Programming Language

Expressions and Assignment Statements

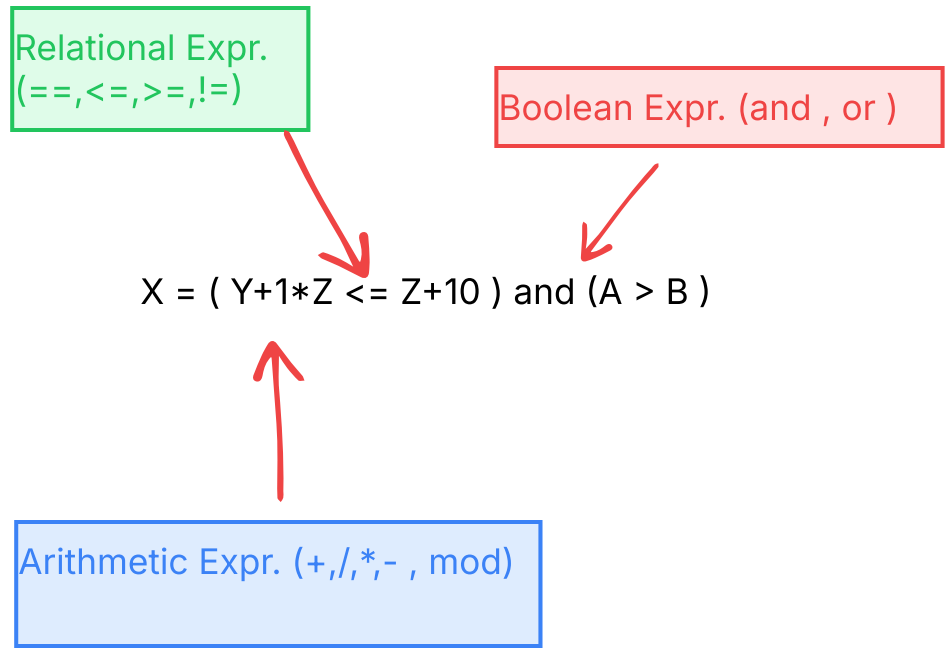
```cpp
#include <stdio.h>
//chapter 9
double addDouble(double a, double b){
    try{ // chapter 14 (try-catch)
        a - a/b;
    }catch(...){
        cout << "divide by zero";
    }
    return a | b;
}
int main(){
    // chapter 5,6
    int x = 5;
    double y;

    // chapter 7
    x = y * 6 + 1 * x;

    // chapter 8
    if(x<10){
        // chapter 9
        y = addDouble(2.5, 1.2);
    }
    // chapter 9
    return 0;
}
```

Contents

Type of Expression



Arithmetic expression

- An arithmetic expression is an expression that uses numbers and arithmetic operators and produces a numeric value.
  - Operator : +, / , *, mod

Relational expression

- A relational expression compares two values and produces a boolean result (true or false).
  - Operator : ==,!= , >, < ,

Boolean Expressions

- A boolean expression is an expression that evaluates to true or false, often using logical operators.
  - Operator : AND , OR , not

Expression Design Issues

Arithmetic:

1. Type System Design
   - Type conversion rules (coercion hierarchy)
   - Mixed-mode operation support
2. Operator Design
   - Precedence
   - Associativity
3. Evaluation Design
   - Order of operand evaluation
   - Side effects

Relational:

1. Type System Design
   - Mixed-mode operation support
   - Type conversion rules
   - Result type (boolean)
2. Operator Design
   - Precedence (relative to arithmetic/boolean)
   - Associativity (non-associative in most languages)
3. Evaluation Design
   - Order of operand evaluation
   - Side effects in operands

Boolean:

1. Type System Design
   - Type representation (dedicated boolean vs. numeric)
   - Truthiness rules (if applicable)
2. Operator Design
   - Precedence (AND, OR, NOT hierarchy)
   - Associativity
3. Evaluation Design
   - Short-circuit evaluation
   - Order of evaluation
   - Side effects

Arithemetic Expression Design Issues

1. Type System Design
   - Type conversion rules (coercion hierarchy)
   - Mixed-mode operation support
2. Operator Design
   - Precedence
   - Associativity
3. Evaluation Design
   - Order of operand evaluation
   - Side effects

Type conversion (1.1)

- A narrowing converion converts a value to a type that cannot store even approximations of all of the values of the original type. In Java, converting `double` to `float` is a narrowing conversion.
- A widening conversion converts a value to a type that can include at least approximations of all of the values of the origirnal type. In Java, converting `int` to `float` is a widening conversion.
- In general, **widening conversion is safer than narrowing**, but in some conversions **the precision may be lost**. For example, 32-bit integers allow at least 9 decimal digits of precision. But 32-bit floating-point values have only about seven decimal digits of precision (because of the space used for the exponent).
- The type of conversions can be either **explicit** or **implicit**.

