



# Documentation

**Course:** CSE439 - Design of compilers

Student ID	Name
21P0186	Saifeldin Mohamed Hatem
22P0122	Basel Ashraf Fikry
2001038	Mohamed Ahmed Abdelraouf
2000052	Ahmed Mohamed Abdelmonem
21P0285	Abdalrahman Khaled Alkawwa

**Submitted to:**

Dr. Wafaa Al Kasas

Eng. Ahmad Salama

# Contents

<b>1</b>	<b>Language Specifications</b>	<b>4</b>
1.1	Keywords in C . . . . .	4
1.1.1	Additional Keywords (C99 and later): . . . . .	6
1.2	C Variable Identifiers . . . . .	6
1.2.1	Naming Rules: . . . . .	7
1.3	Functions . . . . .	7
1.3.1	Function Identifiers . . . . .	11
1.3.2	Data Types . . . . .	12
1.3.3	Inline functions . . . . .	13
1.3.4	No Return functions: . . . . .	16
1.3.5	Function Declarations . . . . .	19
1.3.6	New-style function declaration . . . . .	19
1.3.7	Old-style (K&R) function definition . . . . .	20
1.3.8	Non-prototype function declaration . . . . .	20
1.3.9	Return Statement . . . . .	22
1.4	Expressions . . . . .	23
1.4.1	Boolean . . . . .	24
1.4.1.1	Logical operators . . . . .	25
1.4.1.2	Arithmetic Expressions . . . . .	26
1.4.2	Assignment Operators . . . . .	30
1.4.3	Ternary Operator . . . . .	30
1.4.4	Operator precedence . . . . .	30
1.5	Statements . . . . .	32
1.5.1	Assignment Statement . . . . .	32
1.5.2	Declaration of Variables . . . . .	34
1.5.3	Return Statement . . . . .	36
1.5.4	Iterative Statement . . . . .	37

---

1.5.5	if statement & if-else statement . . . . .	41
1.5.6	switch statement . . . . .	42
<b>2</b>	<b>Lexer</b>	<b>42</b>
2.1	Defined Tokens . . . . .	42
2.2	Symbol Table . . . . .	44
2.3	Implementation Details . . . . .	45
2.4	Lexer Test Cases . . . . .	46
2.4.1	Test Case 1 . . . . .	46
2.4.1.1	Test Case Code . . . . .	46
2.4.1.2	Test Case Output . . . . .	47
2.4.2	Test Case 2 . . . . .	51
2.4.2.1	Test Case Code . . . . .	51
2.4.2.2	Test Case Output . . . . .	51
2.4.3	Test Case 3 . . . . .	54
2.4.3.1	Test Case Code . . . . .	54
2.4.3.2	Test Case Output . . . . .	54
2.4.3.3	Test Case Output . . . . .	58
2.4.4	Test Case 4 . . . . .	59
2.4.4.1	Test Case Code . . . . .	59
2.4.4.2	Test Case Output . . . . .	59
2.4.5	Test Case 5 . . . . .	59
2.4.5.1	Test Case Code . . . . .	59
2.4.5.2	Test Case Output . . . . .	59
2.4.6	Lexer Test Cases Screenshots . . . . .	60
2.4.6.1	Test Case 1 Screenshots . . . . .	60
2.4.6.2	Test Case 2 Screenshots . . . . .	62
2.4.6.3	Test Case 3 Screenshots . . . . .	63
2.4.6.4	Test Case 4 Screenshots . . . . .	66
2.4.6.5	Test Case 5 Screenshots . . . . .	66

<b>3</b>	<b>Parser</b>	<b>66</b>
3.1	Grammar Rules . . . . .	66
3.2	Implementation Details . . . . .	69
3.2.1	Parsing Algorithm . . . . .	69
3.2.2	General Overview . . . . .	70
3.3	Parse Tree Screenshots . . . . .	71

## List of Figures

1	Common Operators . . . . .	24
2	Assignment of operators . . . . .	30
3	Screenshot 1 of Test Case 1 Output . . . . .	60
4	Screenshot 2 of Test Case 1 Output . . . . .	61
5	Screenshot 3 of Test Case 1 Output . . . . .	62
6	Screenshot 1 of Test Case 2 Output . . . . .	62
7	Screenshot 2 of Test Case 2 Output . . . . .	62
8	Screenshot 1 of Test Case 3 Output . . . . .	63
9	Screenshot 2 of Test Case 3 Output . . . . .	64
10	Screenshot 3 of Test Case 3 Output . . . . .	65
11	Screenshot 4 of Test Case 3 Output . . . . .	65
12	Screenshot of Test Case 4 Output . . . . .	66
13	Screenshot of Test Case 5 Output . . . . .	66
14	Parse Tree . . . . .	71

## List of Tables

1	Logical operators . . . . .	25
---	-----------------------------	----

# 1 Language Specifications

## 1.1 Keywords in C

Keywords are predefined, reserved words in the C language that have specific meanings and cannot be used for other purposes like variable names. They form the essential building blocks for constructing C programs, defining control flow, data types, and program behavior.

- **Data types:** `int`, `char`, `float`, `double`
- **Control flow:** `if`, `else`, `for`, `while`, `switch`
- **Functions:** `void`, `return`
- **Storage:** `auto`, `static`, `extern`
- **Others:** `break`, `continue`, `sizeof`, `typedef`, `enum`

Here's a comprehensive list of all C keywords:

<b>auto:</b>	Used for automatic storage duration of variables.
<b>break:</b>	Exits a loop or switch statement.
<b>case:</b>	Used in switch statements to specify conditions.
<b>char:</b>	Declares character variables.
<b>const:</b>	Declares constant variables.
<b>continue:</b>	Skips the remaining code in the current iteration of a loop.
<b>default:</b>	Used in switch statements as a default case.
<b>do:</b>	Used in do-while loop construct.
<b>double:</b>	Declares double-precision floating-point variables.
<b>else:</b>	Used in conditional statements (if-else).

---

<b>enum:</b>	Defines user-defined data types consisting of named integer constants.
<b>extern:</b>	Declares variables or functions defined in another source file.
<b>float:</b>	Declares single-precision floating-point variables.
<b>for:</b>	Used for loop construct with initialization, condition, and increment/decrement.
<b>goto:</b>	Transfers program execution to a labeled statement. (Use with caution due to potential code complexity)
<b>if:</b>	Used for conditional statements.
<b>int:</b>	Declares integer variables.
<b>long:</b>	Declares long integer variables.
<b>register:</b>	Suggests the compiler to store the variable in a register (compiler-specific implementation).
<b>return:</b>	Exits a function and optionally returns a value.
<b>short:</b>	Declares short integer variables.
<b>signed:</b>	Declares signed integer variables (default for integer types).
<b>sizeof:</b>	Returns the size of a variable or data type in bytes.
<b>static:</b>	Declares variables with static storage duration.
<b>struct:</b>	Defines user-defined data types that group variables of different types.
<b>switch:</b>	Used for multi-way branching based on different conditions.
<b>typedef:</b>	Creates aliases for data types.

<b>union:</b>	Defines user-defined data types where all members share the same memory location.
<b>unsigned:</b>	Declares unsigned integer variables (no negative values).
<b>void:</b>	Used for functions that do not return a value or to declare a pointer to any data type.
<b>volatile:</b>	Indicates that a variable's value can be changed outside the program's control.
<b>while:</b>	Used for loop construct based on a condition.

### 1.1.1 Additional Keywords (C99 and later):

- **\_Alignas:** Specifies alignment requirements for variables (compiler-specific).
- **\_Alignof:** Returns the alignment requirements of a data type.
- **\_Atomic:** Denotes atomic operations (thread-safe operations).
- **\_Bool:** Boolean data type (represented by 0 or 1).
- **\_Complex:** Complex number data type.
- **\_Generic:** Used for generic functions (not commonly used).
- **\_Imaginary:** Imaginary unit data type.
- **\_Noreturn:** Indicates a function that does not return.
- **\_Static\_assert:** Used for compile-time assertions.
- **\_Thread\_local:** Declares thread-local storage.

## 1.2 C Variable Identifiers

Variable identifiers in C are used to name and represent data storage locations in memory.

### 1.2.1 Naming Rules:

- They can start with a letter (uppercase or lowercase) or underscore (`_`).
- Subsequent characters can be letters, numbers, or underscores.
- They are case-sensitive, meaning `age` and `Age` are considered different variables.
- They cannot be reserved keywords in C (like `int`, `for`, `while`, etc.).

## 1.3 Functions

Function definition: A function declaration, in simpler terms, acts like a blueprint for creating a function. It provides the name of the function, and optional details about what kind of information it expects (parameters) and what kind of information it provides (return type). This blueprint is available throughout your code, similar to how you might declare a variable at the beginning of your program.

Syntax:

- `noptr-declarator (parameter-list) attr-spec-seq (optional)`
- `noptr-declarator (identifier-list) attr-spec-seq (optional)`
- `noptr-declarator ( ) attr-spec-seq (optional)`

The following keywords were used above in the syntax:

**noptr-declarator:** any declarator except unparenthesized pointer declarator. The identifier that is contained in this declarator is the identifier that becomes the function designator. For example, `int myFunction(int x, int y);` `MyFunction` is a `noptr-declarator`.

**parameter-list:** either the single keyword `void` or a comma-separated list of parameters, which may end with an ellipsis parameter. For example, `void printMessage(char *message);`



**identifier-list:** comma-separated list of identifiers, only possible if this declarator is used as part of an old-style function definition. For example, `int multiply(a, b) { int a, b; return a * b; }`

**attr-spec-seq:** an optional list of attributes, applied to the function type. For example, `[[nodiscard]] int calculate(int x, int y);`

Function declarations: Functions can be declared in several ways, for example:

```
int min(int a, int b);
```

If values of higher precision are passed into the function, the compiler converts those values into the appropriate type that is declared in the function parameters, for example:

```
int n = min(90.23, 32.12);
```

Both 90.23 and 32.12 are of the data type double, but they are converted to int. This is in the New Style C89 function declaration syntax.

Take a look at this example retrieved from the [cpreference](#) website:

```
int max(a, b)
int a, b; // definition expects ints; the second call is undefined
{
    return a > b ? a : b;
}

int n = max(true, (char)'a'); // calls max with two int args (after promotions)
int n = max(12.01f, 3.14); // calls max with two double args (after promotions)
```

Notice that the number of arguments passed to the max function above must be 2. As the function takes two parameters, `a` and `b`, both of which are int type. While the following example might seem confusing, it is important to understand how the C language works in order to understand why passing `true` and the letter `'a'` won't give a syntax error.

The reason is that `true` translates to the value 1 in C, and the character `'a'` has been type casted to char. This causes the Compiler to translate the `'a'` to its ASCII value, which is also an integer, hence two integers are being compared so no error.

The second function call is straightforward, the two arguments are converted into ints by decimal truncation. So 12 and 3 are being compared. Via `a > b ? a : b`, where `?` is the ternary operator.

