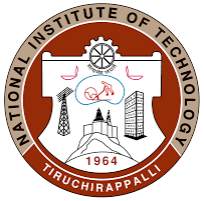


CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
~~5.~~	~~Names, Binding, Type Checking and Scopes~~
6.	Data Types
7.	**Expressions and Assignment Statements**
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 7 – Expressions and Assignment Statements

Objectives

- Semantics rules that determine the order of evaluation of operators in expressions
- Potential problem with operand evaluation order, when functions can have side effects
- Overloaded operators
- Mixed-mode expressions
- Relational and Boolean expressions
- Assignment statement

Introduction

- Fundamental means of specifying computations in a programming language
- Crucial for a programmer to understand both the syntax and semantics of expressions
- To understand expression evaluation, it is necessary to be familiar with the orders of operator and operand evaluation
- Operator evaluation order of expressions is dictated by the associativity and precedence rules

$$(A + B) + C = A + (B + C)$$

$$A + B * C \quad [* \uparrow \quad + \downarrow]$$

- Order of operand evaluation in expressions is often unstated by language designers
  - Allows implementors to choose the order, which leads to the possibility of programs producing different results in different implementations

Introduction

- Other issues -> Type Mismatches, Coercions, and Short-circuit evaluation
- Essence of imperative programming languages is the dominant role of assignment statements
- Purpose of these statements is to cause the side effect of changing the values of variables, or the state, of the program
    - Integral part of all imperative languages is the concept of variables whose values change during program execution
- Functional languages use variables of a different sort -> Parameters of functions
    - Also have declaration statements that bind values to names
    - These declarations are similar to assignment statements, but do not have side effects

Miscellaneous

```
#include<iostream.h>
#include<conio.h>
 #include<stdlib.h>
int add(int, int);
int subtract(int, int);
void main()
{
   int a = 5, b = 10;
   int Summation = add(a, b);
}
```

```
int add(int x, int y)
{
    return subtract(x, y);
}
int subtract(int w, int v)
{
    return (w – v);
}
```

Arithmetic Expressions

- Arithmetic expressions consist of operators, operands, parentheses and function calls **(a + b) + fun(c, d)**
- An operator can be:
  - Unary -> Has single operand
  - Binary -> Has two operands
  - Ternary -> Has three operands
- In most programming languages, binary operators are infix, which means they appear between their operands a + b
- Purpose of an arithmetic expression is to specify an arithmetic computation
- An implementation of such a computation must cause two actions -> fetching the operands + Executing arithmetic operations on those operands

Design Issues

- What are the operator precedence rules?
- What are the operator associativity rules?
- What is the order of operand evaluation?
- Are there restrictions on operand evaluation side effects?
- Does the language allow user-defined operator overloading?
- What type mixing is allowed in expressions?

Operator Evaluation Order

- Operator Precedence and Associativity Rules

Precedence

a + b * c *[a = 3; b = 4; c = 5; L-> R => 35; R -> L => 23]*

- Mathematicians already defined -> Multiplication is considered to be of higher priority than addition, perhaps due to its higher level of complexity
- Operator precedence rules
  - Rules of the common imperative languages are nearly all the same
  - Exponentiation has the highest precedence, followed by multiplication and division on the same level, followed by binary addition and subtraction on the same level
- Unary addition is called the identity operator -> Has no associated operation and thus has no effect on its operand
- Unary minus changes the sign of its operand

Operator Evaluation Order

- Java and C#, unary minus also causes the implicit conversion of short (16-bits) and byte (8-bits) operands to int (32-bits) type

  a + (- b) * c => a + (b -) * c -> Legal

  a + - b * c -> Illegal

- First two, relative precedence of the unary minus operator and the binary operator is irrelevant

  - a / b

  - a * b

  - a ** b

- Exponentiation has higher precedence than unary minus

  - a ** b => - (a ** b)