Type Mixing (Mixed-Mode Expressions) (1.2)

1. Mixed-mode operation is an arithmetic expression where operands have **different numeric data types** (int, float) , causing the language to perform **implicit numeric type conversion** (also called **numeric promotion**) so the operation can be evaluated.

- Example:
  - `int` + `float` → **automatic coercion** to `float` which is an **implicit type conversion** that is initiated by the compiler or runtime system.
  - `int a = (int)5.0;` - this is an explicit type conversion (cast) requested by the programmer, not coercion.

2. Coercion (implicit type conversion) = **one way** to handle mixed-mode expressions by automatically converting types.
    - Coercion can **reduce program reliability** because errors may occur when the compiler **automatically converts between incompatible types without the programmer's explicit approval**.
    - Coercion can cause `overflow` or `underflow` when the converted value cannot fit in the target type
    - Other run-time errors from type operations include division by zero, which raises an exception.

```
x = 5 + 3.14  # This is a mixed-mode expression (int + float)
```

The language can handle this mixed-mode situation in different ways:

- With coercion: Automatically convert 5 to 5.0 and perform float addition
- Without coercion: Raise a type error and require explicit conversion

The computer cannot evaluate correctly and shows overflow instead. The actual result is -4294965296.

https://onlinegdb.com/09BHMaY25

```c
#include <stdio.h>
#include <limits.h>
int main() {
    int A = 1000; int B = INT_MIN;  // Most negative int value (-2147483648 on 32-bit)
    int C = 1000; int D = INT_MIN;  // Most negative int value
    printf("A = %d\n", A);
    printf("B = %d\n", B);
    printf("C = %d\n", C);
    printf("D = %d\n\n", D);
    // This will cause overflow
    int result = A + B + C + D;
    printf("A + B + C + D = %d\n", result);
    // Step by step to show where overflow occurs
    printf("Step by step:\n");
    int step1 = A + B;
    printf("A + B = %d\n", step1);
    int step2 = step1 + C;
    printf("(A + B) + C = %d\n", step2);
    int step3 = step2 + D;
    printf("((A + B) + C) + D = %d (OVERFLOW!)\n", step3);
```

List of languages that support mixed-mode

Language	Mixed-Mode Supported?	Example	Result
C	Yes	3 + 3.14	double
C++	Yes	3 + 3.14	double
Java	Yes	3 + 3.14	double
C#	Yes	3 + 3.14	double
Python	Yes	3 + 3.14	float
Go	No	3 + 3.14	Error
Swift	No	3 + 3.14	Error
Rust	No	3 + 3.14	Error

https://onlinegdb.com/ZBXHlEY3m

```
//let r = 3 + 3.14    // ERROR
let r = Double(3) + 3.14   // OK
print(r)
```

Arithemetic Expression Design Issues

1. Type System Design
   - Type conversion rules (coercion hierarchy)
   - Mixed-mode operation support
2. Operator Design
   - Precedence
   - Associativity
3. Evaluation Design
   - Order of operand evaluation
   - Side effects

Types of Operators

Based on number of operands:

1. Unary operators (single operand)
   - Prefix notation: operator before operand Example: `-x, ++i, !flag`
   - Postfix notation: operator after operand Example: `i++, x--`
2. Binary operators (two operands)
   - Infix notation: operator between operands Example: `a + b, x * y, p / q`
3. Ternary operators (three operands)
   - Example: `condition ? expr1 : expr2` (only common ternary operator in most languages)

Operator Precedence Rules (2.1)

- operator-precedence design is NOT the same across programming languages.
- Operator precedence determines which operator is evaluated first when no parentheses appear.

Operator Precedence Table

Rank	C	Pascal	Python
1	not (!)	not	power (**)
2	*	*	*
3	+	+	+
4	<, >, <=, >=	<, >, =, <>	<, >, <=, >=, ==, !=
5	==, !=	and	not
6	and (&&)	or	and
7	or		or
8	=	:=	=

Are these result the same?

https://onlinegdb.com/Km2_c0xlp

```cpp
#include <iostream>

int main()
{
    int a = 1;
    int b = 6;
    bool c1 = !a*5 == b;
    bool c2 = !(a*5 == b);
    std::cout<< "\n !a*5 == b   ->" << c1  ;
    std::cout<< "\n !(a*5 == b)  ->" << c2  ;

    return 0;
}
```

https://onlinegdb.com/mQ-2N3Tkxt

```python
a = 1
b = 6
print("!a*5 == b" , not a*5 == b)
print("not (a*5 == b)" , not (a*5 == b))
```

Operator Associativity Rules (2.2)

Concept	Controls	Question it answers
Precedence	Which operator goes first	Which operator is evaluated first?
Associativity	Evaluation direction	Left to right or right to left?

```
z = 1 + 2 - 3
z = 3 - 3
console.log(z)
```

- same precedence `+` `-`, then associativity rule `Left to Right` (depends on operator)

```
z = 1 + 2 * 3
z = 1 + 6
console.log(z)
```

- difference precedence `*` (higher) `+` (lower), then associativity rule `Right to Left` (depends on operator)

Info

Most programming languages follow similar operator-precedence rules.