### Return Type of a Function

The return type of a function is marked by the first word that comes before the function identifier, for example

```
int Square(a); // Function is expected to return an integer
```

Here are more examples from the [cppreference website](#):

```
void f(char *s); // return type is void
int sum(int a, int b); // return type of sum is int.
int (*foo(const void *p))[3]; // return type is pointer to array of 3 int
double const bar(void); // declares function of type double(void)
double (*barp)(void) = bar; // OK: barp is a pointer to double(void)
double const (*barpc)(void) = barp; // OK: barpc is also a pointer to double(void)
```

If a function declaration appears outside of any function, the identifier it introduces has file scope and external linkage, unless `static` is used or an earlier `static` declaration is visible. If the declaration occurs inside another function, the identifier has block scope (and also either internal or external linkage).

```
int main(void)
{
    int f(int); // external linkage, block scope
    f(1); // definition needs to be available somewhere in the program
}
```

Take the following example and its function call:

```
int f(int[]); // declares int f(int*)
```

Note: the pointer is to the first index of the array reference being passed into the function.

```
int f(char g(double)); // declares int f(char (*g)(double))
```

As the parameter is a pointer.

Please note that you cannot have `void` data type in parameters of functions:

```
int f(void); // OK
int g(void x); // Error
```

While it may seem confusing, the function `f` actually takes no parameters, so passing parameters would throw an error. This is different from `int f();` which means the function parameters are not yet specified, however, putting `void` in the parameter means that the function is not accepting any parameters.

The following example demonstrates this behavior further:

```
int f(void); // declaration: takes no parameters
int g(); // declaration: takes unknown parameters
int main(void) {
    f(1); // compile-time error
    g(2); // undefined behavior
}
int f(void) { return 1; } // actual definition
int g(a,b,c,d) int a,b,c,d; { return 2; } // actual definition
```

However, the second function, `g`, has a type variable of `void`, `x`, which is not allowed, hence it is an error. **Typedefs in functions**

Typedefs can also be used in functions, which enhances reusability, an important concept in software engineering.

Take this example

```
typedef int p(int q, int r); // p is a function type int(int, int)
p f; // declares int f(int, int)
```

You can declare another function, `f` in the above example with the same parameters and return type by using the `typedef` of the `p` function.

### 1.3.1 Function Identifiers

**Definition:** Function identifiers are unique names assigned to functions in C. They act as labels, allowing you to call the function and execute its code from other parts of your program.

**Rules:**

- **Characters:** Must start with a letter (uppercase or lowercase) or an underscore (`_`). Subsequent characters can be letters, digits, or underscores.
- **Case-sensitivity:** C is case-sensitive. `calculateArea` and `calCUlateArea` are considered different functions.
- **Keywords:** Cannot be the same as C’s reserved keywords (e.g., `int`, `float`, `while`, `if`).

**Examples:**

- `calculateAverage`
- `checkIfPrime`
- `_internalFunction` (functions starting with underscores are generally considered private)

**Invalid Examples:**

- `123abc` (cannot start with a digit)
- `my-function` (hyphens are not allowed)
- `int` (same as a C keyword)

### Compiler Considerations

- The compiler performs name resolution, associating function identifiers with their corresponding code blocks. It checks for duplicate names, ensuring their uniqueness within the program’s scope.

- Some compilers may have limitations on function identifier length. Consult your compiler's documentation for specific restrictions.

### 1.3.2 Data Types

**Definition:** Data types in C specify the kind of data a variable or expression can hold. This determines:

- Type of values the variable can store (integers, floating-point numbers, characters, etc.)
- Amount of memory allocated for the variable
- Operations that can be performed on the data

#### Basic Data Types:

- **char:** Represents a single character (stored as ASCII or Unicode code). Size is typically 1 byte.

```
char letter = 'A';
```

- **short:** Used to store integers with a smaller range than **int**. Size is typically 2 bytes.

```
short num = 10;
```

- **int:** Used for storing integers within a system-specific range. Size can vary depending on the system and compiler (typically 4 bytes).

```
int count = 100;
```

- **long:** Used to store integers with a wider range than **int**. Minimum size is guaranteed to be at least 32 bits (4 bytes) on most systems.

```
long bigNumber = 1000000L;
```

- **float:** Stores single-precision floating-point numbers. Size is usually 4 bytes.

```
float pi = 3.14f;
```

- **double**: Stores double-precision floating-point numbers. Size is usually 8 bytes.

```
double distance = 123.456;
```

- **void**: Represents the absence of a value.

```
void *ptr;
```

### Derived Data Types:

- **Arrays**: Collections of elements of the same data type.
- **Structures**: User-defined types that group variables of different data types under a single name.
- **Unions**: Similar to structures but can only hold one value at a time.
- **Pointers**: Variables that store memory addresses of other variables.

### 1.3.3 Inline functions

A part of compilation of the code is optimization, and the **inline** keyword does that. It lets the compiler perform optimizations.

But how?

By default, when there is no **inline** keyword, the compiler places the call of a function in a stack, which uses resources and cause increased computation and compilation time. This introduces an overhead called the function call overhead.

So what does the **inline** do?

Function inlining replaces the function call with its body directly, which is part of optimization of a code by the compiler.

To further clarify the difference, in large executable codes, without the **inline** keyword, the compiler would have to store multiple calls to the function in the stack, which increases compilation time.

Take the following example:

```
static int x;
inline void f(void)
{
    static int n = 1; // error: non-const static in a non-static inline function
    int k = x; // error: non-static inline function accesses a static variable
}
```

You might have realized that this code has errors. Firstly, you cannot have a static variable which isn't a constant (once declared can't be changed in the runtime) in a non-static inline function definition. So it gives an error.

Assigning a non-static variable like `k` in the example above to `x`, which is static is also not allowed, as `k` is in the scope of the inline function and non-static variables in inline functions can't be assigned to static variables.

An important reserved word is the `extern` keyword, this word is important especially in inline functions, let's say you have `file1` and `file2`. Your inline function is in the source code of `file1`. `File2` source code can't access that function without the `extern` keyword, `extern` allows foreign files to access inline functions.

Example:

```
// File: inline_example.h

#ifndef INLINE_EXAMPLE_H
#define INLINE_EXAMPLE_H
extern inline int add(int a, int b) {
    return a + b;
}
extern int subtract(int a, int b);
#endif
```

```
// File: inline_example.c

#include "inline_example.h"
// Define the non-inline function
```

```
int subtract(int a, int b) {
    return a - b;
}

// File:  main.c

#include <stdio.h>
#include "inline_example.h"
int main() {
    // Calling the inline function
    printf("Result of adding: %d\n", add(5, 3)); // Output: Result of adding: 8
    // Calling the non-inline function
    printf("Result of subtracting: %d\n", subtract(5, 3)); // Output: Result of subtract
    return 0;
}
```

In this example, assuming all files are correctly compiled and linked:

- The output of the `printf` statement calling `add(5, 3)` will be "Result of adding: 8", as the function `add()` adds the two given numbers.
- The output of the `printf` statement calling `subtract(5, 3)` will be "Result of subtracting: 2", as the function `subtract()` subtracts the second parameter from the first.
- There will be no errors if all files are correctly compiled and linked. Another example:

```
inline const char *saddr(void) // the inline definition for use in this file
{
    static const char name[] = "saddr";
    return name;
}

int compare_name(void)
{
    return saddr() == saddr(); // unspecified behavior, one call could be external
}

extern const char *saddr(void); // an external definition is generated, too
```



While the code could compile with no syntax errors, it is important to note that there is a possibility of unspecified behavior if two functions have the same identifier and in different files. In the above example, the `saddr()` function could be declared somewhere else outside that source file which could cause problems for the compiler to determine which `saddr()` function to call. (The one in the same source code file or in an external source code file?). This leads to unexpected behavior.

#### 1.3.4 No Return functions:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>
// causes undefined behavior if i <= 0
// exits if i > 0
noreturn void exit_now(int i) // or _Noreturn void exit_now(int i)
{
    if (i > 0)
        exit(i);
}
int main(void)
{
    puts("Preparing to exit...");
    exit_now(2);
    puts("This code is never executed.");
}
```

Take a look at the above example, the `noreturn` keyword indicates that the function could possibly be exited abruptly and not via the `return` keyword, this can be found by including the `<stdnoreturn.h>` header.

But How abruptly? By using keywords.

1. `abort()`: This function terminates the program abnormally. It does not perform any

cleanup of resources; instead, it just causes the program to terminate immediately, possibly printing a diagnostic message.

2. `exit()`: This function terminates the program normally. It performs cleanup of resources, such as closing files and releasing memory, before terminating the program. The integer argument passed to `exit()` is a status code indicating the reason for termination.
3. `_Exit()`: This function is similar to `exit()`, but it does not perform any cleanup before termination. It immediately terminates the program without performing any cleanup tasks. Like `exit()`, you can also pass an integer argument to `_Exit()` to indicate the termination status.
4. `quick_exit()`: This function is similar to `_Exit()`, but it performs minimal cleanup before termination. It does not flush open streams or call exit handlers registered with `atexit()`. Instead, it just terminates the program and returns control to the operating system.
5. `thrd_exit()`: This function is part of the C11 standard and is used to terminate the current thread in a multithreaded program. It allows the thread to exit with a specified status code.
6. `longjmp()`: This function performs a non-local goto, allowing the program to jump to a previously established point in the code. It is often used for error handling or to unwind the call stack in exceptional circumstances. However, it does not perform stack unwinding or call destructors like C++ exceptions; instead, it directly transfers control to a specified point in the program.

Output of the code above: Preparing to exit...

A function is a C language construct that associates a compound statement (the function body) with an identifier (the function name). Every C program begins execution from the `main` function, which either terminates or invokes other, user-defined or library functions.

**Example:**

```
// function definition.  
// defines a function with the name "sum" and with the  
// body "{ return x+y; }"  
int sum(int x, int y)  
{  
    return x + y;  
}
```

A function is introduced by a function declaration or a function definition. Functions may accept zero or more parameters, which are initialized from the arguments of a function call operator, and may return a value to its caller by means of the return statement.

```
int n = sum(1, 2); // parameters x and y are  
initialized with the arguments 1 and 2
```

The body of a function is provided in a function definition. Each non-inline function that is used in an expression (unless unevaluated) must be defined only once in a program. There are no nested functions (except where allowed through non-standard compiler extensions): each function definition must appear at file scope, and functions have no access to the local variables from the caller.

```
int main(void) // the main function definition  
{  
    int sum(int, int); // function declaration (may appear at any scope)  
    int x = 1; // local variable in main  
    sum(1, 2); // function call  
}  
  
int sum(int a, int b) // function definition  
{  
    // return x + a + b; // error: main's x is not accessible within sum  
    return a + b;  
}
```

### 1.3.5 Function Declarations

A function declaration introduces an identifier that designates a function and, optionally, specifies the types of the function parameters (the prototype). Function declarations (unlike definitions) may appear at block scope as well as file scope.

#### Syntax

In the declaration grammar of a function declaration, the type-specifier sequence, possibly modified by the declarator, designates the return type (which may be any type other than array or function type), and the declarator has one of three forms:

noPtr-declarator ( parameter-list ) attr-spec-seq(optional)

noPtr-declarator ( identifier-list ) attr-spec-seq(optional)

noPtr-declarator ( ) attr-spec-seq(optional)

#### Explanation

- **noPtr-declarator** - any declarator except unparenthesized pointer declarator. The identifier that is contained in this declarator is the identifier that becomes the function designator.
- **parameter-list** - either the single keyword `void` or a comma-separated list of parameters, which may end with an ellipsis parameter.
- **identifier-list** - comma-separated list of identifiers, only possible if this declarator is used as part of old-style function definition.
- **attr-spec-seq** - (C23) an optional list of attributes, applied to the function type.

### 1.3.6 New-style function declaration

This declaration both introduces the function designator itself and also serves as a function prototype for any future function call expressions, forcing conversions from argument expressions to the declared parameter types and compile-time checks for the number of arguments.

