Unit II
Structuring the Data, Computations and Program

Elementary Data Types : Primitive data Types, Character String types, User Defined Ordinal Types, Array types, Associative Arrays, Record Types, Union Types, Pointer and reference Type.
Expression and Assignment Statements: Arithmetic expression, Overloaded Operators, Type conversions, Relational and Boolean Expressions, Short Circuit Evaluation, Assignment Statements, Mixed mode Assignment.
Statement level Control Statements: Selection Statements, Iterative Statements, Unconditional Branching.
Subprograms: Fundamentals of Sub Programs, Design Issues for Subprograms, Local referencing Environments, Parameter passing methods.
Abstract Data Types and Encapsulation Construct: Design issues for Abstraction, Parameterized Abstract Data types, Encapsulation Constructs, Naming Encapsulations.

Expression and Assignment Statements

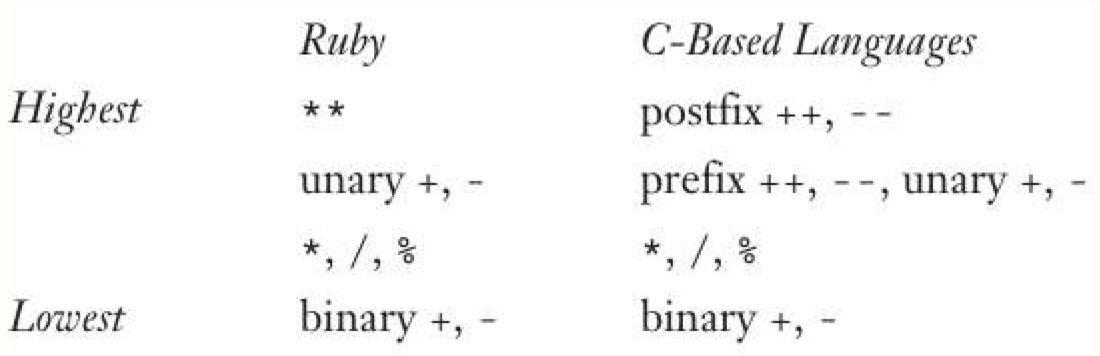
1. Arithmetic expression

- In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.
- An operator can be unary, meaning it has a single operand, binary, meaning it has two operands, or ternary, meaning it has three operands.
- In most programming languages, binary operators are infix, which means they appear between their operands, while in Lisp, it it prefix.

Following are the primary design issues for arithmetic expressions:

A. What are the operator precedence rules?

The precedence of the arithmetic operators of Ruby and the C-based languages are as follows:

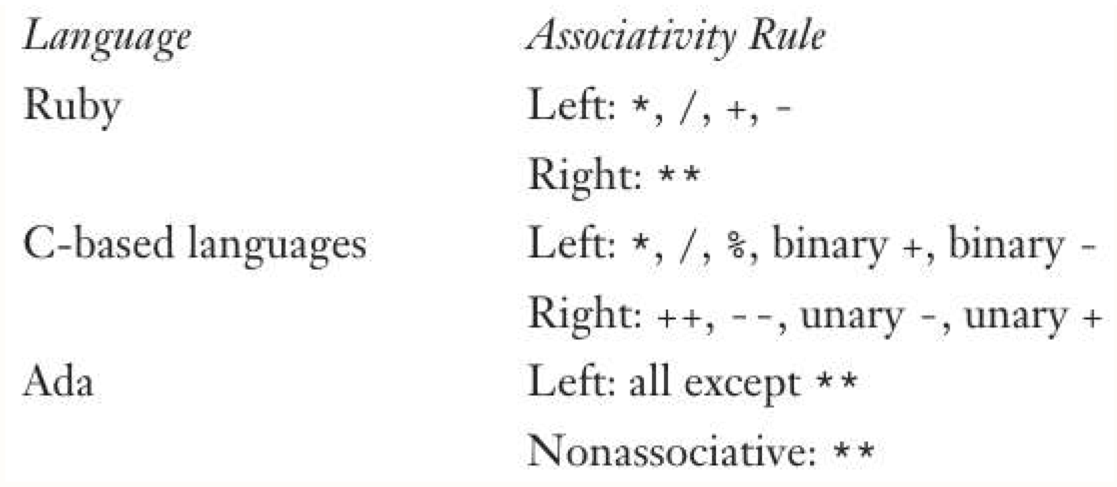


	Ruby	*C-Based Languages*
Highest	**	postfix ++, --
	unary +, -	prefix ++, --, unary +, -
	*, /, %	*, /, %
Lowest	binary +, -	binary +, -

