MODULE-III

SYNTAX DIRECTED TRANSLATION AND TYPE CHECKING

3.1 SYNTAX DIRECTED DEFINITION

A **Syntax-Directed Definition (SDD)** is a formal framework used to define the semantics of programming languages in a way that is directly tied to the syntactic structure of the language. It associates each syntactic construct (i.e., production) of a context-free grammar with semantic rules, which provide meaning to the language constructs.

In essence, an SDD is a specification of the semantics of a language based on its syntax, where the meaning of a program is determined by the structure (syntax) of the program according to the grammar and the semantic rules attached to the grammar's production rules.

Key Components of Syntax-Directed Definitions

1. **Context-Free Grammar (CFG)**: The syntax of the language is specified using a context-free grammar. This defines the syntactic structure, such as how statements, expressions, and other constructs in the language can be formed.
2. **Attributes**: These are associated with the grammar's non-terminals and terminals. They are used to hold information about the language constructs. There are two types of attributes:
   - **Synthesized Attributes**: Information that is propagated **up** the parse tree, from leaves to root. These attributes are computed from the children nodes (e.g., the values of expressions or types of variables).
   - **Inherited Attributes**: Information that is propagated **down** the parse tree, from the root to the leaves. These are passed from parent to child (e.g., context-sensitive information like scopes or the types of variables).
3. **Semantic Rules**: These rules define how the attributes are computed. They specify how the values of attributes should be derived based on the syntactic structure of the language. The rules are usually attached to the productions in the grammar.
4. **Syntax Tree**: A parse tree or abstract syntax tree (AST) represents the syntactic structure of a program. The semantic rules of an SDD are used to annotate this tree with values that correspond to the meaning of the program.

Example of Syntax-Directed Definition

Let's define a simple language of arithmetic expressions consisting of integers, addition, and multiplication. The grammar and corresponding syntax-directed definitions would look as follows:

Grammar

E → E + T

E → T

T → T * F

T → F

F → id

F → num

Syntax-Directed Definition

We will associate attributes with the non-terminals and terminals:

- **E.val**: The value of an expression E.
- **T.val**: The value of a term T.
- **F.val**: The value of a factor F.

The semantic rules that define how to compute these attributes are as follows:

1. **E → E + T**
   - Semantic rule: E.val = E1.val + T.val (The value of E is the sum of the values of E and T.)
2. **E → T**
   - Semantic rule: E.val = T.val (The value of E is just the value of T.)
3. **T → T * F**
   - Semantic rule: T.val = T1.val * F.val (The value of T is the product of the values of T and F.)
4. **T → F**
   - Semantic rule: T.val = F.val (The value of T is just the value of F.)
5. **F → id**
   - Semantic rule: F.val = get_id_value(id) (The value of F is the value of the identifier id.)
6. **F → num**
   - Semantic rule: F.val = num (The value of F is the numerical value of the terminal num.)

How It Works

Consider the expression: id + num * id

Step 1: Parsing and Semantic Rule Application

- First, the grammar is used to parse the input into a parse tree (or abstract syntax tree). For this expression, the parse tree looks like:

```
      E
     / \
    E  +  T
   /    /  \
  T    T  *  F
 /    /  \
F    F    id
/     |
```

id   num

Step 2: Attribute Propagation

- The leaf nodes (id and num) are assigned values. Let's assume:
  - id = 5
  - num = 3
- Now, starting from the leaf nodes and applying the semantic rules:

1. For F → id, the semantic rule is F.val = get_id_value(id). So, for the first F (which is id), we get F.val = 5.
2. For F → num, the semantic rule is F.val = num. So, for the second F (which is num), we get F.val = 3.
3. Moving up to T → T * F, the semantic rule is T.val = T1.val * F.val. Using the computed values for T and F, we get T.val = 3 * 5 = 15.
4. For E → E + T, the semantic rule is E.val = E1.val + T.val. Using the computed values for E and T, we get E.val = 5 + 15 = 20.

Final Result

The final value of the entire expression id + num * id is 20, which is the result of applying the syntax-directed definitions to the parse tree.

Types of Syntax-Directed Definitions

1. **SDDs for Attribute Grammars**:
   - Syntax-directed definitions are a formalization of **attribute grammars**, which are grammars augmented with attributes that represent meaning.
   - SDDs are closely related to **attribute grammars**, where attributes are associated with symbols in the grammar, and semantic rules specify how to compute those attributes.
2. **SDDs for Compiler Phases**:
   - SDDs are commonly used in various phases of a **compiler**. For example:
     - **Lexical analysis**: Tokens are assigned meaning through attributes.
     - **Syntax analysis**: The syntax tree is constructed while associating semantic information.
     - **Semantic analysis**: Checking types, scopes, etc., by propagating attributes through the parse tree.
     - **Code generation**: Mapping the syntax tree to machine instructions using semantic rules.

Advantages of Syntax-Directed Definitions

1. **Formal Semantics**: SDDs provide a formal way of defining the semantics of a programming language, which makes it easier to understand and implement language features.
2. **Clear Structure**: By directly associating semantic rules with the grammar’s productions, SDDs offer a clear structure for managing both syntax and semantics.

3. **Modularity**: SDDs make it easier to add or modify semantics without altering the overall syntax of the language.
4. **Tool Support**: Many **compiler construction tools** like **Yacc**, **ANTLR**, or **Bison** support generating parsers that can handle syntax-directed definitions, making it easier to automate the generation of parsers and semantic analyzers.

3.2 S-ATTRIBUTED AND L-ATTRIBUTEDDEFINITIONS

S-attributed definitions are a subclass of syntax-directed definitions in which **all attributes are synthesized** (i.e., computed from the attributes of the children nodes in the parse tree).

Key Characteristics of S-attributed Definitions:

1. **Only Synthesized Attributes**: In an S-attributed definition, every attribute is **synthesized**. This means that the attribute of a non-terminal or terminal is calculated based on the attributes of its children or terminals.
2. **Attribute Flow**: Since attributes are synthesized, they are propagated **upwards** in the parse tree, starting from the leaves and going towards the root.
3. **Evaluation Order**: Since synthesized attributes are calculated based on the children, it is easy to establish a **bottom-up** evaluation order. The parser can process children first and then propagate the results upwards.
4. **Efficiency**: S-attributed grammars are computationally efficient since all attributes are computed in a straightforward, bottom-up manner.