```
int max(int a, int b); // declaration
int n = max(12.01, 3.14); // OK, conversion from double to int
```

### 1.3.7 Old-style (K&R) function definition

This declaration does not introduce a prototype and any future function call expressions will perform default argument promotions and will invoke undefined behavior if the number of arguments doesn't match the number of parameters.

```
int max(a, b)
int a, b; // definition expects ints; the second call is undefined
{
    return a > b ? a : b;
}
int n = max(true, (char)'a'); // calls max with two int args (after promotions)
int n = max(12.01f, 3.14); // calls max with two double args (after promotions)
```

### 1.3.8 Non-prototype function declaration

This declaration does not introduce a prototype. A new style function declaration equivalent to the parameter-list `void`.

#### Explanation

The return type of the function, determined by the type specifier in specifiers-and-qualifiers and possibly modified by the declarator as usual in declarations, must be a non-array object type or the type `void`. If the function declaration is not a definition, the return type can be incomplete. The return type cannot be cvr-qualified: any qualified return type is adjusted to its unqualified version for the purpose of constructing the function type.

```
void f(char *s); // return type is void
int sum(int a, int b); // return type of sum is int
int (*foo(const void *p))[3]; // return type is pointer to array of 3 int
double const bar(void); // declares function of type double(void)
double (*barp)(void) = bar; // OK: barp is a pointer to double(void)
```

```
double const (*barpc)(void) = barp; // OK: barpc is also a pointer to double(void)
```

Function declarators can be combined with other declarators as long as they can share their type specifiers and qualifiers.

```
int f(void), *fip(), (*pfi>(), *ap[3]; // declares two functions and two objects
inline int g(int), n; // Error: inline qualifier is for functions only
typedef int array_t[3];
array_t a, h(); // Error: array type cannot be a return type for a function
```

If a function declaration appears outside of any function, the identifier it introduces has file scope and external linkage, unless **static** is used or an earlier **static** declaration is visible. If the declaration occurs inside another function, the identifier has block scope (and also either internal or external linkage).

```
int main(void)
{
    int f(int); // external linkage, block scope
    f(1); // definition needs to be available somewhere in the program
}
```

The parameters in a declaration that is not part of a function definition (until C23) do not need to be named:

```
int f(int, int); // declaration
// int f(int, int) { return 7; } // Error: parameters must be named in definitions
// This definition is allowed since C23
```

Each parameter in a parameter-list is a declaration that introduced a single variable, with the following additional properties:

- the identifier in the declarator is optional (except if this function declaration is part of a function definition) (until C23)

- the only storage class specifier allowed for parameters is **register**, and it is ignored in function declarations that are not definitions
- any parameter of array type is adjusted to the corresponding pointer type, which may be qualified if there are qualifiers between the square brackets of the array declarator (since C99)
- any parameter of function type is adjusted to the corresponding pointer type
- the parameter list may terminate with `,...` or be `...` (since C23), see variadic functions for details.
- parameters cannot have type **void** (but can have type pointer to void). The special parameter list that consists entirely of the keyword **void** is used to declare functions that take no parameters.
- any identifier that appears in a parameter list that could be treated as a typedef name or as a parameter name is treated as a typedef name
- parameters may have incomplete type and may use the VLA notation (since C99) (except that in a function definition, the parameter types after array-to-pointer and function-to-pointer adjustment must be complete)

Attribute specifier sequences can also be applied to function parameters. See function call operator for other details on the mechanics of a function call and return for returning from functions.

### 1.3.9 Return Statement

The **return** statement terminates the current function and returns a specified value to the caller function.

#### Syntax

attr-spec-seq(optional) return expression;

attr-spec-seq(optional) return;

**Explanation:**

1. Evaluates the expression, terminates the current function, and returns the result of the expression to the caller (the value returned becomes the value of the function call expression). Only valid if the function return type is not **void**.
2. Terminates the current function. Only valid if the function return type is **void**.

If the type of the expression is different from the return type of the function, its value is converted as if by assignment to an object whose type is the return type of the function, except that overlap between object representations is permitted.

**Example:**

```
struct s { double i; } f(void); // function returning struct s
union { struct { int f1; struct s f2; } u1;
        struct { struct s f3; int f4; } u2; } g;
struct s f(void)
{
    return g.u1.f2;
}
```

Reaching the end of a function returning **void** is equivalent to **return;**.

Reaching the end of any other value-returning function is undefined behavior if the result of the function is used in an expression (it is allowed to discard such return value). For **main**, see main function.

Executing the **return** statement in a no-return function is undefined behavior.

## 1.4 Expressions

Retrieved from cppreference.



Common operators						
<u>assignment</u>	increment decrement	arithmetic	logical	comparison	member access	other
<pre> a = b a += b a -= b a *= b a /= b a %= b a &amp;= b a  = b a ^= b a &lt;&lt;= b a &gt;&gt;= b </pre>	<pre> ++a --a a++ a-- </pre>	<pre> +a -a a + b a - b a * b a / b a % b ~a a &amp; b a   b a ^ b a &lt;&lt; b a &gt;&gt; b </pre>	<pre> !a a &amp;&amp; b a    b </pre>	<pre> a == b a != b a &lt; b a &gt; b a &lt;= b a &gt;= b </pre>	<pre> a[b] *a &amp;a a-&gt;b a.b </pre>	<pre> a(...) a, b (type) a a ? b : c sizeof  _Alignof (since C11) </pre>

Figure 1: Common Operators

### 1.4.1 Boolean

The keywords `true` and `false` are used a lot in Boolean expressions, which could be converted to integers by the C compiler.

Take the following example:

```

#include <stdio.h>

int main(void)
{
    printf("%d\n%d\n", true, false);
}

```

Output:

```

1
0

```

True = 1

False = 0

The keyword `true` could be used in a `while` loop, `while(true)`, which causes the loop to run throughout the entire runtime, unless there is a call to exit the loop, such as `break`.

Also, the double equals `==` is used to compare if two values are the same or not. This will further be clarified in the example below which is retrieved from the cppreference.

#### 1.4.1.1 Logical operators

The following table shows the logical operators used in the C language.

Operator	Operator name
<code>!</code>	logical NOT
<code>&amp;&amp;</code>	logical AND
<code>  </code>	logical OR

Table 1: Logical operators

In addition to that table, the `!` is used before an expression to express its inverse e.g. `if(!a)` which means if not `a`.

#### Examples

```
if (x > 0 && y < 10) {
    // Do something if both conditions are true
}

if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
    // Do something if the character is a vowel
}

if (!flag) {
    // Do something if the flag is false
}
```

Take a look at the following example:

```
#include <stdbool.h>
#include <stdio.h>
```

```
#include <ctype.h>
int main(void)
{
    bool b = !(2+2 == 4); // not true
    printf("!(2+2==4) = %s\n", b ? "true" : "false");
    int n = isspace('a'); // zero if 'a' is a space, nonzero otherwise
    int x = !!n; // "bang-bang", common C idiom for mapping integers to [0,1]
    // (all non-zero values become 1)
    char *a[2] = {"nonspace", "space"};
    printf("%s\n", a[x]); // now x can be safely used as an index to array of 2
    ints
}
```

Output:

```
!(2+2==4) = false
nonspace
```

The standalone expression `2+2 == 4` evaluates to true, as `2+2` is 4 and the compiler compares 4 with 4, which is true.

But adding `!` before the expression means `!true`, which evaluates to false.

Take a look at `int n` and `int x`, `int x` is assigned the `isspace('a')`, this function checks if the parameter of it is a space, if it is, then return a nonzero value, if not, return 0, which as stated above, it is equivalent to false.

Using double `!!` is just saying NOT NOT, and NOT NOT is just equivalent to the original expression. But it is also used to make all nonzero values = 1. It is useful for mapping integers in binary values, 0 or 1.

#### 1.4.1.2 Arithmetic Expressions

While more straightforward than Boolean, it still has some concepts to be aware of.

Take a look at these examples with explanations:

```
#include <stdio.h>

int main() {
    int a = 10;
    int b = 3;

    // Addition: a + b
    int addition = a + b;
    printf("Addition: %d + %d = %d\n", a, b, addition); // Output: Addition: 10 + 3 = 13

    // Subtraction: a - b
    int subtraction = a - b;
    printf("Subtraction: %d - %d = %d\n", a, b, subtraction); // Output: Subtraction: 10 - 3 = 7

    // Multiplication: a * b
    int multiplication = a * b;
    printf("Multiplication: %d * %d = %d\n", a, b, multiplication); // Output: Multiplication: 10 * 3 = 30

    // Division: a / b
    int division = a / b;
    printf("Division: %d / %d = %d\n", a, b, division); // Output: Division: 10 / 3 = 3

    // Modulus: a % b
    int modulus = a % b;
    printf("Modulus: %d %% %d = %d\n", a, b, modulus); // Output: Modulus: 10 % 3 = 1

    // Bitwise NOT: ~a
    int bitwise_not_a = ~a;
    printf("Bitwise NOT: ~%d = %d\n", a, bitwise_not_a); // Output: Bitwise NOT: ~10 = -11

    // Bitwise AND: a & b
    int bitwise_and = a & b;
    printf("Bitwise AND: %d & %d = %d\n", a, b, bitwise_and); // Output: Bitwise AND: 10 & 3 = 2

    // Bitwise OR: a | b
    int bitwise_or = a | b;
    printf("Bitwise OR: %d | %d = %d\n", a, b, bitwise_or); // Output: Bitwise OR: 10 | 3 = 11

    // Bitwise XOR: a ^ b
    int bitwise_xor = a ^ b;
```

```
    printf("Bitwise XOR: %d ^ %d = %d\n", a, b, bitwise_xor); // Output: Bitwise XOR: 10 <
    // Left Shift: a << b
    int left_shift = a << b;
    printf("Left Shift: %d << %d = %d\n", a, b, left_shift); // Output: Left Shift: 10 <
    // Right Shift: a >> b
    int right_shift = a >> b;
    printf("Right Shift: %d >> %d = %d\n", a, b, right_shift); // Output: Right Shift: 1
    return 0;
}
```

Explanation:

- Addition ( $a + b$ ):  $10 + 3 = 13$
- Subtraction ( $a - b$ ):  $10 - 3 = 7$
- Multiplication ( $a * b$ ):  $10 * 3 = 30$
- Division ( $a / b$ ):  $10 / 3 = 3$  (integer division, truncating the fractional part)
- Modulus ( $a \% b$ ):  $10 \% 3 = 1$  (remainder of 10 divided by 3)
- Bitwise NOT ( $\sim a$ ):  $\sim 10 = -11$  (bitwise inversion of 10)
- Bitwise AND ( $a \& b$ ):  $10 \& 3 = 2$  (bitwise AND of 10 and 3)
- Bitwise OR ( $a | b$ ):  $10 | 3 = 11$  (bitwise OR of 10 and 3)
- Bitwise XOR ( $a \wedge b$ ):  $10 \wedge 3 = 9$  (bitwise XOR of 10 and 3)
- Left Shift ( $a \ll b$ ):  $10 \ll 3 = 80$  (left shift of 10 by 3 positions, equivalent to multiplying by  $2^3 = 8$ )
- Right Shift ( $a \gg b$ ):  $10 \gg 3 = 1$  (right shift of 10 by 3 positions, equivalent to integer division by  $2^3 = 8$ )

In addition to those expressions, there are also incremental and decremental.