Feature	Java	C	C#	Python
Unary precedence	+, -, ++, –	+, -, ++, –	+, -, ++, –	+, -, ~, ++ (if it existed)
Multiplicative	*, /, %	*, /, %	*, /, %	*, /, //, %
Additive	+, -	+, -	+, -	+, -
Associativity (binary ops)	L → R	L → R	L → R	L → R
Associativity (unary)	R → L	R → L	R → L	R → L
Associativity (`**`)	—	—	—	R → L (**)
Guaranteed operand evaluation order?	Yes (left → right)	No	Yes	Yes

Info
the arithmetic precedence and associativity rules in all major languages (C, C++, Java, C#, Python, Pascal, Ada) are very similar, with only a few differences.

`Note`

Right-associative (Python's way)
2 ** 3 ** 2 → 2 ** (3 ** 2) → 2 ** 9 → 512

Arithemetic Expression Design Issues

1. Type System Design
   - Type conversion rules (coercion hierarchy)
   - Mixed-mode operation support
2. Operator Design
   - Precedence
   - Associativity
3. Evaluation Design
   - Order of operand evaluation
   - Side effects

Operand Evaluation Order (3.1)

https://onlinegdb.com/tqcgS8RGv

```c
#include <stdio.h>
//https://onlinegdb.com/tqcgS8RGv
//19 for c14
//18 for turboc
int main()
{
    int x = 5;
    int y = x++ + ++x * 2;
    printf("x = %d , y = %d", x, y);

    return 0;
}
```

Note

What undefined behavior means:

- Compiler can generate ANY result
- Different compilers → different results (both "correct")
- Same compiler, different flags → different results
- Same compiler, different versions → different results
- Not a bug - the code itself is invalid

Language	Has Operator Side Effects?	Safe?	Example	Notes
C/C++	Yes	Unsafe (undefined behavior)	y=x++ + ++x*2	Lang order unspecified → results unpredictable
Java	Yes	Safe (defined order)	y=x++ + ++x*2	Always evaluated left → right
C#	Yes	Safe (defined order)	y=x++ + ++x*2	Same guarantee as Java
Python	No	Very safe	N/A (++ not allowed)	No operator affects variables inside expressions

Note

What undefined behavior means:

- Compiler can generate ANY result
- Different compilers → different results (both "correct")
- Same compiler, different flags → different results
- Same compiler, different versions → different results
- Not a bug - the code itself is invalid

https://onlinegdb.com/IgHNe1zG8g

```java
public class Main {
    public static void main(String[] args) {
        int x = 5;
        int y = x++ + ++x * 2;

        System.out.println("x = " + x + ", y = " + y);
    }
}
```

✓ Java Rules

- Java always evaluates left → right
- So evaluation is predictable:
  - step1: `x++` → returns 5, `x` becomes 6
  - step2: `++x` → `x` becomes 7, returns 7
  - step3: 5 + 7 * 2 = 19

https://onlinegdb.com/RK8qnvWIN

```
using System;

class Program
{
    static void Main()
    {
        int x = 5;
        int y = x++ + ++x * 2;

        Console.WriteLine($"x = {x}, y = {y}");
    }
}
```

Side Effects in Operands (3.2)

- A variable's value may depend on the order of evaluation when operands have side effects. This problem occurs in imperative languages but not in functional programming.

https://onlinegdb.com/Gj3hwDF5c

```c
#include <stdio.h>

int a = 5;
int fun1(){
    a = 17;
    return 3;
}
int main()
{
    printf("Before call: a = %d\n", a);
    a = a + fun1();
    printf("After call: a = %d\n", a);
    return 0;
}
```

- A side effect of a function occurs when the function modifies one of its parameters or a global variable.

https://onlinegdb.com/uxHHMfBom

```c
#include <stdio.h>
//pass-by-ref "C"
int myfunction(int *A ){
      *A  = *A +10 ;
     return 0;
}
int main( ){
    int A = 20, B=5;
    int result  = A*B + myfunction( &A )  ;
    printf(" result = %d",result );
    return 1;
}
```

- The result in C14 is `20 * 5 = 100`
- The result in TurboC is `30 * 5 = 150`

Solution for modifying parameter side effect

1. define the language by disallowing functional side effects
   - No two-way parameters in functions
   - No non-local references in functions
   - Disadvantage: inflexibility of one-way parameters and lack of non-local references
2. write the language definition to demand that operand evaluation order be fixed
   - Disadvantage: limits some compiler optimizations
   - Java requires that operands appear to be evaluated in left-to-right order