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

Operator Evaluation Order

Associativity

a - b + c - d

- If addition and subtraction operators have the same level of precedence, the precedence rules say nothing about the order of evaluation of the operators
- Associativity Rules -> Left or Right associativity
- Associativity in common languages is left to right, except that the exponentiation operator sometimes associates right to left

a - b + c -> Left operator is evaluated first

- Exponentiation in Fortran and Ruby is right associative

A ** B ** C

- Ada, exponentiation is nonassociative

(A ** B) ** C => A ** (B ** C)

Operator Evaluation Order

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

- Many compilers for the common languages make use of the fact that some arithmetic operators are mathematically associative -> Arithmetic Addition

  A + B + C + D => [1000000 + (-5900055) + 2000000 + (-499494949)]
- A and C are very large positive numbers, and B and D are negative numbers with very large absolute values
- Adding B to A does not cause an overflow exception, but adding C to A does
- Likewise, adding C to B does not cause overflow, but adding D to B does
- Because of the limitations of computer arithmetic, addition is catastrophically nonassociative in this case
- Therefore, if the compiler reorders these addition operations, it affects the value of the expression

Operator Evaluation Order

Parantheses

- Programmers can alter the precedence and associativity rules by placing parentheses in expressions
- Parenthesized part of an expression has precedence over its adjacent unparenthesized parts

(A + B) * C

  - First operand of the multiplication operator is not available until the addition in the parenthesized sub expression is evaluated

A + B + C + D => (A + B) + (C + D) // Avoids Overflow

Operator Evaluation Order

- Languages that allow parentheses in arithmetic expressions could dispense with all precedence rules and simply associate all operators left to right or right to left
- Programmer would specify the desired order of evaluation with parentheses
- Simple because neither the author nor the readers of programs would need to remember any precedence or associativity rules
- Disadv: Makes writing expressions more tedious, and it also seriously compromises the readability of the code

Operator Evaluation Order

Conditional Expressions

- if-then-else statements can be used to perform a conditional expression assignment

```
if (count == 0)
  average = 0;
else
  average = sum / count;
```

```
expression_1 ? expression_2 : expression_3

average = (count == 0) ? 0 : sum / count;
```

  - where expression_1 is interpreted as a Boolean expression
  - Question mark -> Beginning of the then clause
  - Colon marks -> Beginning of the else clause
- Both clauses are mandatory
- ? is used in conditional expressions as a ternary operator

Operand Evaluation Order

- Variables in expressions are evaluated by fetching their values from memory
- Constants are sometimes evaluated the same way
- Constant may be part of the machine language instruction and not require a memory fetch A + 80 => ADD A, #80
- If an operand is a parenthesized expression, all of the operators it contains, must be evaluated before its value can be used as an operand (A + B – C * D) / F
- If neither of the operands of an operator has side effects, then operand evaluation order is irrelevant
- Only interesting case arises when the evaluation of an operand does have side effects

Operand Evaluation Order

Side Effects

- Side effect of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable

a + fun(a)

- If fun does not have the side effect of changing a, then the order of evaluation of the two operands, a and fun(a), has no effect on the value of the expression
- However, if fun changes a, there is an effect

```
a = 10;
b = a + fun(a);
```

Eg: fun(a) returns value 10 as o/p and also changes the value of a to 20.

O/P: 20 (L -> R); 30 (R -> L)

```
int a = 5;
int fun1() {
  a = 17;
  return 3;
}  /* end of fun1 */
void main() {
  a = a + fun1();
}  /* end of main */
```

Operand Evaluation Order

- Two possible solutions to the problem -> Operand evaluation order and side effects
    - Language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects
    - Language definition could state that operands in expressions are to be evaluated in a particular order and demand that implementors guarantee that order
- Disallowing functional side effects in the imperative languages is difficult, and it eliminates some flexibility for the programmer
- Consider the case of C and C++, which have only functions, meaning that all subprograms return one value
    - To eliminate the side effects of two-way parameters and still provide subprograms that return more than one value, the values would need to be placed in a struct and the struct returned
- Access to globals in functions would also have to be disallowed
- However, when efficiency is important, using access to global variables to avoid parameter passing is an important method of increasing execution speed
- In compilers, for example, global access to data such as the symbol table is commonplace