```
#include <iostream>

int main() {
    int a = 10;
    // Post-increment: a++
    int post_increment = a++;
    std::cout << "Post-increment: a++ = " << post_increment << ", a = " << a << std::endl;
    // Post-decrement: a--
    int post_decrement = a--;
    std::cout << "Post-decrement: a-- = " << post_decrement << ", a = " << a << std::endl;
    // Pre-increment: ++a
    int pre_increment = ++a;
    // Pre-decrement: --a
    int pre_decrement = --a;
    std::cout << "Pre-decrement: --a = " << pre_decrement << ", a = " << a << std::endl;
    return 0;
}
```

Explanation:

- Post-increment (**a++**): The value of **a** is first used in the expression, and then it is incremented by 1.
- Post-decrement (**a--**): The value of **a** is first used in the expression, and then it is decremented by 1.
- Pre-increment (**++a**): The value of **a** is incremented by 1 first, and then it is used in the expression.
- Pre-decrement (**--a**): The value of **a** is decremented by 1 first, and then it is used in the expression.

It is important to understand the difference between **a++** and **++a**, in **for** loops for example, if you want to iterate through an array's contents, and you don't know the difference, you could end up over iterating the array and getting out-of-bounds error.

### 1.4.2 Assignment Operators

Operator	Operator name	Example	Description	Equivalent of
=	basic assignment	a = b	a becomes equal to b	N/A
+=	addition assignment	a += b	a becomes equal to the addition of a and b	a = a + b
-=	subtraction assignment	a -= b	a becomes equal to the subtraction of b from a	a = a - b
*=	multiplication assignment	a *= b	a becomes equal to the product of a and b	a = a * b
/=	division assignment	a /= b	a becomes equal to the division of a by b	a = a / b
%=	modulo assignment	a %= b	a becomes equal to the remainder of a divided by b	a = a % b
&=	bitwise AND assignment	a &= b	a becomes equal to the bitwise AND of a and b	a = a & b
=	bitwise OR assignment	a  = b	a becomes equal to the bitwise OR of a and b	a = a   b
^=	bitwise XOR assignment	a ^= b	a becomes equal to the bitwise XOR of a and b	a = a ^ b
<<=	bitwise left shift assignment	a <<= b	a becomes equal to a left shifted by b	a = a << b
>>=	bitwise right shift assignment	a >>= b	a becomes equal to a right shifted by b	a = a >> b

Figure 2: Assignment of operators

### 1.4.3 Ternary Operator

```
#include <iostream>

int main()
{
    int x = 10;
    int y = 5;
    // Ternary operator: condition ? expression1 : expression2
    // If x is greater than y, assign "x is greater" to result, otherwise assign "y is greater"
    std::string result = (x > y) ? "x is greater" : "y is greater";
    std::cout << result << std::endl; // Output depends on the values of x and y
    return 0;
}
```

### 1.4.4 Operator precedence

In C programming, operators have different precedences, which determine the order in which they are evaluated within expressions. Here's a list of C operator precedences, from highest

to lowest:

1. Parentheses `()` - Highest precedence. Operations within parentheses are evaluated first.
2. Postfix increment/decrement `++`, `--` - Next highest precedence. Increment or decrement operations are performed after the variable is used.
3. Prefix increment/decrement `++`, `--` - Increment or decrement operations are performed before the variable is used.
4. Unary operators `+`, `-`, `!`, `~`, `sizeof`, `*`, `&`, `++`, `--` (as prefix) - Unary operators are applied to a single operand.
5. Multiplication, division, modulus `*`, `/`, `%` - Multiplication, division, and modulus operations are performed next.
6. Addition, subtraction `+`, `-` - Addition and subtraction operations are performed next.
7. Bitwise shift `<<`, `>>` - Bitwise left and right shift operations are performed next.
8. Relational operators `<`, `<=`, `>`, `>=` - Comparison operations are performed next.
9. Equality operators `==`, `!=` - Equality operations are performed next.
10. Bitwise AND `&` - Bitwise AND operation is performed next.
11. Bitwise XOR `^` - Bitwise XOR operation is performed next.
12. Bitwise OR `|` - Bitwise OR operation is performed next.
13. Logical AND `&&` - Logical AND operation is performed next.
14. Logical OR `||` - Logical OR operation is performed next.
15. Conditional operator `?:` - Ternary conditional operator is performed next.
16. Assignment operators `=`, `+=`, `-=`, etc. - Assignment operations are performed next.
17. Comma operator `,` - Lowest precedence. Comma operator evaluates both its operands from left to right and returns the value of the right operand.



## 1.5 Statements

### 1.5.1 Assignment Statement

An Assignment statement is a statement that is used to set a value to the variable name in a program. Assignment statement allows a variable to hold different types of values during its program lifespan.

**Syntax:** The symbol used in an assignment statement is called as an operator. The symbol is '='. Note: The Assignment Operator should never be used for Equality purpose which is double equal sign '=='. The Basic Syntax of Assignment Statement in a programming language is:

$$\text{variable} = \text{expression};$$

where,

- **variable** = variable name
- **expression** = it could be either a direct value or a math expression/formula or a function call

Few programming languages such as Java, C, C++ require data type to be specified for the variable, so that it is easy to allocate memory space and store those values during program execution.

$$\text{data\_type variable\_name} = \text{value};$$

**Example:**

```
int a = 50;
float b;
a = 25;
b = 34.25;
```

In the above-given examples, Variable 'a' is assigned a value in the same statement as per its defined data type. A data type is only declared for Variable 'b'. In the 3rd line of code,

Variable 'a' is reassigned the value 25. The 4th line of code assigns the value for Variable 'b'.

### Assignment Statement Forms

1. **Basic Form:** This is one of the most common forms of Assignment Statements. Here the Variable name is defined, initialized, and assigned a value in the same statement. This form is generally used when we want to use the Variable quite a few times and we do not want to change its value very frequently.

```
int RollNo = 25 ;  
printf("%d",RollNo);
```

Output:

25

2. **Tuple Assignment:** Generally, we use this form when we want to define and assign values for more than 1 variable at the same time. This saves time and is an easy method. Note that here every individual variable has a different value assigned to it.
3. **Sequence Assignment:**

```
x,y,z = 'HEY' ;  
printf('x = ', x) ;  
printf('y = ', y) ;  
printf('z = ', z) ;
```

Output:

```
x = H;  
y = E;  
z = Y;
```

#### 4. Multiple-target Assignment or Chain Assignment:

```
a = b = 40 ;  
print(a, b) ;
```

Output:

```
40 40
```

#### 5. Augmented Assignment:

```
speed = 40 ;  
speed += 10 ;  
print (\Speed = ", speed) ;
```

Output:

```
Speed = 50
```

### 1.5.2 Declaration of Variables

Variables are the basic unit of storage in a programming language. These variables consist of a data type, the variable name, and the value to be assigned to the variable. Unless and until the variables are declared and initialized, they cannot be used in the program. Let us learn more about the Declaration and Initialization of Variables in this article below.

**What is Declaration and Initialization?**

- **Declaration** of a variable in a computer programming language is a statement used to specify the variable name and its data type. Declaration tells the compiler about the existence of an entity in the program and its location. When you declare a variable, you should also initialize it.
- **Initialization** is the process of assigning a value to the Variable. Every programming language has its own method of initializing the variable. If the value is not assigned to the Variable, then the process is only called a Declaration.

**Basic Syntax:** The basic form of declaring a variable is:

type identifier [= value] [, identifier [= value]]...;

OR

data\_type variable\_name = value;

where,

- **type** = Data type of the variable
- **identifier** = Variable name
- **value** = Data to be stored in the variable (Optional field)

### Rules to Declare and Initialize Variables

1. Variable names must begin with a letter, underscore, non-number character. Each language has its own conventions.
2. Few programming languages like PHP, Python, Perl, etc. do not require specifying the data type at the start.
3. Always use the '=' sign to initialize a value to the Variable.
4. Do not use a comma with numbers.
5. Once a data type is defined for the variable, then only that type of data can be stored in it. For example, if a variable is declared as Int, then it can only store integer values.

6. A variable name once defined can only be used once in the program. You cannot define it again to store another type of value.
7. If another value is assigned to the variable which already has a value assigned to it before, then the previous value will be overwritten by the new value.

### Types of Initialization

1. **Static Initialization** – In this method, the variable is assigned a value in advance. Here, the values are assigned in the declaration statement. Static Initialization is also known as Explicit Initialization.

```
int a;  
a = 5;  
int b = 10;  
int x = 4, y = 5;
```

2. **Dynamic Initialization** – In this method, the variable is assigned a value at the run-time. The value is either assigned by the function in the program or by the user at the time of running the program. The value of these variables can be altered every time the program runs. Dynamic Initialization is also known as Implicit Initialization.

```
int speed;  
printf("Enter the value of speed");  
scanf("%d", &speed);
```

### 1.5.3 Return Statement

A Return statement is a statement in a programming language that is used to terminate the execution of a function and return a value to the calling code. It is an essential part of functions or methods, allowing them to produce results that can be utilized elsewhere in the program.

**Syntax:**

```
int function_name(parameters){  
    // Function code  
    return expression;  
}
```

**Explanation:**

- **function\_name:** The name of the function.
- **parameters:** The input parameters the function takes (if any).
- **expression:** The value to be returned by the function.

**Example:**

```
int add_numbers(x, y):  
sum_result = x + y  
return sum_result
```

In the example above, the `add_numbers` function takes two parameters (`x` and `y`), adds them, and returns the result. The `return` statement is used to send the result back to the calling code, and it is then printed.

Return statements are crucial for encapsulating functionality in functions and enabling modular and readable code. They allow values to be passed from one part of the program to another, enhancing code organization and reusability.

#### 1.5.4 Iterative Statement

Iterative statements are used to create loops, allowing a set of instructions to be repeatedly executed as long as a certain condition is met. The primary iterative statements are:

1. **for loop:**

Executes `init-statement` once, then executes `statement` and `iteration-expression` repeatedly, until the value of `condition` becomes false. The test takes place before each iteration.

**Syntax:**

attr-spec-seq for ( init clause ; condexpression ; iterationexpression )

loop-statement

**Explanation:**

Behaves as follows:

- init-clause may be an expression or a declaration.
- An init-clause, which is an expression, is evaluated once before the first evaluation of cond-expression and its result is discarded.
- An init-clause, which is a declaration, is in scope in the entire loop body, including the remainder of init-clause, the entire cond-expression, the entire iteration-expression, and the entire loop-statement. Only auto and register storage class specifiers are allowed for the variables declared in this declaration.
- cond-expression is evaluated before the loop body. If the result of the expression is zero, the loop statement is exited immediately.
- iteration-expression is evaluated after the loop body, and its result is discarded. After evaluating iteration-expression, control is transferred to cond-expression.

init-clause, cond-expression, and iteration-expression are all optional. If cond-expression is omitted, it is replaced with a non-zero integer constant, which makes the loop endless:

```
for(;;) {  
    printf("endless loop!");  
}
```

loop-statement is not optional, but it may be a null statement:

```
for(int n = 0; n < 10; ++n, printf("%d\n", n))  
    ; // null statement
```

If the execution of the loop needs to be terminated at some point, a `break` statement can be used anywhere within the loop-statement. The `continue` statement used anywhere within the loop-statement transfers control to iteration-expression.

A program with an endless loop has undefined behavior if the loop has no observable behavior (I/O, volatile accesses, atomic or synchronization operation) in any part of its cond-expression, iteration-expression, or loop-statement. This allows the compilers to optimize out all unobservable loops without proving that they terminate. The only exceptions are the loops where cond-expression is omitted or is a constant expression; `for(;;)` is always an endless loop.

As with all other selection and iteration statements, the `for` statement establishes block scope: any identifier introduced in the init-clause, cond-expression, or iteration-expression goes out of scope after the loop-statement.

`attr-spec-seq` is an optional list of attributes, applied to the `for` statement.

## 2. **while loop:**

Executes a statement repeatedly, until the value of expression becomes equal to zero. The test takes place before each iteration.

### **Syntax:**

```
attr-spec-seq(optional) while ( expression ) statement
```

### **Explanation:**

A `while` statement causes the statement (also called the loop body) to be executed repeatedly until the expression (also called controlling expression) compares equal to zero. The repetition occurs regardless of whether the loop body is entered normally or by a `go-to` into the middle of the statement.



The evaluation of the expression takes place before each execution of the statement (unless entered by a go-to). If the controlling expression needs to be evaluated after the loop body, the do-while loop may be used.

If the execution of the loop needs to be terminated at some point, a break statement can be used as a terminating statement. If the execution of the loop needs to be continued at the end of the loop body, the continue statement can be used as a shortcut.

A program with an endless loop has undefined behavior if the loop has no observable behavior (I/O, volatile accesses, atomic or synchronization operation) in any part of its statement or expression. This allows the compilers to optimize out all unobservable loops without proving that they terminate. The only exceptions are the loops where the expression is a constant expression; while(true) is always an endless loop.

As with all other selection and iteration statements, the while statement establishes block scope: any identifier introduced in the expression goes out of scope after the statement.

### 3. do-while loop:

Executes a statement repeatedly until the value of the condition expression becomes false. The test takes place after each iteration.

#### **Syntax:**

```
attr-spec-seq(optional) do statement while ( expression ) ;
```

#### **Explanation:**

A do-while statement causes the statement (also called the loop body) to be executed repeatedly until the expression (also called controlling expression) compares equal to 0. The repetition occurs regardless of whether the loop body is entered normally or by a goto into the middle of the statement.

The evaluation of the expression takes place after each execution of the statement (whether entered normally or by a goto). If the controlling expression needs to be

evaluated before the loop body, the while loop or the for loop may be used.

If the execution of the loop needs to be terminated at some point, a break statement can be used as a terminating statement. If the execution of the loop needs to be continued at the end of the loop body, the continue statement can be used as a shortcut.

A program with an endless loop has undefined behavior if the loop has no observable behavior (I/O, volatile accesses, atomic or synchronization operation) in any part of its statement or expression. This allows the compilers to optimize out all unobservable loops without proving that they terminate. The only exceptions are the loops where the expression is a constant expression; `do ... while(true);` is always an endless loop.

As with all other selection and iteration statements, the do-while statement establishes block scope: any identifier introduced in the expression goes out of scope after the statement.

### 1.5.5 if statement & if-else statement

Conditionally executes code. Used where code needs to be executed only if some condition is true.

#### Syntax

attr-spec-seq(optional) if ( expression ) statement-true

attr-spec-seq(optional) if ( expression ) statement-true else statement-false

#### Explanation

- **expression** must be an expression of any scalar type.
- If **expression** compares not equal to the integer zero, **statement-true** is executed.
- In the form, if **expression** compares equal to the integer zero, **statement-false** is executed.

### 1.5.6 switch statement

Executes code according to the value of an integral argument. Used where one or several out of many branches of code need to be executed according to an integral value.

#### Syntax

attr-spec-seq(optional) switch ( expression ) statement

#### Explanation

- The body of a switch statement may have an arbitrary number of **case:** labels, as long as the values of all **constant-expressions** are unique (after conversion to the promoted type of **expression**).
- At most one **default:** label may be present.
- If **expression** evaluates to the value that is equal to the value of one of **constant-expressions** after conversion to the promoted type of **expression**, then control is transferred to the statement that is labeled with that **constant-expression**.
- If **expression** evaluates to a value that doesn't match any of the **case:** labels, and the **default:** label is present, control is transferred to the statement labeled with the **default:** label.
- If **expression** evaluates to a value that doesn't match any of the **case:** labels, and the **default:** label is not present, none of the switch body is executed.

## 2 Lexer

### 2.1 Defined Tokens

---

#### Defined Tokens

---

1. NUMBER - Matches various forms of numbers, including integers and floats.
  - Pattern: `[+-]?(?:0[bB] [01]+|0[xX] [0-9a-fA-F]+|0[0-7]*|[1-9] [0-9]*|0(?:\.\. [0-9]+)?(?:[eE] [-+]?[0-9]+)?`

2. COMPARISON\_OP - Identifies comparison operators.
  - Pattern: `==|!=|>|=|<|=|<&&|\\|\\|`
3. COMPOUND\_OP - Matches compound assignment operators.
  - Pattern: `-=|\\+=|\\*=|/=|%|=|<|=|>|=|&|=|\\^|=|\\|=`
4. ASSIGNMENT\_OP - Recognizes assignment operator.
  - Pattern: `=`
5. TERNARY\_OP - Matches the ternary operator.
  - Pattern: `\\?`
6. COLON - Identifies colon punctuation.
  - Pattern: `:`
7. SHIFT\_OP - Matches shift operators.
  - Pattern: `<<|>>`
8. UNARY\_OP - Identifies unary operators.
  - Pattern: `\\+|\\+|--`
9. ARITHMETIC\_OP - Recognizes arithmetic operators.
  - Pattern: `[+|\\-|*|/|%]`
10. BITWISE\_OP - Identifies bitwise operators.
  - Pattern: `[<>|&|^]`
11. RELATIONAL\_OP - Matches relational operators.
  - Pattern: `[<>]`
12. LEFT\_PAREN - Recognizes left parenthesis.
  - Pattern: `\\(`
13. RIGHT\_PAREN - Matches right parenthesis.
  - Pattern: `\\)`
14. LEFT\_BRACKET - Identifies left square bracket.
  - Pattern: `\\[`
15. RIGHT\_BRACKET - Matches right square bracket.
  - Pattern: `\\]`
16. LEFT\_BRACE - Recognizes left curly brace.

- Pattern: `\{`
- 17. RIGHT\_BRACE - Matches right curly brace.
  - Pattern: `\}`
- 18. SEMICOLON - Identifies semicolon punctuation.
  - Pattern: `;`
- 19. COMMA - Matches comma punctuation.
  - Pattern: `,`
- 20. DOT - Identifies dot punctuation.
  - Pattern: `\.`
- 21. TYPE\_SPECIFIER - Recognizes C type specifiers.
  - Pattern: `int|float|char|double|short|long|signed|unsigned|void|String|bool`
- 22. KEYWORD - Matches C keywords.
  - Pattern:
    - ↪ `if|while|for|switch|case|default|break|continue|goto|sizeof|typedef|extern`
    - `|static|const|volatile|return|auto|struct|enum|do|else`
- 23. STRING\_LITERAL - Identifies string literals.
  - Pattern: `"\"[^\\"\\r\\n]*\""`
- 24. CHARACTER\_LITERAL - Matches character literals.
  - Pattern: `'\'.?'"`
- 25. ID - Recognizes identifiers.
  - Pattern: `[a-zA-Z_][a-zA-Z0-9_]*(:[a-zA-Z_][a-zA-Z0-9_]*)*`
- 26. PUNCTUATION - Matches various punctuation marks.
  - Pattern: `[(){};,,]`

## 2.2 Symbol Table

---

### Implementation Details

---

The symbol table is generated during lexical analysis to keep track of identifiers encountered in the code. Each identifier is associated with a unique index in the symbol table.

The symbol table is implemented as an ArrayList of Tokens, where each Token represents an identifier. The Token class has the following attributes:

TokenType enum representing the type of the token, which is ID for identifiers.  
String representing the actual identifier.  
Integer representing the unique index assigned to the identifier in the symbol table.

The process of building the symbol table involves iterating through the tokens generated  
→ by the lexical analyzer. When encountering an identifier token, the lexer checks if  
→ the identifier already exists in the symbol table. If it does, the index of the  
→ existing identifier is assigned to the token. If not, a new entry is added to the  
→ symbol table with a unique index.

The symbol table is printed after lexical analysis is complete. Each entry in the symbol  
→ table is printed in the format:

Symbol Table Token: <Type, Identifier> Index: Index

where Type is the type of the identifier token, Identifier is the actual identifier, and  
→ Index is the unique index assigned to the identifier.

---

## 2.3 Implementation Details

---

### Implementation Details

---

#### 1. Token Enumeration and Definition:

- The lexer starts by defining an enum named TokenType to represent various token  
→ types in C.
- Each token type is associated with a specific pattern defined using regular  
→ expressions to match the corresponding token in the input.

#### 2. Token Class:

- Within the lexer, there exists a nested class called Token to encapsulate details  
→ about each identified token.
- The Token class consists of type, data, index, and isFloat attributes, representing  
→ the token type, matched data, index (for identifiers), and whether it's a float  
→ value respectively.

#### 3. Tokenization Process:

- The tokenize() method orchestrates the tokenization of the input source code.
- It constructs a regex pattern dynamically by combining patterns for all token types  
→ defined in TokenType.

- A Matcher traverses the input using the compiled regex pattern, identifying tokens  
↳ based on specified patterns.

#### 4. Handling Comments:

- The lexer incorporates logic to handle single-line and multi-line comments in the  
↳ source code.
- Single-line comments starting with `//` are removed, while multi-line comments  
↳ enclosed within `/* */` are appropriately managed.

#### 5. Skipping Whitespace:

- Whitespace characters are disregarded during tokenization to prevent unnecessary  
↳ tokens.

#### 6. Iterative Tokenization:

- Tokenization occurs iteratively, line by line, ensuring accurate identification of  
↳ tokens within each line.
- This approach facilitates efficient tokenization and preserves the structure of the  
↳ source code.

#### 7. Error Handling:

- Although the primary focus is on token identification, error handling mechanisms  
↳ can be implemented to detect and report lexical errors.

#### 8. File Input and Output:

- The lexer reads input from a specified source code file using Java's file handling  
↳ capabilities.
- After tokenization, the lexer can output the identified tokens for further analysis  
↳ or processing.

## 2.4 Lexer Test Cases

### 2.4.1 Test Case 1

#### 2.4.1.1 Test Case Code

---

##### Test Case Code

---

```
#include <iostream>
#include <iostream>

int i,choice;
char c,ch;
```

```
enum Weekday {
    MONDAY,
    TUESDAY ,
    wednesday ,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};

int factorial(int n) {
    if (n == 0) {
        return 1;
    }else {
        return 0 ;
    }
}

int main()
{
    int x ;
    if ( x == 1){
        x = x + 1 ;
    }
    // comment
    for(int i = 4; i < 3; i++)
    {
        x = x + 1 ;
    }
}
```

#### 2.4.1.2 Test Case Output

---

Test Case Output