** stands for exponentiation operator.

B. What are the operator associativity rules?

- When an expression contains two adjacent 2 occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the associativity rules of the language.
- Associativity in common languages is left to right, except that the exponentiation operator (when provided) sometimes associates right to left.
- The associativity rules for a few common languages are given here:



Language	*Associativity Rule*
Ruby	Left: *, /, +, -
	Right: **
C-based languages	Left: *, /, %, binary +, binary -
	Right: ++, --, unary -, unary +
Ada	Left: all except **
	Nonassociative: **

2. Overloaded Operators

- Resolution of overloaded operators can be done at translation time.
- For readability purposes, operators are often overloaded.
- For example, + is used for both integer and real addition, * is used for both integer and real multiplication. In each program context, however, it should be clear which specific hardware operation is to be invoked, since integer and real arithmetic differ.
- In a statically typed language, where all variables are bound to their type at translation time, the binding between an overloaded operator and its corresponding machine operation can be established at translation time, since the types of the operands are known.
- This makes the implementation more efficient than in dynamically typed languages, for which it is necessary to keep track of types in run-time descriptor.

- e.g. **+ & -** All these operators are used for multiple purpose.

3. Type conversions

- narrowing conversion e.g. float to int
- widening conversion e.g. int to float

4. Relational and Boolean Expressions

- **A relational operator is an operator that compares the values of its two operands. A relational expression has two operands and one relational operator.**
- **The value of a relational expression is Boolean.**
- **The syntax of the relational operators for equality and inequality differs among some programming la**nguages. For example, for inequality, the C-based languages use != , Ada uses /= , Lua uses ~= , Fortran 95+ uses .NE. or <> , and ML and F# use <> .
- JavaScript and PHP have two additional relational operators, === and !== . These are similar to their relatives, == and !=, but prevent their operands from being coerced. For example, the expression

"7" == 7

is true in JavaScript, because when a string and a number are the operands of a relational operator, the string is coerced to a number. However,

"7" === 7

is false, because no coercion is done on the operands of this operator.

- The relational operators always have lower precedence than the arithmetic operators, so that in expressions such as

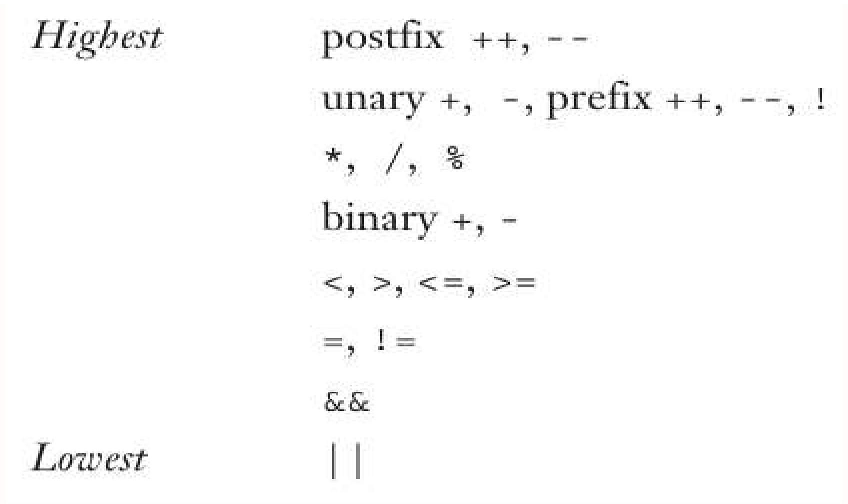
a + 1 > 2 * b

the arithmetic expressions are evaluated first.

- Boolean expressions consist of Boolean variables, Boolean constants, relational expressions, and Boolean operators.

- The operators usually include those for the AND, OR, and NOT operations, and sometimes for exclusive OR and equivalence.
- Boolean operators usually take only Boolean operands (Boolean variables, Boolean literals, or relational expressions) and produce Boolean values.