Note

// Well-defined code

A + B + C

- Compiler can evaluate as (A + B) + C or A + (B + C)
- Both are mathematically equivalent and valid
- The language allows flexibility, compiler optimizes
- This is allowed optimization on well-defined code

Language	Guaranteed Left-to-Right Evaluation?	Safe With Side-Effects in Expressions?	A * B + MyFunction(ref A)	Notes
Ada	✅ Yes	⚠️ Safe but should avoid complex side effects	Predictable	Ada strictly defines left-to-right evaluation
C/C++	❌ No	❌ Unsafe (undefined behavior)	Unpredictable	Order of operands often unspecified; modifications cause UB
Java	✅ Yes	👍 Safe (well-defined order)	No support	Java always evaluates left→right for operands & function calls
C#	✅ Yes	👍 Safe	Predictable	C# guarantees left→right operand evaluation
Python	✅ Yes	👍 Safe	No support	Python always evaluates expressions left→right

For C/C++

- C/C++ is not fully defined, which can cause unpredictable results when side effects occur.

The solutions are:

- C/C++ would need major restrictions:
  - no global variable access,
  - no pass-by-reference,
  - everything returns through return values.

For C#

- guaranteed to evaluate from left to right

https://onlinegdb.com/V8oDXBC59

```
using System;

class Program
{
    // pass-by-ref in C#
    static int MyFunction(ref int A)
    {
        A = A + 10;
        return 0;
    }
    static void Main()
    {
        int A = 20, B = 5;
        //20*5 and call
        int result = A * B + MyFunction(ref A) /*+A * B*/ ;
        Console.WriteLine("result = " + result);
    }
}
```

For Java

- It does not support pass by ref method

For Ada

- Not allow modifiy parameter

https://onecompiler.com/ada/447bau984

```
with Ada.Text_IO; use Ada.Text_IO;
with Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Hello is

FUNCTION myfunction ( p :  in out INTEGER  ) return Integer
BEGIN
    Put_Line("DoIt  before - > p = " & Integer'Image (p) );
    p := p + 10;
    Put_Line("DoIt  after - > p = " & Integer'Image (p) );
    return 0;
END myfunction;
```

```
 A : Integer ;
 B  : Integer;
 result :  Integer;
begin

  A := 20;
  B := 5;
  Put_Line("  main before - > A = " & Integer'Image (A) );
  Put_Line("  main before - > B = " & Integer'Image (B) );
  Put_Line("  main before - > result1 = " & Integer'Image (result

end Hello;
```

Function Side Effect: global modifies

- A side effect of a function occurs when the function modifies a global variable

https://onlinegdb.com/ERl7JIKMl

```c
#include <stdio.h>

int a = 5;
int b = 10;
int fun1(){
    a = 17;
    return 3;
}
int main()
{
    printf("Before call: a = %d\n", a);
    a = a*b + fun1();
    printf("After call: a = %d\n", a);
```

The value computed for a in main depends on the order of evaluation of the operands in the expression `a*b + fun1()`.

1. `53` a is evaluated first

2. `173` : if the function call is evaluated first.

Referential transparency

- A function call can be replaced with its return value without changing program behavior.
- Functions have no side effects (no I/O, no state modification, no exceptions).
- Same inputs always produce the same outputs.

https://onlinegdb.com/YW2-7AiS1

```c
#include <stdio.h>

int fun2(int x){
    return x + 5;
}

int main()
{
    int x = 5;
    int y = fun2(x);

    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Relational Expression Design Issues

1. Type System Design
   - Mixed-mode operation support
   - Type conversion rules
   - Result type (boolean)
2. Operator Design
   - Precedence (relative to arithmetic/boolean)
   - Associativity (non-associative in most languages)
3. Evaluation Design
   - Order of operand evaluation
   - Side effects in operands

Relational Operator (1.1)

Operation	Swift	C#	C++	Pascal
Equal	==	==	==	=
Not Equal	!=	!=	!=	<>
Less Than	<	<	<	<
Greater Than	>	>	>	>
Less or Equal	<=	<=	<=	<=
Greater or Equal	>=	>=	>=	>=

Language	Example
C	`bool r = (3 < 4);` // binary `int r = (x > 0) ? x : -x;` //ternary
Python	`r = 3 < 4` # binary `r = x if x > 0 else -x` # ternary #equ to `int r = (x > 0) ? x : -x;`
Standard Pascal	`r := 3 < 4;`

Mixed-mode operation support (1.2)

- Mixed-mode operation is an relation expression where operands have different numeric data types (int, float) , causing the language to perform implicit numeric type conversion (also called numeric promotion) so the operation can be evaluated.

- A mixed-mode operation occurs when an operator (arithmetic or relational) is applied to two operands of different numeric types (such as int, float, double).