```
Token: <TYPE_SPECIFIER , int>
Token: <ID , (0, i)>
Token: <COMMA , ,>
Token: <ID , (1, choice)>
Token: <SEMICOLON , ;>
Token: <TYPE_SPECIFIER , char>
Token: <ID , (2, c)>
Token: <COMMA , ,>
Token: <ID , (3, ch)>
```



```
Token: <SEMICOLON , ;>
Token: <KEYWORD , enum>
Token: <ID , (4, Weekday)>
Token: <LEFT_BRACE , {>
Token: <ID , (5, MONDAY)>
Token: <COMMA , ,>
Token: <ID , (6, TUESDAY)>
Token: <COMMA , ,>
Token: <ID , (7, wednesday)>
Token: <COMMA , ,>
Token: <ID , (8, THURSDAY)>
Token: <COMMA , ,>
Token: <ID , (9, FRIDAY)>
Token: <COMMA , ,>
Token: <ID , (10, SATURDAY)>
Token: <COMMA , ,>
Token: <ID , (11, SUNDAY)>
Token: <RIGHT_BRACE , }>
Token: <SEMICOLON , ;>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (12, factorial)>
Token: <LEFT_PAREN , (>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (13, n)>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <KEYWORD , if>
Token: <LEFT_PAREN , (>
Token: <ID , (13, n)>
Token: <COMPARISON_OP , ==>
Token: <NUMBER , INTEGER value 0>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <KEYWORD , return>
Token: <NUMBER , INTEGER value 1>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Token: <KEYWORD , else>
Token: <LEFT_BRACE , {>
Token: <KEYWORD , return>
Token: <NUMBER , INTEGER value 0>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Token: <RIGHT_BRACE , }>
```

```
Token: <TYPE_SPECIFIER , int>
Token: <ID , (14, main)>
Token: <LEFT_PAREN , (>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (15, x)>
Token: <SEMICOLON , ;>
Token: <KEYWORD , if>
Token: <LEFT_PAREN , (>
Token: <ID , (15, x)>
Token: <COMPARISON_OP , ==>
Token: <NUMBER , INTEGER value 1>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <ID , (15, x)>
Token: <ASSIGNMENT_OP , =>
Token: <ID , (15, x)>
Token: <ARITHMETIC_OP , +>
Token: <NUMBER , INTEGER value 1>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Token: <KEYWORD , for>
Token: <LEFT_PAREN , (>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (0, i)>
Token: <ASSIGNMENT_OP , =>
Token: <NUMBER , INTEGER value 4>
Token: <SEMICOLON , ;>
Token: <ID , (0, i)>
Token: <COMPARISON_OP , <>
Token: <NUMBER , INTEGER value 3>
Token: <SEMICOLON , ;>
Token: <ID , (0, i)>
Token: <UNARY_OP , ++>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <ID , (15, x)>
Token: <ASSIGNMENT_OP , =>
Token: <ID , (15, x)>
Token: <ARITHMETIC_OP , +>
Token: <NUMBER , INTEGER value 1>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
```

```
Token: <RIGHT_BRACE , }>
Symbol Table Token: <ID , i> Index: 0
Symbol Table Token: <ID , choice> Index: 1
Symbol Table Token: <ID , c> Index: 2
Symbol Table Token: <ID , ch> Index: 3
Symbol Table Token: <ID , Weekday> Index: 4
Symbol Table Token: <ID , MONDAY> Index: 5
Symbol Table Token: <ID , TUESDAY> Index: 6
Symbol Table Token: <ID , wednesday> Index: 7
Symbol Table Token: <ID , THURSDAY> Index: 8
Symbol Table Token: <ID , FRIDAY> Index: 9
Symbol Table Token: <ID , SATURDAY> Index: 10
Symbol Table Token: <ID , SUNDAY> Index: 11
Symbol Table Token: <ID , factorial> Index: 12
Symbol Table Token: <ID , n> Index: 13
Symbol Table Token: <ID , main> Index: 14
Symbol Table Token: <ID , x> Index: 15
Parsing complete.
```

---

## 2.4.2 Test Case 2

### 2.4.2.1 Test Case Code

---

```
int main() {
    int a = 10;
    float b_var = 3.14;
    char initial = 'A';

    if (a > b_var) {
        printf("Integer is larger\n");
    } else {
        printf("Float is larger or equal\n");
    }

    // Comments (single line and multi-line)
    // single line comment

    /* multi-line comment
       multiple lines */

    char message[] = "Hello, world!";
    printf("%s\n", message);

    return 0;
}
```

---

### 2.4.2.2 Test Case Output

---

```
Token: <TYPE_SPECIFIER , int>
Token: <ID , (0, main)>
Token: <LEFT_PAREN , (>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (1, a)>
Token: <ASSIGNMENT_OP , =>
Token: <NUMBER , INTEGER value 10>
Token: <SEMICOLON , ;>
```

---

```
Token: <TYPE_SPECIFIER , float>
Token: <ID , (2, b_var)>
Token: <ASSIGNMENT_OP , =>
Token: <NUMBER , INTEGER value 3.14>
Token: <SEMICOLON , ;>
Token: <TYPE_SPECIFIER , char>
Token: <ID , (3, initial)>
Token: <ASSIGNMENT_OP , =>
Token: <CHARACTER_LITERAL , 'A'>
Token: <SEMICOLON , ;>
Token: <KEYWORD , if>
Token: <LEFT_PAREN , (>
Token: <ID , (1, a)>
Token: <COMPARISON_OP , >>
Token: <ID , (2, b_var)>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <ID , (4, printf)>
Token: <LEFT_PAREN , (>
Token: <STRING_LITERAL , "Integer is larger\n">
Token: <RIGHT_PAREN , )>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Token: <KEYWORD , else>
Token: <LEFT_BRACE , {>
Token: <ID , (4, printf)>
Token: <LEFT_PAREN , (>
Token: <STRING_LITERAL , "Float is larger or equal\n">
Token: <RIGHT_PAREN , )>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Token: <TYPE_SPECIFIER , char>
Token: <ID , (5, message)>
Token: <LEFT_BRACKET , [>
Token: <RIGHT_BRACKET , ]>
Token: <ASSIGNMENT_OP , =>
Token: <STRING_LITERAL , "Hello, world!">
Token: <SEMICOLON , ;>
Token: <ID , (4, printf)>
Token: <LEFT_PAREN , (>
Token: <STRING_LITERAL , "%s\n">
Token: <COMMA , ,>
Token: <ID , (5, message)>
Token: <RIGHT_PAREN , )>
```

```
Token: <SEMICOLON , ;>
Token: <KEYWORD , return>
Token: <NUMBER , INTEGER value 0>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Symbol Table Token: <ID , main> Index: 0
Symbol Table Token: <ID , a> Index: 1
Symbol Table Token: <ID , b_var> Index: 2
Symbol Table Token: <ID , initial> Index: 3
Symbol Table Token: <ID , printf> Index: 4
Symbol Table Token: <ID , message> Index: 5
```

---

## 2.4.3 Test Case 3

### 2.4.3.1 Test Case Code

---

#### Test Case Code

---

```
#include <stdio.h>

int main() {
    int arr[] = {5, 12, 3, 21, 8, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] < arr[j + 1]) {

                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    printf("Array in descending order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

### 2.4.3.2 Test Case Output

---

#### Test Case Output

---

```
Token: <TYPE_SPECIFIER , int>
Token: <ID , (0, main)>
Token: <LEFT_PAREN , (>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
```

```
Token: <RIGHT_BRACKET , ]>
Token: <ASSIGNMENT_OP , =>
Token: <LEFT_BRACE , {>
Token: <NUMBER , INTEGER value 5>
Token: <COMMA , ,>
Token: <NUMBER , INTEGER value 12>
Token: <COMMA , ,>
Token: <NUMBER , INTEGER value 3>
Token: <COMMA , ,>
Token: <NUMBER , INTEGER value 21>
Token: <COMMA , ,>
Token: <NUMBER , INTEGER value 8>
Token: <COMMA , ,>
Token: <NUMBER , INTEGER value 10>
Token: <RIGHT_BRACE , }>
Token: <SEMICOLON , ;>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (2, n)>
Token: <ASSIGNMENT_OP , =>
Token: <KEYWORD , sizeof>
Token: <LEFT_PAREN , (>
Token: <ID , (1, arr)>
Token: <RIGHT_PAREN , )>
Token: <ARITHMETIC_OP , />
Token: <KEYWORD , sizeof>
Token: <LEFT_PAREN , (>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
Token: <NUMBER , INTEGER value 0>
Token: <RIGHT_BRACKET , ]>
Token: <RIGHT_PAREN , )>
Token: <SEMICOLON , ;>
Token: <KEYWORD , for>
Token: <LEFT_PAREN , (>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (3, i)>
Token: <ASSIGNMENT_OP , =>
Token: <NUMBER , INTEGER value 0>
Token: <SEMICOLON , ;>
Token: <ID , (3, i)>
Token: <COMPARISON_OP , <>
Token: <ID , (2, n)>
Token: <ARITHMETIC_OP , ->
Token: <NUMBER , INTEGER value 1>
```



```
Token: <SEMICOLON , ;>
Token: <ID , (3, i)>
Token: <UNARY_OP , ++>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <KEYWORD , for>
Token: <LEFT_PAREN , (>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (4, j)>
Token: <ASSIGNMENT_OP , =>
Token: <NUMBER , INTEGER value 0>
Token: <SEMICOLON , ;>
Token: <ID , (4, j)>
Token: <COMPARISON_OP , <>
Token: <ID , (2, n)>
Token: <ARITHMETIC_OP , ->
Token: <ID , (3, i)>
Token: <ARITHMETIC_OP , ->
Token: <NUMBER , INTEGER value 1>
Token: <SEMICOLON , ;>
Token: <ID , (4, j)>
Token: <UNARY_OP , ++>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <KEYWORD , if>
Token: <LEFT_PAREN , (>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
Token: <ID , (4, j)>
Token: <RIGHT_BRACKET , ]>
Token: <COMPARISON_OP , <>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
Token: <ID , (4, j)>
Token: <ARITHMETIC_OP , +>
Token: <NUMBER , INTEGER value 1>
Token: <RIGHT_BRACKET , ]>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (5, temp)>
Token: <ASSIGNMENT_OP , =>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
```

```
Token: <ID , (4, j)>
Token: <RIGHT_BRACKET , ]>
Token: <SEMICOLON , ;>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
Token: <ID , (4, j)>
Token: <RIGHT_BRACKET , ]>
Token: <ASSIGNMENT_OP , =>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
Token: <ID , (4, j)>
Token: <ARITHMETIC_OP , +>
Token: <NUMBER , INTEGER value 1>
Token: <RIGHT_BRACKET , ]>
Token: <SEMICOLON , ;>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
Token: <ID , (4, j)>
Token: <ARITHMETIC_OP , +>
Token: <NUMBER , INTEGER value 1>
Token: <RIGHT_BRACKET , ]>
Token: <ASSIGNMENT_OP , =>
Token: <ID , (5, temp)>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Token: <RIGHT_BRACE , }>
Token: <RIGHT_BRACE , }>
Token: <ID , (6, printf)>
Token: <LEFT_PAREN , (>
Token: <STRING_LITERAL , "Array in descending order: ">
Token: <RIGHT_PAREN , )>
Token: <SEMICOLON , ;>
Token: <KEYWORD , for>
Token: <LEFT_PAREN , (>
Token: <TYPE_SPECIFIER , int>
Token: <ID , (3, i)>
Token: <ASSIGNMENT_OP , =>
Token: <NUMBER , INTEGER value 0>
Token: <SEMICOLON , ;>
Token: <ID , (3, i)>
Token: <COMPARISON_OP , <>
Token: <ID , (2, n)>
Token: <SEMICOLON , ;>
Token: <ID , (3, i)>
```

```
Token: <UNARY_OP , ++>
Token: <RIGHT_PAREN , )>
Token: <LEFT_BRACE , {>
Token: <ID , (6, printf)>
Token: <LEFT_PAREN , (>
Token: <STRING_LITERAL , "%d ">
Token: <COMMA , ,>
Token: <ID , (1, arr)>
Token: <LEFT_BRACKET , [>
Token: <ID , (3, i)>
Token: <RIGHT_BRACKET , ]>
Token: <RIGHT_PAREN , )>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Token: <ID , (6, printf)>
Token: <LEFT_PAREN , (>
Token: <STRING_LITERAL , "\n">
Token: <RIGHT_PAREN , )>
Token: <SEMICOLON , ;>
Token: <KEYWORD , return>
Token: <NUMBER , INTEGER value 0>
Token: <SEMICOLON , ;>
Token: <RIGHT_BRACE , }>
Symbol Table Token: <ID , main> Index: 0
Symbol Table Token: <ID , arr> Index: 1
Symbol Table Token: <ID , n> Index: 2
Symbol Table Token: <ID , i> Index: 3
Symbol Table Token: <ID , j> Index: 4
Symbol Table Token: <ID , temp> Index: 5
Symbol Table Token: <ID , printf> Index: 6
```