Example of S-attributed Definition

Consider a simple arithmetic expression grammar:

E → E + T

E → T

T → T * F

T → F

F → id

F → num

Let's define the **synthesized** attribute val for each non-terminal. The value of an expression or term is computed by the following semantic rules:

- E→E+T:E.val=E1.val+T.val
  (The value of E is the sum of the values of E and T.)
- E→T:E.val=T.val
  (The value of E is the value of T.)
- T→T*F:T.val=T1.val*F.val
  (The value of T is the product of the values of T and F.)
- T→F:T.val=F.val
  (The value of T is the value of F.)
- F→id:F.val=get_id_value(id)
  (The value of F is the value of the identifier id.)

- F→num:F.val=num
  (The value of F is the numerical value of num.)

All attributes (val) are synthesized in this grammar, and the computation of val propagates upwards in the parse tree, making it an **S-attributed definition**.

L-attributed Definitions

L-attributed definitions are another subclass of syntax-directed definitions, but they impose a restriction on the order in which attributes can be evaluated. Specifically, in an L-attributed definition:

1. **Both Synthesized and Inherited Attributes**: In an L-attributed grammar, both **synthesized** and **inherited** attributes can be used.
   - **Synthesized attributes** are propagated upwards in the parse tree, similar to S-attributed definitions.
   - **Inherited attributes** are passed **downwards** from the parent node to the child nodes (or passed across siblings in the parse tree).
2. **Evaluation Order**: L-attributed definitions impose a restriction on the **order of evaluation**. The inherited attributes of a node can depend on both the attributes of its parent node and its siblings. However, the inherited attributes of a node must only depend on **the already-evaluated attributes of its parent or left siblings** (not right siblings), which ensures that an efficient **left-to-right** evaluation order is possible.
3. **Efficiency**: L-attributed grammars are still efficient in terms of computation because the inherited attributes can be passed down in a left-to-right manner, and both synthesized and inherited attributes can be evaluated in a single left-to-right pass over the input.

Example of L-attributed Definition

Consider a simple example involving variable declarations and expressions:

S → var id : T

T → int

T → bool

E → E1 + E2

E → id

Here, we define the following attributes:

- **T.type**: Type of the variable or expression.
- **E.val**: Value of the expression (this is synthesized).
- **id.type**: The type of the identifier (inherited from the declaration).

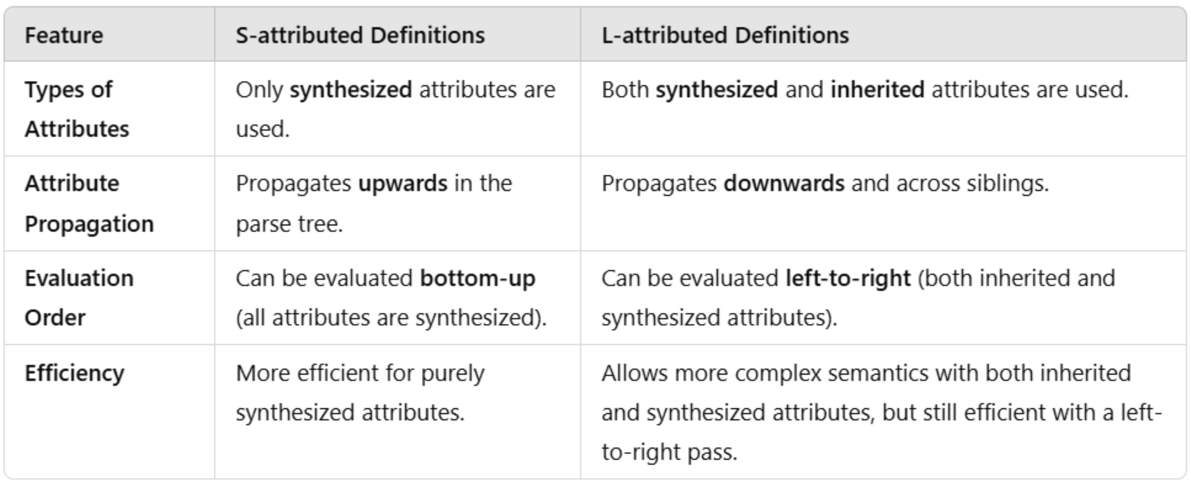
Now, let's define the semantic rules:

- S → var id : T:
  - id.type = T.type (Inherited attribute: id gets the type of T from the parent.)
- T → int:
  - T.type = "int" (Synthesized attribute: The type of T is int.)
- T → bool:

    - T.type = "bool" (Synthesized attribute: The type of T is bool.)
- E → E1 + E2:
    - E.val = E1.val + E2.val (Synthesized attribute: The value of E is the sum of the values of E1 and E2.)
- E → id:
    - E.val = get_id_value(id) (Synthesized attribute: The value of E is the value of the identifier id.)

In this case, the type of an identifier is passed **downwards** from the declaration (id.type = T.type), which makes this definition **L-attributed**.

Key Differences Between S-attributed and L-attributed Definitions



Feature	S-attributed Definitions	L-attributed Definitions
Types of Attributes	Only **synthesized** attributes are used.	Both **synthesized** and **inherited** attributes are used.
Attribute Propagation	Propagates **upwards** in the parse tree.	Propagates **downwards** and across siblings.
Evaluation Order	Can be evaluated **bottom-up** (all attributes are synthesized).	Can be evaluated **left-to-right** (both inherited and synthesized attributes).
Efficiency	More efficient for purely synthesized attributes.	Allows more complex semantics with both inherited and synthesized attributes, but still efficient with a left-to-right pass.

3.3 CONSTRUCTION OF SYNTAX TREES

A **syntax tree**, also called a **parse tree**, is a tree representation of the syntactic structure of a source program according to a formal grammar. Each internal node of the tree represents a non-terminal symbol in the grammar, while the leaf nodes represent the terminal symbols (tokens). Constructing a syntax tree is a key part of the **syntax analysis** phase of a compiler or interpreter.

Purpose of Syntax Trees

- **Representation of the Syntax**: The syntax tree reflects the structure of the source program as defined by the grammar of the language.
- **Semantic Analysis**: The tree is used in later stages (e.g., semantic analysis, code generation) to derive meaning from the syntactic structure.
- **Facilitating Optimization**: In compilers, the syntax tree is often transformed into an abstract syntax tree (AST) for further processing or optimization.

Types of Syntax Trees

1. **Parse Tree (Full Syntax Tree)**: A parse tree is a full representation of a grammar, showing every rule applied during the derivation of the input string.
2. **Abstract Syntax Tree (AST)**: An abstract syntax tree simplifies the parse tree by removing non-essential nodes (like intermediate grammatical constructs) and focusing on the actual operations and data structures of the program.

