[' '*' ']'

| direct_declarator '[' STATIC type_qualifier_list assignment_expression ']'

| direct_declarator '[' STATIC assignment_expression ']'

| direct_declarator '[' type_qualifier_list '*' ']'

| direct_declarator '[' type_qualifier_list STATIC assignment_expression ']'

| direct_declarator '[' type_qualifier_list assignment_expression ']'

| direct_declarator '[' type_qualifier_list ']'

| direct_declarator '[' assignment_expression ']'

| direct_declarator '(' parameter_type_list ')'

| direct_declarator '(' ' ')

| direct_declarator '(' identifier_list ')'

;

pointer

: '*' type_qualifier_list pointer

| '*' type_qualifier_list

| '*' pointer

| '*'

;

type_qualifier_list

: type_qualifier

| type_qualifier_list type_qualifier

;

parameter_type_list

: parameter_list ',' ELLIPSIS

| parameter_list

;

parameter_list

: parameter_declaration

| parameter_list ',' parameter_declaration

;

parameter_declaration

: declaration_specifiers declarator

| declaration_specifiers abstract_declarator

| declaration_specifiers

;

identifier_list

: IDENTIFIER

| identifier_list ',' IDENTIFIER

;

type_name

: specifier_qualifier_list abstract_declarator

| specifier_qualifier_list

;

abstract_declarator

: pointer direct_abstract_declarator

| pointer

| direct_abstract_declarator

;

direct_abstract_declarator

: '(' abstract_declarator ')'

| '[' ']'

| '[' '*' ']'

| '[' STATIC type_qualifier_list assignment_expression ']'

| '[' STATIC assignment_expression ']'

| '[' type_qualifier_list STATIC assignment_expression ']'

| '[' type_qualifier_list assignment_expression ']'

| '[' type_qualifier_list ']'

| '[' assignment_expression ']'

| direct_abstract_declarator '[' ']'

| direct_abstract_declarator '[' '*' ']'

| direct_abstract_declarator '[' STATIC type_qualifier_list assignment_expression ']'

| direct_abstract_declarator '[' STATIC assignment_expression ']'

| direct_abstract_declarator '[' type_qualifier_list assignment_expression ']'

| direct_abstract_declarator '[' type_qualifier_list STATIC assignment_expression ']'

| direct_abstract_declarator '[' type_qualifier_list ']'

| direct_abstract_declarator '[' assignment_expression ']'

| '(' ')'

| '(' parameter_type_list ')'

| direct_abstract_declarator '(' ')'

| direct_abstract_declarator '(' parameter_type_list ')'

;

initializer

: '{' initializer_list '}'

| '{' initializer_list ',' '}'

| assignment_expression

;

initializer_list

: designation initializer

| initializer

| initializer_list ',' designation initializer

| initializer_list ',' initializer

;

designation

: designator_list '='

;

designator_list

: designator

| designator_list designator

;

designator

: '[' constant_expression ']'

| '.' IDENTIFIER

;

static_assert_declaration

: STATIC_ASSERT '(' constant_expression ',' STRING_LITERAL ')' ';' ;

statement

: labeled_statement
| compound_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement ;

labeled_statement

: IDENTIFIER ':' statement
| CASE constant_expression ':' statement
| DEFAULT ':' statement ;

compound_statement

: '{' '
| '{' block_item_list '}' ;

block_item_list

: block_item
| block_item_list block_item ;

block_item

: declaration
| statement
;

expression_statement

: ';'
| expression ';'
;

selection_statement

: IF '(' expression ')' statement ELSE statement
| IF '(' expression ')' statement
| SWITCH '(' expression ')' statement
;

iteration_statement

: WHILE '(' expression ')' statement
| DO statement WHILE '(' expression ')' ';'
| FOR '(' expression_statement expression_statement ')' statement
| FOR '(' expression_statement expression_statement expression ')' statement
| FOR '(' declaration expression_statement ')' statement
| FOR '(' declaration expression_statement expression ')' statement
;

jump_statement

: GOTO IDENTIFIER ';'
| CONTINUE ';'
| BREAK ';'
| RETURN ';'
;

| RETURN expression ';' ;

translation_unit
: external_declaration
| translation_unit external_declaration ;

external_declaration
: function_definition
| declaration ;

function_definition
: declaration_specifiers declarator declaration_list compound_statement
| declaration_specifiers declarator compound_statement ;

declaration_list
: declaration
| declaration_list declaration ;

The implementation of the grammar rules went as follows:

- 1- Create a function with the name of the grammar rule
- 2- Implement the rules of the grammar rule
- 3- Follow each grammar rule and implement the contained grammar till we reach the end