---

#### 2.4.3.3 Test Case Output

---

## 2.4.4 Test Case 4

### 2.4.4.1 Test Case Code

---

#### Test Case Code

---

```
int main()
{
  int x ;
  if ( x == 1){
    x = x + 1
  }
```

### 2.4.4.2 Test Case Output

---

#### Test Case Output

---

```
Syntax error: Missing semicolon after assignment statement
```

---

## 2.4.5 Test Case 5

### 2.4.5.1 Test Case Code

---

#### Test Case Code

---

```
int factorial(int n)
{
  if (n == 0) {
    return 1;
  }else {
    return 0 ;
  }
}
```

### 2.4.5.2 Test Case Output

---

#### Test Case Output

---

```
Syntax error: Expecting '{' to start the compound statement
```

---

## 2.4.6 Lexer Test Cases Screenshots

### 2.4.6.1 Test Case 1 Screenshots

```
Token: <KEYWORD int>
Token: <ID main>
Token: <KEYWORD int>
Token: <ID i>
Token: <NUMBER 0>
Token: <KEYWORD int>
Token: <ID sum>
Token: <NUMBER 0>
Token: <KEYWORD while>
Token: <ID i>
Token: <REALOP <>
Token: <NUMBER 5>
Token: <ID sum>
Token: <BINARYOP +>
Token: <ID i>
Token: <ID i>
Token: <BINARYOP +>
Token: <BINARYOP +>
Token: <ID printf>
Token: <STRINGLITERAL "Sum from while loop: %d\n">
Token: <ID sum>
Token: <ID sum>
Token: <NUMBER 0>
Token: <KEYWORD for>
Token: <KEYWORD int>
Token: <ID j>
Token: <NUMBER 0>
Token: <ID j>
Token: <REALOP <>
Token: <NUMBER 5>
```

Figure 3: Screenshot 1 of Test Case 1 Output

```
Token: <ID j>
Token: <BINARYOP +>
Token: <BINARYOP +>
Token: <KEYWORD switch>
Token: <ID j>
Token: <KEYWORD case>
Token: <NUMBER 0>
Token: <KEYWORD case>
Token: <NUMBER 1>
Token: <ID sum>
Token: <BINARYOP +>
Token: <ID j>
Token: <KEYWORD break>
Token: <KEYWORD case>
Token: <NUMBER 2>
Token: <ID sum>
Token: <BINARYOP +>
Token: <NUMBER 2>
Token: <BINARYOP *>
Token: <ID j>
Token: <KEYWORD break>
Token: <KEYWORD default>
Token: <ID sum>
Token: <BINARYOP +>
Token: <ID j>
Token: <BINARYOP *>
Token: <NUMBER 2>
Token: <BINARYOP ->
Token: <NUMBER 1>
Token: <KEYWORD break>
Token: <ID printf>
```

Figure 4: Screenshot 2 of Test Case 1 Output

```
Token: <STRINGLITERAL "Sum from for loop with switch case: %d\n">  
Token: <ID sum>  
Token: <KEYWORD return>  
Token: <NUMBER 0>
```

Figure 5: Screenshot 3 of Test Case 1 Output

#### 2.4.6.2 Test Case 2 Screenshots

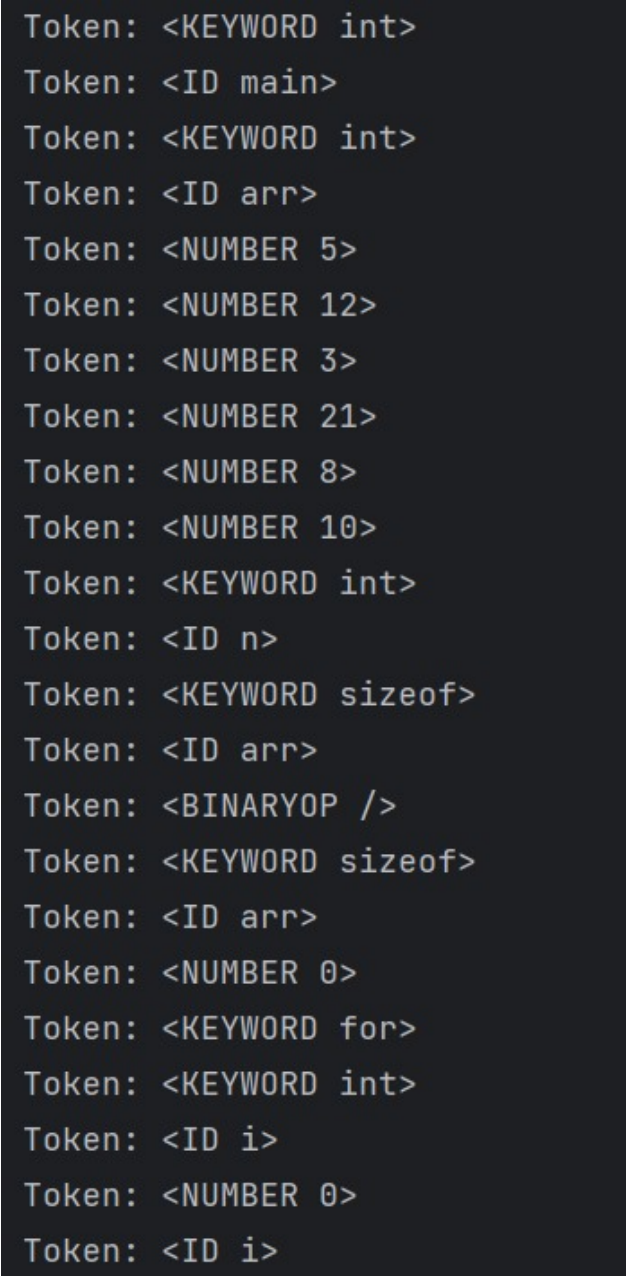
```
Token: <KEYWORD int>  
Token: <ID main>  
Token: <KEYWORD int>  
Token: <ID a>  
Token: <NUMBER 10>  
Token: <KEYWORD float>  
Token: <ID b_var>  
Token: <NUMBER 3>  
Token: <NUMBER 14>  
Token: <KEYWORD char>  
Token: <ID initial>  
Token: <ID A>  
Token: <KEYWORD if>  
Token: <ID a>  
Token: <REALOP >>  
Token: <ID b_var>  
Token: <ID printf>  
Token: <STRINGLITERAL "Integer is larger\n">  
Token: <KEYWORD else>  
Token: <ID printf>  
Token: <STRINGLITERAL "Float is larger or equal\n">  
Token: <KEYWORD char>  
Token: <ID message>  
Token: <STRINGLITERAL "Hello, world!">
```

Figure 6: Screenshot 1 of Test Case 2 Output

```
Token: <STRINGLITERAL "Hello, world!">  
Token: <ID printf>  
Token: <STRINGLITERAL "%s\n">  
Token: <ID message>  
Token: <KEYWORD return>  
Token: <NUMBER 0>
```

Figure 7: Screenshot 2 of Test Case 2 Output

### 2.4.6.3 Test Case 3 Screenshots

A screenshot of a terminal window with a dark background and light gray text. It displays a sequence of 20 tokens from a lexical analysis process, each on a new line. The tokens are: <KEYWORD int>, <ID main>, <KEYWORD int>, <ID arr>, <NUMBER 5>, <NUMBER 12>, <NUMBER 3>, <NUMBER 21>, <NUMBER 8>, <NUMBER 10>, <KEYWORD int>, <ID n>, <KEYWORD sizeof>, <ID arr>, <BINARYOP />, <KEYWORD sizeof>, <ID arr>, <NUMBER 0>, <KEYWORD for>, <KEYWORD int>, <ID i>, <NUMBER 0>, and <ID i>.

```
Token: <KEYWORD int>
Token: <ID main>
Token: <KEYWORD int>
Token: <ID arr>
Token: <NUMBER 5>
Token: <NUMBER 12>
Token: <NUMBER 3>
Token: <NUMBER 21>
Token: <NUMBER 8>
Token: <NUMBER 10>
Token: <KEYWORD int>
Token: <ID n>
Token: <KEYWORD sizeof>
Token: <ID arr>
Token: <BINARYOP />
Token: <KEYWORD sizeof>
Token: <ID arr>
Token: <NUMBER 0>
Token: <KEYWORD for>
Token: <KEYWORD int>
Token: <ID i>
Token: <NUMBER 0>
Token: <ID i>
```

Figure 8: Screenshot 1 of Test Case 3 Output



```
Token: <REALOP <>
Token: <ID n>
Token: <BINARYOP ->
Token: <NUMBER 1>
Token: <ID i>
Token: <BINARYOP +>
Token: <BINARYOP +>
Token: <KEYWORD for>
Token: <KEYWORD int>
Token: <ID j>
Token: <NUMBER 0>
Token: <ID j>
Token: <REALOP <>
Token: <ID n>
Token: <BINARYOP ->
Token: <ID i>
Token: <BINARYOP ->
Token: <NUMBER 1>
Token: <ID j>
Token: <BINARYOP +>
Token: <BINARYOP +>
Token: <KEYWORD if>
Token: <ID arr>
Token: <ID j>
```