Steps for Constructing a Syntax Tree

1. **Input Token Stream**: The first step in constructing a syntax tree is having an input token stream. This stream is typically generated by the **lexical analyzer** (lexer) and consists of tokens like keywords, identifiers, operators, literals, etc.
2. **Parse Using Grammar**: To construct the syntax tree, you need to parse the input according to a **context-free grammar (CFG)**. Parsing involves applying the production rules of the grammar to match the tokens in the input.
3. **Recursive Descent Parsing**: This is a top-down parsing method where you start from the start symbol of the grammar and recursively apply production rules to match parts of the input. As you apply each production, you create nodes in the syntax tree corresponding to the non-terminal and terminal symbols.
4. **Bottom-Up Parsing**: In bottom-up parsing, you begin with the input tokens (the leaves of the tree) and try to reduce them to the start symbol by applying the production rules in reverse. This approach is commonly used in **shift-reduce parsers**.
5. **Handling Ambiguities**: Sometimes, the grammar may be ambiguous, meaning there are multiple ways to parse the same input. In such cases, the parser may need to use additional strategies to select the correct parse tree (such as prioritizing certain rules, or using a **grammar disambiguation** technique).

Example 1: Constructing a Parse Tree

Consider the following grammar for arithmetic expressions:

E → E + T | T

T → T * F | F

F → (E) | id

Let's construct the parse tree for the expression: id + id * id.

Step-by-Step Parsing

Start with the start symbol (E).

Apply the production E → E + T:

```
     E
    / \
   E  +  T
```

Now, parse the left E. Apply E → T:

```
     E
    / \
   E  +  T
  /
 T
```

Now, parse T. Apply T → F:

```
     E
    / \
      E   +   T
     /     \
    T       F
   /
  F
```

Now, parse F. Apply F → id:

```
       E
      / \
      E   +   T
     /     \
    T       F
   /         \
  F           id
 /
Id
```

Now, parse the right side of the expression (T → T * F). Apply the production T → T * F:

```
       E
      / \
      E   +   T
     /     \
    T       T
   /         \
  F           F
 /             \
id              id


/ id
```

7. Continue parsing `F → id` for both `T` and `F`.

8. Finally, the full parse tree for `id + id * id` is:

```
      E
     / \
     E   +   T
```

```
    /    \
   T      T
  / \    / \
 F   *   F   F
/    /  /  \

id id id id
```

Example 2: Constructing an Abstract Syntax Tree (AST)

An **abstract syntax tree (AST)** simplifies the parse tree by omitting intermediate grammatical rules that do not contribute to the actual computation or interpretation of the program. In the case of the expression `id + id * id`, we can omit the multiplication and addition operators as separate nodes and instead focus on the operands and their relationships.

For `id + id * id`, the AST would look like this:

```
+
/ \

id                                    *                                    /
id id
```

Here:

- The AST focuses on the operands and operators that directly contribute to the evaluation of the expression.

- We don't need to explicitly represent the `T → T * F` production because the multiplication operation is naturally represented in the AST.

Techniques for Constructing Syntax Trees

1. **Recursive Descent Parsing**:

- This is a top-down parsing technique where each non-terminal has a corresponding procedure or function that recursively processes the input according to the grammar's rules.

- The parse tree is constructed during the recursive calls by creating nodes for non-terminals and terminals as the parser processes the input.

2. **LL(1) Parsing**:

- An **LL(1) parser** is a type of **top-down** parser that constructs a syntax tree using a **predictive parsing** approach, where the choice of production rule to apply is determined by looking at the next input token.

- The parser uses a stack to represent the parse tree, pushing non-terminals and popping them as it applies rules to match the input tokens.

3. **LR Parsing**:

- An **LR parser** is a **bottom-up** parser that constructs the syntax tree by starting from the input tokens and reducing them into higher-level constructs. In the case of **shift-reduce** parsing, the parser "shifts" tokens onto a stack and "reduces" them by applying production rules when possible.

- This type of parsing is suitable for constructing syntax trees for languages with complex and more powerful grammars.

4. **Shift-Reduce Parsing**:

- **Shift-reduce parsing** is a bottom-up parsing method commonly used in **LR parsers**. The parser shifts tokens onto a stack and, when it matches a right-hand side of a production rule, it reduces the stack by replacing the matched sequence with the non-terminal on the left-hand side of the rule.

- The syntax tree is built incrementally as reductions occur.

Challenges in Syntax Tree Construction

- **Ambiguity**: If the grammar is ambiguous, the same input can have multiple valid parse trees. In this case, the parser must either choose one or employ strategies for resolving ambiguity (e.g., using precedence rules or rewriting the grammar).

- **Left Recursion**: Some grammars may be left-recursive, meaning they can lead to infinite recursion in top-down parsers. To avoid this, left recursion must be eliminated, or an appropriate parsing method (like **LR parsing**) must be used.

- **Efficiency**: Parsing can be computationally expensive, especially for large programs. Optimizations like **memoization** or using **table-driven parsers** (e.g., LL or LR) can improve performance.

Summary

Constructing a **syntax tree** is essential for representing the syntactic structure of a program. It is created through the process of **parsing** using a context-free grammar. The process involves either top-down (e.g., recursive descent, LL parsing) or bottom-up (e.g., shift-reduce, LR parsing) parsing techniques. The **parse tree** represents the complete syntax, while the **abstract syntax tree (AST)** simplifies this representation by removing unnecessary details, focusing only on the meaningful components of the program. The construction of syntax trees is foundational in compiler design, semantic analysis, and code generation.

3.4 TYPE CHECKING

Type checking is a crucial phase in the compilation process, where the types of variables, expressions, and function return values are verified to ensure that they are used consistently according to the rules of the programming language. It helps detect errors like trying to perform arithmetic on non-numeric types or calling a function with the wrong number of arguments.

In this context, **Type Expressions** play an essential role in specifying and understanding the types of various constructs in a program, such as variables, expressions, and function signatures.

3.4.1 TYPE EXPRESSIONS

1. What Are Type Expressions?

A **type expression** is a formal way of describing the type of an entity (like a variable, function, or expression) in a programming language. It is essentially a formula or notation that defines the type of an object or a value in terms of simpler types.

In the context of **type checking**, type expressions are used to describe the type of variables, parameters, and the result of operations. These expressions are evaluated to determine if operations on variables are type-safe (i.e., if they can be correctly performed according to the type rules of the language).

For example:

- An **integer** type might be represented as Int.
- A **boolean** type might be represented as Bool.
- A **function** type that takes an integer and returns a boolean might be written as Int -> Bool.