Implementaiont

```
bool unaryOperator() {
    if (currentIndex >= tokens.size()) {
        return false;
    }
    string tokenType = tokens[currentIndex].type;
    if (tokenType == "BITWISE_AND" || tokenType == "MULTIPLY" || tokenType == "PLUS" ||
        tokenType == "MINUS" || tokenType == "BITWISE_NOT" || tokenType == "LOGICAL_NOT") {
        get_next_token();
        return true;
    }
    return false;
}

bool isTypeName(const Token& token) {
    return token.type == "IDENTIFIER" || token.type == "TYPE_KEYWORD";
}

bool typeSpec() {
    if (currentIndex >= tokens.size()) {
        return false;
    }
    string tokenType = tokens[currentIndex].type;
    if (tokenType == "INT_KEYWORD" || tokenType == "FLOAT_KEYWORD" || tokenType == "CHAR_KEYWORD" || tokenType == "DOUBLE_KEYWORD") {
        get_next_token();
        return true;
    }
    else if (tokenType == "LONG_KEYWORD") {
        get_next_token();
        typeSpec();
        return true;
    }
    return false;
}
```



```

bool primaryExpression() {
    if (currentIndex >= tokens.size()) {
        return false;
    }
    string tokenType = tokens[currentIndex].type;
    if (tokenType == "IDENTIFIER" || tokenType == "INTEGER_LITERAL" || tokenType == "STRING_LITERAL") {
        get_next_token();
        return true;
    }
    else if (tokenType == "OPEN_PAREN") {
        get_next_token();
        if (expression()) {
            if (tokenType == "CLOSE_PAREN") {
                get_next_token();
                return true;
            }
        }
    }
    else if (genericSelection()) {
        get_next_token();
        return true;
    }
    return false;
}

bool genericSelection() {
    if (currentIndex >= tokens.size()) {
        return false;
    }
    string tokenType = tokens[currentIndex].type;
    if (tokenType == "GENERIC_KEYWORD") {
        get_next_token();
        if (tokenType == "OPEN_PAREN") {
            get_next_token();
            if (assignmentExpression()) {
                if (tokenType == "COMMA") {
                    get_next_token();
                    if (genericAssocList()) {
                        if (tokenType == "CLOSE_PAREN") {
                            get_next_token();
                            return true;
                        }
                    }
                }
            }
        }
    }
    return false;
}

```

```

bool typeName() {
    if (currentIndex >= tokens.size()) {
        return false;
    }

    string tokenType = tokens[currentIndex].type;
    if (tokenType == "INT_KEYWORD" || tokenType == "CHAR_KEYWORD" || tokenType == "") {
        get_next_token(); // Consume the type keyword
        // Handle additional specifiers if necessary
        return true;
    }

    // Handle other type name cases if needed
    return false;
}

// Function to parse generic associations
bool genericAssociation() {
    if (currentIndex >= tokens.size()) {
        return false;
    }

    string tokenType = tokens[currentIndex].type;

    if (typeName()) {
        if (tokenType == "COLON") {
            get_next_token();
            if (assignmentExpression())
                return true;
        }
    }

    else if (tokenType == "DEFAULT_KEYWORD") {
        get_next_token();
        if (tokenType == "COLON") {
            get_next_token();
            if (assignmentExpression())
                return true;
        }
    }

    // Handle other cases if needed
    return false;
}

```

```

// Function to parse postfix expressions
bool postfixExpression() {
    if (currentIndex >= tokens.size()) {
        return false;
    }
    string tokenType = tokens[currentIndex].type;
    if (primaryExpression()) {
        while (true) {
            if (tokenType == "OPEN_SQUARE_BRACKET") {
                get_next_token();
                if (!expression())
                    return false;
                if (tokenType == "CLOSE_SQUARE_BRACKET")
                    get_next_token();
                else
                    return false;
            }
            else if (tokenType == "OPEN_PAREN") {
                get_next_token();
                if (argumentExpressionList()) {
                    if (tokenType == "CLOSE_PAREN")
                        get_next_token();
                    else
                        return false;
                }
                else
                    return false;
            }
            else if (tokenType == "DOT" || tokenType == "ARROW") {
                get_next_token();
                if (tokenType == "IDENTIFIER")
                    get_next_token();
                else
                    return false;
            }
            else if (tokenType == "INCREMENT" || tokenType == "DECREMENT") {
                get_next_token();
            }
            else {
                break;
            }
        }
        return true;
    }
    // Handle other cases if needed
    return false;
}

```

Following the same manner the rest of the rules were implemented

Output sample for the parser:

```
INT_KEYWORD : int
IDENTIFIER  : main
2 2
Parsing successful!
```

```
INT_KEYWORD : int
IDENTIFIER  : main
INTEGER_LITERAL : 76
2 3
Parsing failed! Unexpected tokens remain.
```