Figure 9: Screenshot 2 of Test Case 3 Output

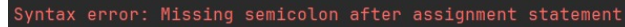
```
Token: <REALOP <>
Token: <ID arr>
Token: <ID j>
Token: <BINARYOP +>
Token: <NUMBER 1>
Token: <KEYWORD int>
Token: <ID temp>
Token: <ID arr>
Token: <ID j>
Token: <ID arr>
Token: <ID j>
Token: <ID arr>
Token: <ID j>
Token: <BINARYOP +>
Token: <NUMBER 1>
Token: <ID arr>
Token: <ID j>
Token: <BINARYOP +>
Token: <NUMBER 1>
Token: <ID temp>
Token: <ID printf>
Token: <STRINGLITERAL "Array in descending order: ">
Token: <KEYWORD for>
Token: <KEYWORD int>
```

Figure 10: Screenshot 3 of Test Case 3 Output

```
Token: <KEYWORD int>
Token: <ID i>
Token: <NUMBER 0>
Token: <ID i>
Token: <REALOP <>
Token: <ID n>
Token: <ID i>
Token: <BINARYOP +>
Token: <BINARYOP +>
Token: <ID printf>
Token: <STRINGLITERAL "%d ">
Token: <ID arr>
Token: <ID i>
Token: <ID printf>
Token: <STRINGLITERAL "\n">
Token: <KEYWORD return>
Token: <NUMBER 0>
```

Figure 11: Screenshot 4 of Test Case 3 Output

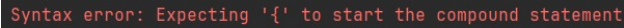
#### 2.4.6.4 Test Case 4 Screenshots



Syntax error: Missing semicolon after assignment statement

Figure 12: Screenshot of Test Case 4 Output

#### 2.4.6.5 Test Case 5 Screenshots



Syntax error: Expecting '{' to start the compound statement

Figure 13: Screenshot of Test Case 5 Output

## 3 Parser

### 3.1 Grammar Rules

Grammar Rules
<pre> declaration_list -&gt; (declaration declaration_list)   ε </pre>
<pre> declaration -&gt; (optional_keywords enum_declaration)                 (optional_keywords struct_declaration)                 var_declaration                 fun_declaration                 (type_specifier (pointer)* identifier (array_declaration                 ↪ fun_declaration   var_declaration)) </pre>
<pre> struct_declaration -&gt; STRUCT identifier LEFT_BRACE struct_member RIGHT_BRACE SEMICOLON </pre>
<pre> struct_member -&gt; type_specifier (pointer)* identifier (array_declaration)? (COMMA ↪ type_specifier (pointer)* identifier (array_declaration)?) * SEMICOLON </pre>
<pre> array_declaration -&gt; (optional_keywords identifier parse_array_dimensions ↪ (parse_array_initialization)?)   (parse_array_dimensions ↪ (parse_array_initialization)?) </pre>
<pre> parse_array_dimensions -&gt; LEFT_BRACKET (NUMBER)? RIGHT_BRACKET parse_array_dimensions? </pre>
<pre> parse_array_initialization -&gt; ASSIGNMENT_OP (STRING_LITERAL   CHARACTER_LITERAL   ↪ LEFT_BRACE parse_array_initialization_values RIGHT_BRACE) </pre>

```
enum_declaration -> ENUM identifier LEFT_BRACE enum_item RIGHT_BRACE SEMICOLON

enum_item -> ID (ASSIGNMENT_OP (NUMBER | ID))? (COMMA enum_item)?

var_declaration -> (optional_keywords) (identifier ASSIGNMENT_OP expression)? SEMICOLON

fun_declaration -> type_specifier identifier LEFT_PAREN param_list RIGHT_PAREN
  ↳ (compound_stmt | SEMICOLON)

param_list -> param (COMMA param)* | ε

param -> type_specifier (pointer)* identifier (array_call)?

compound_stmt -> LEFT_BRACE statement_list RIGHT_BRACE

statement_list -> statement statement_list | ε

statement -> if_statement | while_statement | for_statement | return_statement |
  ↳ do_while_statement | switch_statement | enum_declaration | declaration |
  ↳ assignment_statement | break_statement | continue_statement | expression_statement

array_call -> ID parse_array_dimensions_in_call parse_assignment_or_expression SEMICOLON

parse_array_dimensions_in_call -> LEFT_BRACKET expression RIGHT_BRACKET
  ↳ parse_optional_array_dimensions_in_call

parse_optional_array_dimensions_in_call -> LEFT_BRACKET expression RIGHT_BRACKET
  ↳ parse_optional_array_dimensions_in_call | ε

parse_assignment_or_expression -> ASSIGNMENT_OP (CHARACTER_LITERAL | STRING_LITERAL |
  ↳ expression) SEMICOLON | ε

break_statement -> BREAK SEMICOLON

continue_statement -> CONTINUE SEMICOLON

switch_statement -> SWITCH LEFT_PAREN expression RIGHT_PAREN switch_body SEMICOLON

switch_body -> LEFT_BRACE switch_cases RIGHT_BRACE

switch_cases -> switch_case default_case | switch_case | default_case | ε

switch_case -> CASE expression COLON statement_list
```

```
default_case -> DEFAULT COLON statement_list

assignment_statement -> ID (ASSIGNMENT_OP | COMPOUND_OP) parse_assignment_or_expression

if_statement -> IF LEFT_PAREN expression RIGHT_PAREN if_body

if_body -> statement SEMICOLON | LEFT_BRACE statement_list RIGHT_BRACE | statement ELSE
    ↪ if_statement | ELSE statement

expression -> conditional_expression

conditional_expression -> logical_or_expression (TERNARY_OP expression COLON
    ↪ expression)?

logical_or_expression -> logical_and_expression (LOGICAL_OR logical_and_expression)*

logical_and_expression -> equality_expression (LOGICAL_AND equality_expression)*

equality_expression -> relational_expression ((EQUALITY_OP | INEQUALITY_OP)
    ↪ relational_expression)*

relational_expression -> additive_expression ((RELATIONAL_OP | RELATIONAL_OR_EQUAL_OP |
    ↪ RELATIONAL_AND_EQUAL_OP) additive_expression)*

additive_expression -> multiplicative_expression ((PLUS_OP | MINUS_OP)
    ↪ multiplicative_expression)*

multiplicative_expression -> unary_expression ((MULTIPLICATION_OP | DIVISION_OP |
    ↪ MODULUS_OP) unary_expression)*

unary_expression -> (PLUS_OP | MINUS_OP | UNARY_OP) unary_expression |
    ↪ primary_expression

primary_expression -> NUMBER
    | LEFT_PAREN expression RIGHT_PAREN
    | ID parse_primary_expression_suffix

parse_primary_expression_suffix -> LEFT_PAREN function_call_arguments RIGHT_PAREN
    | LEFT_BRACKET expression RIGHT_BRACKET

function_call_arguments -> (expression (COMMA expression)*)?

function_call -> ID LEFT_PAREN args_opt RIGHT_PAREN (DOT function_call)* SEMICOLON
```

```

args_opt -> args_list |  $\epsilon$ 

args_list -> expression (COMMA expression)*

while_statement -> WHILE LEFT_PAREN expression RIGHT_PAREN statement SEMICOLON

do_while_statement -> DO statement WHILE LEFT_PAREN expression RIGHT_PAREN SEMICOLON

for_statement -> FOR LEFT_PAREN for_init_declaration SEMICOLON expression SEMICOLON
    ↪ for_update_exp RIGHT_PAREN statement

for_init_declaration -> TYPE_SPECIFIER (TYPE_SPECIFIER)? ID (ASSIGNMENT_OP expression)?
    ↪ SEMICOLON

for_update_exp -> (ID (UNARY_OP)?)?

return_statement -> RETURN LEFT_PAREN expression? RIGHT_PAREN SEMICOLON

```

---

## 3.2 Implementation Details

### 3.2.1 Parsing Algorithm

---

#### Parsing Algorithm

---

Top-down Parsing, specifically Recursive Descent, is a parsing technique commonly used  
 ↪ in compiler construction to parse the syntax of a programming language.

In Recursive Descent parsing, the parsing process starts from the top-level function,  
 ↪ usually named ``parse()``. This function initiates the parsing of the entire program.

The ``parse()`` function typically calls other functions representing different syntactic  
 ↪ constructs of the language, such as ``program()``, ``statement_list()``, and  
 ↪ ``statement()``.

- The ``program()`` function is responsible for parsing the structure of the program,  
 ↪ which may consist of a sequence of statements.

- The ``statement_list()`` function parses a list of statements within the program. It  
 ↪ iterates through the input until it reaches the end, parsing each statement  
 ↪ individually.

- The ``statement()`` function is where individual statements are parsed based on their  
 ↪ type. It determines the type of statement (e.g., ``while``, ``for``, ``return``) and calls  
 ↪ corresponding parsing functions for each type.

Specific parsing functions are defined for different types of statements, such as

- ``while_statement()``, ``for_statement()``, and ``return_statement()``. These functions
- handle the details of parsing respective statement types and are called from within
- the ``statement()`` function.

Each parsing function is responsible for recognizing and processing the syntactic

- elements corresponding to its statement type. For example, ``while_statement()``
- parses while loop statements, ``for_statement()`` parses for loop statements, and so
- on.

By organizing parsing logic into separate functions, Recursive Descent parsing enables

- modular and readable code, making it easier to understand and maintain.
- Additionally, it allows for straightforward implementation of language-specific
- syntax rules and error handling mechanisms.

---

### 3.2.2 General Overview

---

#### General Overview

---

The provided code implements a parser for a programming language using the top-down

- parsing approach known as recursive descent. The parser is designed to analyze the
- syntax of the input code and build a corresponding syntax tree.

The main components of the parser include functions for parsing expressions, statements,

- and control flow constructs such as while loops, do-while loops, for loops, and
- return statements. Additionally, there are functions for handling function calls,
- variable declarations, unary expressions, and more.

The code follows a structured approach, with each function responsible for parsing a

- specific grammar rule or syntactic construct. For example, the "expression" function
- handles the parsing of general expressions, while the "statement" function parses
- various types of statements such as if statements, while loops, and return
- statements.

The parser is designed to handle different types of tokens, including identifiers,

- numbers, arithmetic operators, unary operators, parentheses, and semicolons. Error
- handling mechanisms are incorporated throughout the code to detect syntax errors and
- provide informative error messages.

Overall, the parser serves as a fundamental component of a compiler or interpreter for

- ↪ the target programming language, enabling the analysis and interpretation of source
- ↪ code written in that language.

### 3.3 Parse Tree Screenshots

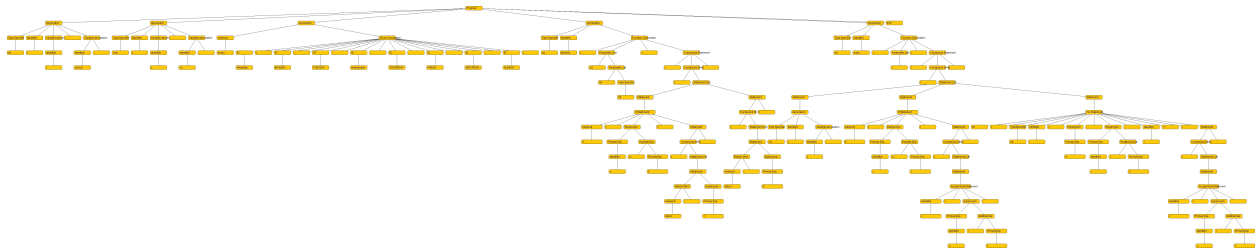


Figure 14: Parse Tree