Type expressions can be quite complex in languages with advanced features, such as polymorphism, higher-order functions, or type constructors.

2. Types of Type Expressions

Type expressions can be broadly classified into the following categories:

a. Basic Types

Basic or primitive types represent the simplest data types, such as:

- **Integer (Int)**: A type for whole numbers.
- **Boolean (Bool)**: A type for logical values, i.e., true or false.

- **Character (Char)**: A type for individual characters.

These types can be represented directly in type expressions as:

- Int
- Bool
- Char

b. Function Types

Function types represent functions that take arguments of certain types and return values of a specific type. A function type is expressed as a **function arrow** (->), which denotes that the function takes one type as input and produces another type as output.

Example:

- A function that takes an integer and returns a boolean is written as Int -> Bool.
- A function that takes two integers and returns an integer is written as Int -> Int -> Int (or equivalently, Int -> (Int -> Int)).

In some languages, **higher-order functions** (functions that take other functions as arguments or return them as results) are supported, and function types can become more complex.

c. Product Types (Tuples)

A product type (also known as a **tuple** or **record**) combines multiple types into one composite type. A tuple represents a fixed-size collection of heterogeneous types.

Example:

- A tuple containing an integer and a boolean is written as (Int, Bool).
- A record type in some languages might be represented as {x: Int, y: Bool}, where each field is typed.

d. Sum Types (Variants or Discriminated Unions)

A sum type represents a value that can be one of several different types, commonly used for **tagged unions** or **enums**. This allows values to belong to one of a finite set of types, with a tag that distinguishes them.

Example:

- A sum type that can either be an integer or a boolean might be represented as Int | Bool (where | means "or").
- A discriminated union might represent a Shape that can be either a Circle or a Rectangle, where Shape = Circle | Rectangle.

e. Recursive Types

Recursive types are types that refer to themselves. They are useful in representing structures that are naturally recursive, such as linked lists or trees.

Example:

- A **linked list** in a functional programming language might have a type like List = Empty | Cons of Int * List, where Cons is a constructor that holds an integer and another list, and Empty represents an empty list.

3. Type Checking Using Type Expressions

Type checking using type expressions involves ensuring that the types of expressions are consistent with the rules of the language. During this process, the **type system** will validate whether a given expression adheres to the expected type and whether operations on values of different types are permissible.

Example 1: Type Checking a Simple Expression

Consider the expression:

x + y

Where x and y are variables. To check if this expression is valid, we must check the types of x and y. For instance:

- If x and y are both of type Int, then the expression is valid (since the + operator is typically defined for integers).
- If x is of type Int and y is of type Bool, then the expression is **type incorrect** because + is not defined for integers and booleans.

Example 2: Type Checking a Function

Consider the following function definition in a language like Haskell or ML:

```
add :: Int -> Int -> Int

add x y = x + y
```

This type annotation states that add is a function that takes two arguments of type Int and returns a result of type Int.

- The type expression Int -> Int -> Int tells us that the function is curried (i.e., it takes one argument and returns a function that takes another argument).
- The function body x + y ensures that the arguments x and y are both of type Int because addition (+) is defined for Int types.

4. Advanced Type Expressions in More Complex Languages

In more advanced programming languages, type expressions may include concepts such as:

a. Polymorphism

Polymorphic types allow functions to operate on values of different types. These types are often expressed using **type variables** that act as placeholders for any type.

Example:

- In a language with parametric polymorphism, a generic function that takes a list of any type and returns the length of the list might be written as:

```
length :: [a] -> Int
```

- Here, a is a **type variable**, and the function works for any type a.

b. Dependent Types

Dependent types are types that depend on values. This means that the type of an expression can depend on the value of another expression, leading to more expressive type systems.

Example:

- A function that takes a list and returns the number of elements in the list, where the type of the list depends on its length, can be represented as:

```
length :: List n -> Int
```

- Here, n is not just a type variable but a value that represents the length of the list.

5. Type Inference

Many modern programming languages allow for **type inference**, where the compiler can deduce the types of expressions without explicit type annotations. Type inference uses the type rules of the language and the structure of the code to automatically assign types to variables and expressions.

Example:

- In Haskell, the expression:

```
add x y = x + y
```

The compiler infers the type of add to be Int -> Int -> Int, based on the + operator.

Challenges in Type Checking

- **Type Coercion**: Some languages allow **implicit type conversion** or coercion (e.g., converting int to float automatically). Type checking needs to determine whether such coercions are allowed and safe.
- **Polymorphic and Higher-Order Types**: Handling **polymorphic** types (like generics) and **higher-order types** (functions that take other functions as arguments or return them) can be complex and requires sophisticated type inference and checking techniques.
- **Dependent Types**: Checking dependent types requires reasoning about both the structure and the values in the program, which can make type checking computationally expensive and complex.

3.4.2 TYPE EQUIVALENCE

Type equivalence refers to the concept of comparing types to determine whether they are the same or compatible within a given programming language. It plays an essential role in **type checking**, **type inference**, and **type assignment** in compilers and interpreters. The goal is to ensure that operations on variables or expressions are type-safe and that the types involved are compatible according to the rules of the language.

There are different approaches to defining **type equivalence**, and understanding how two types are compared is crucial for a programming language's type system. In general, type equivalence is used in operations such as function argument checking, type assignment, and assignment statements in programs.

Types of Type Equivalence

The key forms of **type equivalence** used in programming languages are as follows:

1. **Structural Equivalence** (also known as **Content Equivalence**):
   - Two types are considered **structurally equivalent** if they have the same internal structure or composition, regardless of their names or where they are defined.
   - This approach compares types based on their **composition** rather than their **name**.
   - For example, two different type declarations, such as struct Point { int x; int y; } and typedef struct { int x; int y; } Point;, would be considered **structurally equivalent** because they represent the same structure (two integers for x and y).

Example:

```
typedef struct { int x; int y; } Point1;
```

typedef struct { int x; int y; } Point2;

// Point1 and Point2 are structurally equivalent,

// though they have different names.

2. **Name Equivalence** (also known as **Nominal Equivalence**):

- In **name equivalence**, two types are considered equivalent if they have the same name, regardless of whether their structures are identical.
- This means that two types that are defined with different names but are structurally the same are considered **not equivalent** in languages that use name equivalence.
- **Name equivalence** is used in languages that treat types based on their declared name rather than their structure.

Example:

typedef struct { int x; int y; } Point1;

typedef struct { int x; int y; } Point2;

// Point1 and Point2 are **not** equivalent under name equivalence,

// even though their structure is the same.