```
int x = 5;
float y = 3.14;

if (x < y) {  // Mixed-mode relational expression
    // Step 1: x (int) and y (float) - different types
    // Step 2: Coerce x to float: 5 → 5.0
    // Step 3: Compare: 5.0 < 3.14 → false
    // Step 4: Result is boolean: false
}

// Example 1: int vs float
10 > 9.5     // → 10.0 > 9.5 → true

// Example 2: char vs int (in C)
'A' == 65    // → 65 == 65 → true (char promoted to int)

// Example 3: Different precision
5 == 5.0     // → 5.0 == 5.0 → true

// Example 4: Precision issues!
0.1 + 0.2 == 0.3  // → false (floating-point error!)
```

Language	Mixed-Mode Supported?	Example	Result
C	✓ Yes	3 < 3.14	Int
C++	✓ Yes	3 < 3.14	Int
Java	✓ Yes	3 < 3.14	boolean
C#	✓ Yes	3 < 3.14	boolean
Python	✓ Yes	3 < 3.14	boolean
Go	❌ No	3 < 3.14	Error
Swift	❌ No	3 < 3.14	Error
Rust	❌ No	3 < 3.14	Error

https://onlinegdb.com/YeyNdvdThN

Swift

```swift
let x = 5     // Int
let y = 5.0   // Double

let b = Double (x) == y

print(b)
```

Type conversion rules (1.3)

- When comparing different types, convert to the type that can hold more information (no data loss).
- **Type Hierarchy (Widening Path)**

```
char → int → long → float → double
      (narrower)  →  (wider)
```

Conversion Rules:

1. Integer vs Float/Double

```
int x = 5;
double y = 3.14;

x < y  // Convert: int → double
       // Compare: 5.0 < 3.14 → false
```

Rule: Integer converts to floating-point

2. Different Integer Sizes

```
short a = 10;
long b = 20;

a < b  // Convert: short → long
       // Compare: 10L < 20L → true
```

Rule: Smaller integer converts to larger

3. Float vs Double

```
float x = 3.14f;
double y = 2.71;

x > y  // Convert: float → double
       // Compare: 3.14 > 2.71 → true
```

Rule: Float converts to double

4. Character vs Integer

```
char c = 'A';
int x = 65;

c == x  // Convert: char → int
        // Compare: 65 == 65 → true
```

Rule: Character converts to integer (ASCII/Unicode value)

Result Type Rule:

ALL relational expressions return BOOLEAN (or int in C/C++)

```
// C/C++: returns int (0 or 1)
int result = (5 < 10);  // result = 1

// Java/C#/Python: returns boolean
boolean result = (5 < 10);  // result = true
```

Relational Expression Design Issues

1. Type System Design
   - Mixed-mode operation support
   - Type conversion rules
   - Result type (boolean)
2. Operator Design
   - Precedence (relative to arithmetic/boolean)
   - Associativity (non-associative in most languages)
3. Evaluation Design
   - Order of operand evaluation
   - Side effects in operands

Precedence (2.1)

Language	Higher Precedence	Lower Precedence	Notes
C / C++	< , <= , > , >=	== , !=	Two distinct levels
Java	< , <=, > , >=	==, !=	Same as C/C++ (except instanceof)
C#	<, <= , > , >=	== , !=	Same as C/Java
Python	<, <=, >, >=, == , !=	(Same level)	All comparison operators have the **same precedence**

Info

the relation precedence in all major languages (C, C++, Java, C#, Python, Pascal, Ada) are very similar, with only a few differences.

Associate rule (2.2)

Operator	C	Pascal	Python
*	Left → Right	Left → Right	Left → Right
+	Left → Right	Left → Right	Left → Right
< , > , ==	Left → Right	Left → Right	Chained (special)

C vs Python

https://onlinegdb.com/18nR0S-qm

```cpp
#include <iostream>

using namespace std;

int main()
{
    if(3<4 == 2<3) {  //3<4 == 2<3 will evaluate to true
        cout<<"3<4 == 2<3 => true";
    }
    else {
        cout<<"3<4 == 2<3 => false";
    }
    return 0;
}
```

https://onlinegdb.com/Wpf3Lxtb6

Chained (special)

`(3 < 4) and (4 == 2) and (2 < 3)`

```python
if 3<4 == 2<3 :
    print("3<4 == 2<3 => true")
else :
    print("3<4 == 2<3 => false")
```

Relational Expression Design Issues

1. Type System Design
   - Mixed-mode operation support
   - Type conversion rules
   - Result type (boolean)
2. Operator Design
   - Precedence (relative to arithmetic/boolean)
   - Associativity (non-associative in most languages)
3. Evaluation Design
   - Order of operand evaluation
   - Side effects in operands

Order of operand evaluation and Side effects in operands (3.1, 3.2)

https://onlinegdb.com/gh1xyx_rd

- C/C++ - Unspecified Order `Left-to-Right` or `Right-to-Left`
- Side Effects are `++x` and `x++`

```
int x = 5;
int result = (++x + 1) < (x++ + 2);
//            ^^^^^^^     ^^^^^^^
//              Left        Right
// Which x++ happens first? UNSPECIFIED!