- The precedence of the arithmetic, relational, and Boolean operators in the C-based languages is as follows:



Highest	postfix ++, --
	unary +, -, prefix ++, --, !
	*, /, %
	binary +, -
	<, >, <=, >=
	=, !=
	&&
Lowest	||

5. Short Circuit Evaluation

- <u>A short-circuit evaluation</u> of an expression is one in which the <u>result is determined without evaluating all of the operands and/or operators</u>.
- For example, the value of the arithmetic expression

  (13 * a) * (b / 13 - 1)

  Is independent of the value of (b / 13 - 1) if a is 0 , because 0 * x = 0 for any x.

- So, when a is 0, there is no need to evaluate (b / 13 - 1) or perform the second multiplication.
- However, in arithmetic expressions, this shortcut is not easily detected during execution, so it is never taken.
- The value of the Boolean expression

  (a >= 0) && (b < 10)

  Is independent of the second relational expression if a < 0, because the expression (FALSE && (b < 10)) is FALSE for all values of b.

- A language that provides short-circuit evaluations of Boolean expressions and also has <u>side effects</u> in expressions <u>allows **subtle errors** to occur</u>.

- Suppose that short-circuit evaluation is used on an expression and part of the expression that contains a side effect is not evaluated; then the side effect will occur only in complete evaluations of the whole expression.
- If program correctness depends on the side effect, short-circuit evaluation can result in a serious error.

- For example, consider the **Java** expression

(a > b) || ((b++) / 3)

- In this expression, b is changed (in the second arithmetic expression) only when a <= b. If the programmer assumed b would be changed every time this expression is evaluated during execution (and the program's correctness depends on it), the program will fail.
- In the **C-based languages**, the usual AND and OR operators, && and || , respectively, are short-circuit.
- **Ada** allows the programmer to specify short-circuit evaluation of the Boolean operators AND and OR by using the two-word operators **and then** and **or else** . Ada also has non–short-circuit operators, **and** and **or** .

6. Assignment Statements

Simple Assignments

i. Nearly all programming languages currently being used use the equal sign for the assignment operator.

ii. ALGOL 60 and Ada makes use of := as the assignment operator.

iii. In some languages, such as FORTRAN and Ada, an assignment can appear only as a stand-alone statement, and the destination is restricted to a single variable.

Conditional Targets

Perl allows conditional targets on assignment statements.

For example, consider

```
($flag ? $count1 : $count2) = 0;
```

which is equivalent to

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

Compound Assignment Operators

The form of assignment that can be abbreviated with this technique has the destination variable also appearing as the first operand in the expression on the right side, as in

```
a = a + b
```

By Compound Assignment Operator, above statement is written as follows:

```
a+=b
```

It is provided by ALGOL 68, C-based languages, Perl, JavaScript, Python, and Ruby.

Unary Assignment Operators

The C-based languages, Perl, and JavaScript include two special unary arithmetic operators that are actually abbreviated assignments. They combine increment and decrement operations with assignment.

In the assignment statement

```
sum = ++ count;
```

It is similar to

```
count = count + 1;
sum = count;
```

If the same operator is used as a postfix operator, as in

```
sum = count ++;
```

It is similar to

```
sum = count;
count = count + 1;
```

```
count ++
```

It is similar to

```
count=count+1
```

When two unary operators apply to the same operand, the association is right to left. For example, in

```
- count ++
```

count is first incremented and then negated. So, it is equivalent to

- (count ++)

Assignment as an Expression

Expression is evaluated and then it is assigned.

e.g.

1. while ((ch = getchar()) != EOF) { ... }

2. a = b + (c = d / b) - 1

Multiple Assignments

- Some programming languages like Perl, Ruby, and Lua supports multiple-target, multiple-source assignment statements.
- For example, in Perl one can write

($first, $second, $third) = (20, 40, 60);

- Here, $first is assigned 20, $second is assigned 40 and $third is assigned 60.

7. Mixed mode Assignment

One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types.

Languages that allow such expressions, which are called mixed-mode expressions, must define conventions for implicit operand type conversions because computers do not have binary operations that take operands of different types.

e.g.

```
int a=4;

float b=3.14;

float c;

c= a+b;
```