3. **Weak Equivalence**:

- **Weak equivalence** allows for some flexibility, meaning that two types are considered equivalent if their structure is sufficiently similar, even if they have different names.
- This is a middle ground between **name equivalence** and **structural equivalence**, where certain differences (like typedefs) are ignored.

Example:

typedef struct { int x; int y; } Point1;

typedef struct { int x; int y; } Point2;

// Point1 and Point2 are equivalent under weak equivalence,

// as their structure is the same, regardless of name.

4. **Intersection Equivalence**:

- **Intersection equivalence** involves comparing types based on the intersection of their structures. This approach is often used in type systems that support **subtyping** or **union types**.
- Two types are considered equivalent if they share the same common properties (i.e., their intersection is non-empty).

Examples of Type Equivalence

1. C/C++ Style (Name Equivalence)

In C and C++, type equivalence is typically **name-based**. For example:

```
typedef struct { int x; int y; } Point1;

typedef struct { int x; int y; } Point2;


// Point1 and Point2 are not considered equivalent, because they have different names.
```

Here, Point1 and Point2 are not equivalent because C uses **name equivalence**: the types must have the same name to be considered equal, even if their structures are identical.

2. Type Equivalence in Functional Languages (Structural Equivalence)

In languages like **Haskell** or **ML**, type equivalence is often **structural**. For example:

```
type Point1 = (Int, Int)

type Point2 = (Int, Int)


-- Point1 and Point2 are structurally equivalent,

-- because they represent the same tuple structure (Int, Int).
```

In this case, Point1 and Point2 are **structurally equivalent** because both represent a tuple containing two integers, even though they are defined with different names.

3. Type Equivalence with Records and Tuples

In **record types** or **tuple types**, equivalence can depend on how the types are constructed:

```
typedef struct { int x; int y; } Point1;

typedef struct { int a; int b; } Point2;


// Point1 and Point2 are considered different types under name equivalence,

// but may be equivalent under structural equivalence, since their structures are identical.
```

When is Type Equivalence Used?

- **Function Arguments**: When checking if function arguments match the expected types, equivalence rules determine whether the argument and parameter types are compatible.
- **Assignment Statements**: Type equivalence is checked to ensure that the types of the left-hand side and the right-hand side of an assignment are compatible.
- **Type Inference**: During type inference, a system needs to ensure that different occurrences of the same variable or expression are compatible by checking type equivalence.
- **Subtyping**: In languages that support subtyping (like Java), checking for subtype relationships often involves comparing the types of two entities and seeing if one is a subtype of the other.
- **Generics/Parametric Types**: In languages with **generic types** (e.g., Java, C#), type equivalence is used to check whether types in generic functions or classes are compatible.

Subtype and Supertype Equivalence

In languages with **subtyping** (such as those that support object-oriented programming), type equivalence may extend to **subtype** and **supertype** relationships:

- A type T is considered a **supertype** of another type S if every instance of S can also be an instance of T.
- **Subtype equivalence** allows one type to be considered equivalent to another if it is a subtype, meaning it has at least the same behavior as the other type but potentially additional capabilities (fields or methods).

For example, in **object-oriented languages** (like Java), a class Dog might be considered a subtype of a class Animal if Dog inherits from Animal. Therefore, a variable of type Animal can hold an instance of type Dog.

Challenges in Type Equivalence

1. **Ambiguity in Mixed-Type Systems**: Some languages have complex type systems, with combinations of structural and name-based equivalence. Deciding which rules to apply in such cases can be challenging.
2. **Type Aliases and Typedefs**: Type aliases or typedef in languages like C/C++ can make it harder to check type equivalence since the same structure may be represented with different names, which may or may not be considered equivalent depending on the system's rules.
3. **Polymorphism and Generics**: In polymorphic languages (such as C++ or Java with templates), type equivalence becomes complex when generics or templates are involved. Types can be instantiated with different concrete types, making it necessary to check equivalence in a more dynamic way.
4. **Subtyping**: In systems with subtyping, defining when two types are equivalent is tricky because one type could be a subtype of the other, and type equivalence might not hold even if the types are compatible in many contexts.

3.4.3 RULES FOR TYPE CHECKING

Type checking is the process of verifying that the types of variables and expressions in a program are used correctly according to the rules of the language. The goal is to ensure that operations on variables are type-safe and that the program adheres to the language’s type system.

Here are the key **rules for type checking** that are commonly applied in most programming languages:

1. Type Compatibility Rules

- **Unary and Binary Operators**: Ensure that the operands of operators like +, -, *, /, ==, etc., have compatible types.
    - For example, + can be used on numeric types (Int, Float) but not on Bool or String.
    - Example: In an expression like x + y, both x and y must be of a type that supports addition (e.g., both Int or both Float).
- **Relational Operators**: Ensure that the operands of relational operators like ==, <, >, <=, >= are of types that can be compared.
    - Example: x < y requires that both x and y are of types that support comparison (e.g., Int, Float).
- **Boolean Operators**: Ensure that operands of boolean operators like &&, ||, !, etc., are of type Bool.

    - Example: x && y requires both x and y to be of type Bool.
- **Assignment Compatibility**: In an assignment x = y, ensure that the type of y matches the type of x or is compatible with it (considering type casting or coercion).
    - Example: x = 5 assigns an Int value 5 to x if x is of type Int.

2. Variable Declaration and Type Assignment Rules

- **Variable Declaration**: Each variable must be declared with a specific type, and any subsequent use of the variable must match its declared type.
    - Example: If int x; is declared, then x can only be used in contexts where an integer is expected.
- **Type Consistency in Expressions**: When variables are used in expressions, the expression must adhere to the expected type.
    - Example: In the expression x + y, if x is an Int, y must also be of type Int for the expression to be valid.
- **Constant and Literal Values**: Constant values such as 5, true, or "hello" have inherent types, and their types must be compatible with where they are used.
    - Example: "hello" + "world" is valid for string concatenation but invalid for integer addition.

3. Function Type Checking Rules

- **Function Parameters**: When a function is called, the arguments passed must match the types of the corresponding parameters. The type of each argument must be checked against the parameter type.
    - Example: For a function int add(int x, int y), calling add(2, "3") is an error because "3" is a string, not an integer.
- **Return Type**: The return type of a function must be checked for consistency with the expected return type. If the function is expected to return an integer, returning a string or a boolean is an error.
    - Example: For a function float divide(int x, int y), returning an Int instead of a Float would be an error.
- **Function Overloading**: In languages that support **function overloading** (like C++, Java), the function signatures must be checked to ensure the correct function is called based on the number and types of arguments.
    - Example: Overloading the function add(a, b) where one takes int and the other takes float requires that the correct overload is selected based on the argument types.