// Possible outcomes:
// Scenario 1: Left first  → 6 < 8 → true  (x becomes 7) Turbo C
// Scenario 2: Right first → 8 < 7 → false (x becomes 7) C14
// UNDEFINED BEHAVIOR!
```

Problem: C/C++ doesn't guarantee left-to-right evaluation!

Java - Left-to-Right (Safe!)

https://onlinegdb.com/ZwrH4Lfsi

```java
public class Main
{
    public static void main(String[] args) {
        int x = 5;
        boolean result = (++x + 1) < (x++ + 2);
        System.out.println("result = " + result +", x = " + x);
    }
}
```

Safe: Java always evaluates left-to-right!

Boolean Expression Design Issues

1. Type System Design
   - Type representation (dedicated boolean vs. numeric)
   - Truthiness rules (if applicable)
2. Operator Design
   - Precedence (AND, OR, NOT hierarchy)
   - Associativity
3. Evaluation Design
   - Short-circuit evaluation
   - Order of evaluation
   - Side effects

Type representation (dedicated boolean vs. numeric) (3.1)

- Modern languages include a Boolean type which represents `true` and `false`.
- It increases readability and writability.
- It does not allow implicit conversion to integer without explicit cast to prevent mixed usage of integers and booleans.

```
boolean flag = true;
```

- Older languages like C (before C99) use `0` for `false` and any non-zero value for `true` (though `1` is conventionally used).

```
int flag = 1;  // true
if (flag) {    // any non-zero is true
    // executes
}
```

Truthiness rules (3.2)

- Truthiness extends beyond simple true/false values to describe how different types of values are evaluated in boolean contexts (like conditional statements). This concept is particularly important in dynamically-typed languages.

Falsy Values

- Most languages that support truthiness define a specific set of values that evaluate to false. Common falsy values include:

https://onlinegdb.com/4yWimDXoF

```
if not "":
    print("Empty string is falsy")  # This executes
if []:
    print("This won't execute")
else:
    print("Empty list is falsy")    # This executes
```

JavaScript and Python both treat these as falsy:

- The boolean `false` itself
- Numeric zero (0, 0.0, -0)
- Empty strings ("" or '')
- `null` (JavaScript) or `None` (Python)
- `undefined` (JavaScript only)
- `NaN` (JavaScript)
- Empty collections (Python: `[]`, `{}`, `set()`)

Truthy Values

Everything that isn't explicitly falsy is truthy.

This includes:

- Non-zero numbers
- Non-empty strings (even "0" or "false")
- Non-empty collections
- Objects and function references

```
if ("0") {
    console.log("String '0' is truthy!");  // Executes
}
if (42) {
    console.log("Non-zero number is truthy!");  // Executes
}
```

```
String '0' is truthy!
Non-zero number is truthy!
```

Boolean Expression Design Issues

1. Type System Design
   - Type representation (dedicated boolean vs. numeric)
   - Truthiness rules (if applicable)
2. Operator Design
   - Precedence (AND, OR, NOT hierarchy)
   - Associativity
3. Evaluation Design
   - Short-circuit evaluation
   - Order of evaluation
   - Side effects

Boolean operator and Precedence level (2.1)

Precedence Level	Swift	C#(Java)	C++	Free Pascal
Hight	!	!	!	not
	&&	&&	&&	and
	||	||	||	or
		&		Non-short-circuit AND
Low		|		Non-short-circuit OR

Boolean Operator Example

Languages	Example	Type
C	bool r = !a;	unary
C	bool r = a && b;	binary
Python	r = not a	unary
Python	r = a and b	binary
Standard Pascal	r := not a;	unary
Standard Pascal	r := a and b;	binary

Boolean Operator Assocative

Precedence	Operator	Associativity	Example	Evaluates As
Highest	! (NOT)	Right → Left	!a && !b	(!a) && (!b)
Medium	&& (AND)	Left → Right	a && b && c	(a && b) && c
Lowest	|| (OR)	Left → Right	a || b || c	(a || b) || c

Boolean Expression Design Issues

1. Type System Design
   - Type representation (dedicated boolean vs. numeric)
   - Truthiness rules (if applicable)
2. Operator Design
   - Precedence (AND, OR, NOT hierarchy)
   - Associativity
3. Evaluation Design
   - Short-circuit evaluation
   - Order of evaluation
   - Side effects

Short Circuit vs Non Short Circuit (3.1)

- Short-Circuit Evaluation:
  - The second operand is not evaluated if the result is already known from the first operand. (optimization)
- Non-short-circuit means:
  - All operands are always evaluated, even if the first operand determines the result.

Example 1:

```
if(10 > 11 && 12 < 15 && 12 == 12)
//     F          T           T
```

- **Short Circuit** : evaluate only 10 > 11 (1 time)
- **Non-short Circuit** : evaluate all (5 times)

Example 2:

```
if(11 > 10 && 12 < 15 || 11 == 12)
//     T          T           F
```

- **Short Circuit** : evaluate 11 > 10 && 12 < 15 (3 times)
- **Non-short Circuit** : evaluate all (5 times)

Non-Short Circuit Evaluation Problem

- C, C++, Java,C#: use short-circuit evaluation for the usual Boolean operators (`&&` and `||`)
- Short-circuit evaluation exposes the potential problem of side effects in expressions

```
let a = 10, b = 8;
if((a>b) || (b++ /3));
console.log(`a = ${a}, b = ${b}`)
```

```
a = 10, b = 8
```

b does not change, so program might not work as expected

Short Circuit Problem

https://onlinegdb.com/VsJJ-d8KF

```c
#include <stdio.h>

int main() {
    int a = 10, b = 8;

    if (a > b || b++ / 3 > 0) {
        printf("a>b || b++ /3 => true\n");
    }

    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

Cannot predict ‘b’ value

Non-Short circuit evaluation Problem

```
index = 0;
while (index < listlen) &&
      (LIST[index] != key)
     index = index + 1;
```

With **non-short-circuit evaluation**:

1. **ALWAYS evaluate BOTH conditions**
2. Both `index < listlen` AND `LIST[index] != key` are checked **every time**

- When `index==listlen`, `LIST[index]` causes an indexing problem (if `LIST` has `listlen-1` elements)

Example with listlen = 5:

```
Iteration 1: index=0, check (0 < 5)=true AND LIST[0] ✓
Iteration 2: index=1, check (1 < 5)=true AND LIST[1] ✓
...
Iteration 5: index=4, check (4 < 5)=true AND LIST[4] ✓
Iteration 6: index=5, check (5 < 5)=FALSE AND LIST[5] ❌ CRASH!
```

Python support non-short circuit

https://onlinegdb.com/91L6PpHuk

```
A = [2, 8, 1, -3, 4]
key = 6
i = 0
while (i < len(A)) & (A[i] != key):
    print("inner i =", i)
    i += 1

print("outer i =", i)
```

Modern Lang. support both short & nonshort operator

- C, C++, Java,C#: use short-circuit evaluation for the usual Boolean operators (&& and ||)
- Python support non short-circuit evaluation
- C# , Java : also support non-short-circuit evaluation for the usual Boolean operators (& and |)
- **JavaScript** supports short circuit and non-short circuit (bit-wise)

Example 1

```
let a = 10, b = 8;
if((a>b) || (b++ /3));
console.log(`short circuit, a = ${a}, b = ${b}`)
if((a>b) | (b++ /3));
console.log(`non-short circuit, a = ${a}, b = ${b}`)
```

```
short circuit, a = 10, b = 8
non-short circuit, a = 10, b = 9
```

Order of evaluation (3.2)

Example 1:

```
not a or b and c
```

Evaluation order:

1. `not a` (NOT has highest precedence)
2. `b and c` (AND has medium precedence)
3. `(not a) or (b and c)` (OR has lowest precedence)
4. **So it evaluates as:** `(not a) or (b and c)`

Example 2:

```
!a && b || c && !d
```

Evaluation order:

1. `!a and !d` (NOT operations first)
2. `(!a) && b` (left AND)
3. `c && (!d)` (right AND)
4. `((!a) && b) || (c && (!d))` (OR last)
5. **Fully parenthesized:** `((!a) && b) || (c && (!d))`

Example 3:

```
a or b and not c
```

Evaluation order:

1. `not c` → (not c)
2. `b and (not c)` → (b and (not c))
3. `a or (b and (not c))`
4. **Final:** `a or (b and (not c))`

Assignment Statements

- Simple Assignments
  - Mostly used `=` but in some languages, ALGOL 60 and Ada use `:=` to avoid confusion for assign value to variable.

Assignment Operator

Language	Assignment Operator	Example	Notes
Swift	=	x = 10	Simple assignment only. ← not used. No :=.
C / C++	=	x = 10;	Standard assignment. Also has compound assignments (+=, -=, etc.).
Pascal	:=	x := 10;	Uses colon-equals for assignment. = is comparison.
COBOL	MOVE ... TO	MOVE 10 TO x.	No operator symbol; assignment uses keywords.