4. Type Checking for Complex Data Types

- **Arrays/Lists**: Ensure that the elements of an array or list are of the specified type. If an array is declared as int arr[10], then all elements assigned to it must be of type Int.
    - Example: arr[0] = 10; is valid if arr is an integer array, but arr[0] = "hello"; is invalid.
- **Records/Structs**: Each field in a record or struct must be assigned a value of the correct type, as specified in the type definition.
    - Example: For a struct Point { int x, y; }, assigning a string to x would be an error.

- **Union/Variant Types**: If a type can be one of several types (as in **sum types** or **discriminated unions**), type checking must ensure that the correct type is selected from the union and that operations are performed on that specific type.
    - Example: A variable of type Shape that could be either Circle or Rectangle must ensure that a Circle-specific operation (like calculating the area) isn't performed on a Rectangle.
- **Type Aliases**: In languages that support type aliases (e.g., C, C++), an alias is treated as equivalent to the original type. Type checking must account for this equivalence and allow the alias to be used in place of the original type.
    - Example: typedef int Integer; means that Integer is equivalent to Int, and the same type rules apply.

5. Polymorphism and Type Checking

- **Polymorphism**: In languages with **polymorphism** (like C++ or Java), type checking must account for **subtypes** (in object-oriented languages) or **generic types** (in functional programming languages).
    - Example: In Java, a method expecting a parameter of type Animal should accept an argument of type Dog if Dog is a subclass of Animal.
- **Generic Types**: For **generic functions** or **templates** (in languages like Java or C++), type checking ensures that the parameters passed match the types expected by the generic definition.
    - Example: List<Integer> is a list of integers, and only integer elements can be added to it.

6. Type Checking for Expressions

- **Unary Expressions**: In expressions involving unary operators (e.g., -, !), type checking ensures that the operand has a valid type for the operator.
    - Example: -x is valid only if x is a numeric type (like Int or Float).
    - Example: !x is valid only if x is a Bool.
- **Binary Expressions**: In expressions involving binary operators (e.g., +, -, *, /), both operands must have compatible types for the operation.
    - Example: x + y is valid only if x and y are both integers (Int) or both floating-point numbers (Float).
    - Example: x == y requires both x and y to be of a comparable type (Int, Float, String, etc.).
- **Type Promotions**: Some languages allow for **automatic type conversions** or **type promotions** (e.g., from Int to Float), and the type checking process must account for such conversions.
    - Example: int x = 5; float y = x; (implicitly promotes x from Int to Float).

7. Context-Sensitive Type Checking

- **Scope Checking**: Ensure that variables are declared before they are used in expressions. Variables must be checked to see if they exist in the current scope, and their types should be consistent with the declaration.

- Example: Using a variable before its declaration would result in a type error.

- **Type Compatibility Across Scopes**: Ensure that variables declared in different scopes (e.g., in different functions or blocks) have consistent types when they are referenced or passed as arguments.
  - Example: Passing a variable from one function to another should be valid if the types match.

8. Type System Enforcement Rules

- **Strong vs Weak Typing**: In **strongly-typed languages**, types are strictly enforced, and implicit conversions are disallowed unless explicitly defined. In **weakly-typed languages**, type conversions might occur automatically.
  - Example: In a **strongly typed** language like Haskell, trying to add a string and an integer will result in a type error. In **weakly-typed** languages like JavaScript, implicit type conversion might allow such an operation.
- **Type Inference**: In languages with **type inference** (like Haskell or Scala), the type system infers the types of variables and expressions based on their usage.
  - Example: If x = 10, the type checker will infer that x is of type Int.

3.4.4 TYPE CONVERSIONS

Type conversion (or **type casting**) refers to the process of converting a value from one type to another. This is an essential part of type systems in programming languages, as it allows operations to be performed on values of different types. Type conversions are typically categorized into two types: **implicit** and **explicit** conversions.

1. Implicit Type Conversion (Automatic Conversion)

Implicit type conversion is done automatically by the compiler or interpreter. It occurs when the compiler can safely convert one type to another without any loss of information, often when types are compatible or when a smaller type is promoted to a larger type.

- **Automatic type conversion** happens **without the need for explicit instructions** from the programmer.
- The compiler determines when it is safe to convert the types.
- This is often referred to as **type coercion** or **type promotion**.

Examples of Implicit Type Conversion

Promotion of Integer to Floating-Point: When performing an operation involving both integers and floating-point numbers, the integer is typically promoted to a floating-point number to ensure precision is maintained.

```
int x = 5;

float y = 2.5;

float result = x + y;  // x is implicitly converted to float

// result will be 7.5
```

Smaller Integer to Larger Integer (or Float): When a smaller type is assigned to a larger type (e.g., from int to long or float), the compiler performs the conversion automatically.

```
short s = 10;
```

```
int i = s;  // short is automatically converted to int
```

Char to Int or Int to Float: A char (character) can be implicitly converted to its integer value (ASCII value). Similarly, an int can be converted to float automatically.

```
char ch = 'A';  // 'A' has an ASCII value of 65

int num = ch;   // char is implicitly converted to int
```

Restrictions of Implicit Type Conversion

- Implicit conversions can lead to **loss of information** or **precision loss**. For example, when converting from float to int, the fractional part is discarded, which could lead to incorrect results.
- Implicit conversions only occur if there is no **risk of data loss** or **semantic errors** in the conversion.

2. Explicit Type Conversion (Type Casting)

Explicit type conversion, also known as **type casting**, requires the programmer to explicitly specify how one type should be converted into another. This is done using casting operators or functions, and is used when the compiler cannot automatically perform the conversion.

- This type of conversion allows the programmer to control the conversion process.
- It is used when converting between incompatible types, or when implicit conversion would result in **loss of data** or **precision**.

Examples of Explicit Type Conversion

C-style Cast (C/C++): In languages like **C** and **C++**, explicit conversion is done using the (type) syntax. The programmer specifies the target type inside parentheses.

```
float f = 3.14;

int i = (int) f;  // f is explicitly converted to an integer

// The result is i = 3, because the fractional part is discarded
```

Function-style Cast (C++/Java): Some languages like C++ and Java allow you to use a function-like syntax for casting.

```
double d = 3.14;

int x = int(d);  // Explicit conversion from double to int
```

Java-style Cast (Java): In **Java**, type casting can be done explicitly by using the cast operator (type).

```
double d = 3.75;

int i = (int) d;  // Explicit conversion from double to int

// i = 3, the fractional part is truncated
```

Using Conversion Functions: Some languages provide built-in functions for explicit type conversions. For instance, in Python, functions like int(), float(), and str() are used for type casting.

```
x = "10"

y = int(x)  # Converts string to integer
```

Explicit Conversion in Object-Oriented Languages

In **object-oriented** programming languages (e.g., **Java**, **C++**), explicit conversions are often used for **casting objects** to different classes in the inheritance hierarchy. This is especially relevant for **upcasting** and **downcasting**.

- **Upcasting**: Casting a subclass object to a superclass type.
- **Downcasting**: Casting a superclass object to a subclass type.

```
class Animal { }

class Dog extends Animal { }


Animal animal = new Dog();  // Upcasting (implicitly done)

Dog dog = (Dog) animal;  // Downcasting (explicitly done)
```

Risks of Explicit Type Conversion

- **Data Loss**: When converting between incompatible types (e.g., float to int), explicit conversion may lead to truncation or loss of information.
    - Example: Converting a large float value into an int could cause **overflow** or loss of precision.
- **ClassCastException (in Java)**: In Java, casting between incompatible types can lead to exceptions at runtime if the cast is invalid.

```
Object obj = new Integer(10);

String str = (String) obj;  // Throws ClassCastException
```

3. Types of Type Conversions

Type conversions can also be categorized based on the nature of conversion between different types:

1. Implicit Conversions (Type Promotion)

- **Numeric Promotion**: A smaller type is promoted to a larger type, typically from int to long, float, or double.
- **Boolean to Numeric**: In some languages, a boolean value true or false can be promoted to 1 or 0, respectively.

Example:

```
int x = 10;

float y = 5.5;

float result = x + y;  // x is promoted to float
```

2. Explicit Conversions

- **Narrowing Conversion**: Converting from a larger type to a smaller type. This is typically done explicitly, because it may result in data loss.
- **Type Casting**: Explicit conversion between unrelated or incompatible types, for example, from double to int.
- **Pointer Conversion**: In languages like **C/C++**, pointers to objects of one type can be converted to pointers to objects of another type, though this requires caution.

4. Special Considerations in Type Conversion

- **Floating Point Precision**: When converting between float and double, or between floating-point and integer types, the precision may be affected.

Example: Converting 3.14159 (double) to int would discard the fractional part.

- **String to Numeric Conversion**: Many languages allow strings to be converted to numeric types, but this can fail if the string doesn't represent a valid number.

Example (in Python)

```
s = "123"

num = int(s)  # Converts string to integer
```

- **Complex Data Types (e.g., Arrays, Objects)**: Type conversion between complex data structures like arrays or objects may require deep copying or constructing new instances, especially when converting between completely different data structures.

3.5 OVERLOADING OF FUNCTIONS

Function overloading is a feature in many programming languages (such as **C++, Java,** and **C#**) that allows multiple functions with the **same name** to coexist in the same scope, as long as they have different **parameters** (i.e., a different number or types of arguments). This enables a function to be called in different ways depending on the type or number of arguments passed.

Function overloading enhances the **flexibility** and **readability** of the code by allowing you to use the same function name for different operations that are conceptually similar but work with different input types or quantities.

1. How Function Overloading Works

Function overloading is based on the **function signature**, which consists of the function's **name** and its **parameter list**. The return type is **not** considered part of the signature, meaning that two overloaded functions cannot differ only by their return types.

For function overloading to be successful:

- **Parameters** (number, type, or order of parameters) must be different.
- **Return type** alone cannot be used to distinguish between overloaded functions.
- **Default parameters** can also affect overloading if they are used in one version of the function.

2. Examples of Function Overloading

1. Overloading Based on Number of Parameters

In this case, the same function name can be used, but the number of parameters differs.

```
#include <iostream>

using namespace std;


// Function to add two integers

int add(int x, int y) {

  return x + y;
}

// Overloaded function to add three integers
int add(int x, int y, int z) {
    return x + y + z;
}

int main() {
    cout << add(2, 3) << endl;      // Calls the first version (two parameters)
    cout << add(2, 3, 4) << endl;   // Calls the second version (three parameters)
    return 0;
}
```

Output:

```
5
9
```

2. Overloading Based on Different Parameter Types

Functions can also be overloaded based on the types of their parameters. The function name stays the same, but the types of arguments differ.

```
#include <iostream>
using namespace std;

// Function to add two integers
int add(int x, int y) {
    return x + y;
}

// Overloaded function to add two floating-point numbers
float add(float x, float y) {
    return x + y;
}

int main() {
    cout << add(2, 3) << endl;      // Calls the integer version
    cout << add(2.5f, 3.5f) << endl; // Calls the float version
    return 0;
}
```

Output:

```
5
6
```

3. Overloading Based on Parameter Order

Overloading can also be based on the order of parameters, where the function signatures are different in terms of the type order.

```
#include <iostream>
using namespace std;


// Function to multiply two numbers (int, float)
float multiply(int x, float y) {
    return x * y;
}


// Overloaded function to multiply two numbers (float, int)
float multiply(float x, int y) {
    return x * y;
}


int main() {
    cout << multiply(2, 3.5f) << endl;  // Calls the first version (int, float)
    cout << multiply(2.5f, 3) << endl;  // Calls the second version (float, int)
    return 0;
}
```

Output:

```
7
7.5
```

3. **Overloading in Object-Oriented Languages**

In **Object-Oriented Programming (OOP)** languages like **C++** or **Java**, function overloading is often used in methods to provide different behaviors for objects based on the number or type of arguments. This helps in creating a more **intuitive** and **clean** API for the class.

Example in C++:

```
#include <iostream>
```

```
using namespace std;

class Printer {
public:
    // Method to print a single integer
    void print(int x) {
        cout << "Printing integer: " << x << endl;
    }

    // Overloaded method to print a double
    void print(double x) {
        cout << "Printing double: " << x << endl;
    }

    // Overloaded method to print a string
    void print(string x) {
        cout << "Printing string: " << x << endl;
    }
};

int main() {
    Printer p;
    p.print(10);         // Calls print(int)
    p.print(3.14);       // Calls print(double)
    p.print("Hello!");   // Calls print(string)
    return 0;
}
```

Output:

```
Printing integer: 10
Printing double: 3.14
Printing string: Hello!
```

4. **Overloading in Java**

In Java, function overloading works similarly. Overloaded methods can be defined within the same class, but their parameter lists must differ.