Conditional Targets

- Like Perl, PHP, C/C++, JavaScript, C# ,etc.

https://onlinegdb.com/5gV2yswTgd

```
$flag = 0;
($flag ? $count1 : $count2) = 0;
print "flag = ".$flag." count 1 = " . $count1 .
" count2 = " . $count2 . "\n";

if($flag){
    $count1 = 0;
}else{
    $count2 = 0;
}
print "flag = ".$flag." count 1 = " . $count1 .
" count2 = " . $count2 . "\n";


$flag = 1;
($flag ? $count1 : $count2) = 0;
print "flag = ".$flag." count 1 = " . $count1 .
" count2 = " . $count2 . "\n";

if($flag){
    $count1 = 0;
}else{
    $count2 = 0;
}
```

Conditional expressions by ternary operator ?:

- C-based languages (e.g., C, C++)

```
let average=0.0,sum=10,count=2;
average = (count == 0)? 0 : sum / count
console.log(average)
```

```
5
```

- Evaluates as if written like

```
let average=0.0,sum=10,count=2;
if(count == 0){
   average = 0;
}else{
   average = sum / count;
}
console.log(average)
```

```
5
```

Conditional expressions in Python

https://onlinegdb.com/xCvRo4To_

```
# multiple target assignments
average = sum = count = 1

# conditional expression
average = 0 if count == 0 else sum / count

print(average)
```

Compound Assignment Operators

- A compound assignment operator is a shorthand method of specifying a commonly needed from of assignment.
- In C like language, `sum += value;` is equal to `sum = sum + value;`

Unary Assignment Operators

Unary operators like increment (++) and decrement (–) can appear in two forms:

- Pre-increment (++count): increments first, then uses the new value
- Post-increment (count++): uses the current value, then increments

For example:

- `sum = ++count;` is equivalent to: `count = count + 1; sum = count;`
- `sum = count++;` is equivalent to: `sum = count; count = count + 1;`
- When two unary operators apply to the same operand, the association is right to left.
- `- count ++` is equal to `- (count ++)`

Assignement as an Expression

- In the C-based languages, Perl, and JavaScript, the assiignment statement produces a result, which is the same as the value assigned to the target.

https://onlinegdb.com/GbYKGFkpy

```c
#include <stdio.h>

int main()
{
    char name[30] = {'H','e','l','l','o',',',' ','s','t','u','d','e','n','t','.','\0'};
    char c;
    int i = 0;
    while((c = name[i])!='\0'){
        i++;
    }
    printf("Total characters of \"%s\" = %d",name, i);
    return 0;
}
```

- The disadvantage of assignment as an expresion is hard to read.

https://onlinegdb.com/5a2xCMhF8i

```c
#include <stdio.h>

int main()
{
    int a,b=1,c,d=2;

    a = b + (c=d/b) - 1;
    printf("a = %d, b = %d, c = %d, d = %d",a,b,c,d);
    return 0;
}
```

To explain:

- Assign d / b to c
- Assign b + c to temp
- Assign temp - 1 to a

- In python, allows the effect of multiple-target assignments

https://onlinegdb.com/xCvRo4To_

```
# multiple target assignments
average = sum = count = 1

# conditional expression
average = 0 if count == 0 else sum / count

print(average)
```

- Serveral languages like Perl and Ruby provide multiple-target assignment statements.

https://onlinegdb.com/XOTAs8NL2f

```
($first, $second, $third) = (20,40,60);
print "first = $first, second = $second, third = $third\n";

($first, $second) = ($second, $first);
print "first = $first, second = $second, third = $third\n";
```

- In C and C++, the compiler allows the folling expression which is safety indeficiencies(unsafe, confusing, error-prone).

https://onlinegdb.com/ISm_40Y3f_

```c
#include <stdio.h>

int main()
{
    int x,y=1;
    if(x=y){
        printf("x = %d, y = %d\n", x, y);
    }else{
        printf("x = %d, y = %d\n", x, y);
    }
    return 0;
}
```

Another example

https://onlinegdb.com/SIwlPPEA4

```c
#include <stdio.h>

int main()
{
    int b=10 , d=21 ,  c=1;
    int a = d+c + (c=d/b)-1;
    printf("\nc = %d",c);
    printf("\nb = %d",b);
    printf("\na = %d",a);

    return 0;
}
```

Assignment in Functional Programming

- F# Assignment:
  - For F#, when a variable name is assigned a value, it never changes.
  - To change it, create a new scope with a new binding not related to previous one which is hiding.

https://shorturl.at/OTJOw

```
// This works - nested scope
let demo() =
    let cost = 10 * 5
    printfn "First cost: %d" cost    // 50

    let cost = 20 * 3                // OK - new scope shadows previous
    printfn "Second cost: %d" cost   // 60

demo()
```

Assignment in Functional Programming

- ML Assignment:
  - In ML, `val` does not create a new scope; it creates a new binding that shadows the previous one.

https://shorturl.at/C7RLz

```
val cost = 10 * 5;
val cost = 20 * 3;
print (Int.toString cost ^ "\n");
```

Assignment in Functional Programming

- ML's `val` declarations are often nested in `let` constructs for scoping. (`let...in...end`)

```
val cost = 10 * 5;
print (Int.toString cost ^ "\n");  (* prints 50 *)

val result = let
    val cost = 20 * 3
    val tax = cost div 10
in
    cost + tax
end;

print (Int.toString cost ^ "\n");    (* prints 50 - unchanged! *)
print (Int.toString result ^ "\n");  (* prints 66 *)
```