Example in Java:

```
class Calculator {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));        // Calls add(int, int)
        System.out.println(calc.add(2, 3, 4));     // Calls add(int, int, int)
        System.out.println(calc.add(2.5, 3.5));    // Calls add(double, double)
    }
}
```

Output:

```
5
9
6.0
```

5. Key Points to Remember About Function Overloading

- **Overloading is based on the function signature** (name + parameters), but **not on return type**. Functions with the same name cannot be overloaded if the only difference between them is their return type.

- **Order of parameters** and **parameter types** can be used to distinguish overloaded functions.
- **Default parameters** can affect overloading in certain cases. If a function has default arguments, the compiler may have difficulty distinguishing which function to call if there are multiple candidates.

```
void print(int x = 10);  // Default argument
void print(int x = 20);  // This will cause ambiguity when calling print();
```

- Overloading can **increase readability** by providing multiple ways to perform the same operation with different input types, without needing to create distinct function names.

6. Advantages of Function Overloading

1. **Improved Readability**: By using the same function name for different operations that are conceptually similar, code becomes more readable and easier to maintain.
2. **Avoids Duplication**: Function overloading avoids the need to create multiple functions with different names for similar operations, reducing redundancy.
3. **Flexibility**: Provides flexibility to call the same function in different contexts, such as handling different data types or numbers of parameters.

7. Disadvantages of Function Overloading

1. **Ambiguity**: If not designed properly, overloaded functions might cause ambiguity, where the compiler is unable to decide which version of the function to call.
2. **Increased Complexity**: Overloading with many versions of the same function can make it harder for developers to understand the different variations of the function.
3. **Limits in Some Languages**: Not all languages support function overloading. Some languages, like **Python**, don't support it directly, relying on *dynamic typing* and *default arguments* to handle similar functionality.

3.6 OPERATORS

Operators are symbols or keywords that perform operations on variables and values. They are fundamental to programming, enabling the manipulation of data and the execution of logic within programs. Operators can be categorized based on their function and how they are used in expressions.

1. Types of Operators

Operators can be broadly classified into several types based on their functionality:

1.1 Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication, division, and modulus.

Addition (+): Adds two operands.

```
int a = 5, b = 3;
int sum = a + b;  // sum = 8
```

Subtraction (-): Subtracts the second operand from the first.

```
int diff = a - b;  // diff = 2
```

Multiplication (*): Multiplies two operands.

int prod = a * b; // prod = 15

Division (/): Divides the first operand by the second (integer division in many languages).

int div = a / b; // div = 1 (integer division)

Modulus (%): Returns the remainder of division of the first operand by the second.

int mod = a % b; // mod = 2

1.2 Relational (Comparison) Operators

Relational operators are used to compare two values or expressions and return a boolean value (true or false).

Equal to (==): Checks if two values are equal.

if (a == b) { /* true if a is equal to b */ }

Not equal to (!=): Checks if two values are not equal.

if (a != b) { /* true if a is not equal to b */ }

Greater than (>): Checks if the left operand is greater than the right operand.

if (a > b) { /* true if a is greater than b */ }

Less than (<): Checks if the left operand is less than the right operand.

if (a < b) { /* true if a is less than b */ }

Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.

if (a >= b) { /* true if a is greater than or equal to b */ }

Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

if (a <= b) { /* true if a is less than or equal to b */ }

1.3 Logical Operators

Logical operators are used to combine or invert boolean values or expressions.

AND (&&): Returns true if both operands are true.

if (a > 0 && b > 0) { /* true if both a and b are greater than 0 */ }

OR (||): Returns true if at least one operand is true.

if (a > 0 || b > 0) { /* true if either a or b is greater than 0 */ }

NOT (!): Inverts the boolean value of the operand.

if (!(a > 0)) { /* true if a is not greater than 0 */ }

1.4 Assignment Operators

Assignment operators are used to assign values to variables.

Simple Assignment (=): Assigns the value of the right operand to the left operand.

int a = 5; // a is assigned the value 5

Addition Assignment (+=): Adds the right operand to the left operand and assigns the result to the left operand.

a += 3; // equivalent to a = a + 3

Subtraction Assignment (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.

a -= 2; // equivalent to a = a – 2

Multiplication Assignment (*=): Multiplies the left operand by the right operand and assigns the result to the left operand.

a *= 2; // equivalent to a = a * 2

Division Assignment (/=): Divides the left operand by the right operand and assigns the result to the left operand.

a /= 2; // equivalent to a = a / 2

Modulus Assignment (%=): Takes the modulus of the left operand by the right operand and assigns the result to the left operand.

a %= 3; // equivalent to a = a % 3

1.5 Unary Operators

Unary operators operate on a single operand.

Increment (++): Increases the value of the operand by 1.

a++; // Post-increment: first uses the value of a, then increases a by 1

++a; // Pre-increment: first increases a by 1, then uses the value of a

Decrement (--): Decreases the value of the operand by 1.

a--; // Post-decrement: first uses the value of a, then decreases a by 1

--a; // Pre-decrement: first decreases a by 1, then uses the value of a

Negation (-): Negates the value of the operand.

int x = -a; // if a is 5, x becomes -5

Logical NOT (!): Negates the boolean value of the operand.

bool result = !(a > b); // if a > b is true, result becomes false

1.6 Bitwise Operators

Bitwise operators operate on the bit-level representation of integers.

AND (&): Performs bitwise AND on the bits of two operands.

int result = a & b; // bitwise AND

OR (|): Performs bitwise OR on the bits of two operands.

int result = a | b; // bitwise OR

XOR (^): Performs bitwise XOR on the bits of two operands.

int result = a ^ b; // bitwise XOR

NOT (~): Inverts the bits of the operand.

int result = ~a; // bitwise NOT

Left shift (<<): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.

```
int result = a << 2;  // shifts a left by 2 positions
```

Right shift (>>): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

```
int result = a >> 2;  // shifts a right by 2 positions
```

1.7 Conditional (Ternary) Operator

The conditional (ternary) operator is a shorthand for an if-else statement.

Ternary (?:): Evaluates a condition and returns one of two values depending on whether the condition is true or false.

```
int result = (a > b) ? a : b;  // returns a if a > b, otherwise returns b
```

1.8 Type-Casting Operators

Type casting allows you to explicitly convert between different data types.

C-style Cast ((type)): Converts one type to another.

```
float f = 5.7;
int i = (int) f;  // cast float to int (i will be 5)
```

Static Cast (static_cast<type>): In C++, this is a safer alternative to C-style casting.

```
double d = 5.6;
int i = static_cast<int>(d);  // safely cast double to int
```