Operand Evaluation Order

- Problem with having a strict evaluation order is that some code optimization techniques used by compilers involve reordering operand evaluations
- A guaranteed order disallows those optimization methods when function calls are involved
- No perfect solution, as is borne out by actual language designs
- Java language definition guarantees that operands appear to be evaluated in left-to-right order, eliminating the problem

Operand Evaluation Order

Referential Transparency and Side Effects

- Concept of referential transparency is related to and affected by functional side effects
- A program has the property of referential transparency if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program
- Value of a referentially transparent function depends entirely on its parameters

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

- If *fun* changes the value of *a*, then that side effect violates the referential transparency of the program in which the code appears

Operand Evaluation Order

- Advantages to referentially transparent programs
  - Semantics of such programs is much easier to understand
  - Makes a function equivalent to a mathematical function, in terms of ease of understanding
- Because they do not have variables, programs written in pure functional languages are referentially transparent
- Functions in a pure functional language cannot have state, which would be stored in local variables
- If such a function uses a value from outside the function, that value must be a constant, since there are no variables
- Therefore, the value of the function depends on the values of its parameters

Overloaded Operators

- Arithmetic operators are often used for more than one purpose
  - "+" -> Addition, Subtraction, Concatenation
- Multiple use of an operator is called operator overloading and is generally thought to be acceptable, as long as neither readability nor reliability suffers
- "&" -> Binary Operator (Logical AND); Unary Operator (Address)

```
x = &y;
```

- Problems:
  - Using the same symbol for two completely unrelated operations is detrimental to readability
  - Simple keying error can go undetected
- "-" Operator

Overloaded Operators

- Some languages allow the programmer to further overload operator symbols
- Suppose a user wants to define the * operator between a scalar integer and an integer array

```
a * b     // a = 5; b is as int array containing 5 elements
```

- + and * are overloaded for a matrix abstract data type and A, B, C, and D are variables of that type

```
A * B + C * D  => MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))
```

- C++ has a few operators that cannot be overloaded
  - Class or structure member operator (.) and the scope resolution operator (::)
- Operator overloading was one of the C++ features that was not copied into Java
- Reappear in C#

Type Conversions

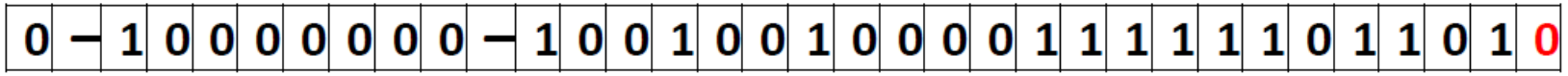
- Type conversions are either narrowing or widening
- A narrowing conversion converts a value to a type that cannot store even approximations of all of the values of the original type. Eg: Double (64-bits) to Float (32-bits)
- A widening conversion converts a value to a type that can include at least approximations of all of the values of the original type. Eg: int (32-bit) to float (32-bit)
- Widening conversions are nearly always safe, meaning that the magnitude of the converted value is maintained
- Narrowing conversions are not always safe -> Sometimes the magnitude of the converted value is changed in the process. Eg: Converting Float value 1.3E25 to Integer

Type Conversions

- Although widening conversions are usually safe, they can result in reduced accuracy
- In many language implementations, although integer-to-floating-point conversions are widening conversions, some precision may be lost
- Integers are stored in 32-bits, which allows at least nine decimal digits of precision

$$0 \text{ to } 2^{32} - 1 = 0 \text{ to } 429496729$$

- Floating-point values are also stored in 32 bits, with only about seven decimal digits of precision



0. 1415926**6**53589793 *(Obtained Value using Calc)*

Fraction Part of Pi = 0.1415925**5**02593994140625

- Integer-to-floating-point widening can result in the loss of two digits of precision

Type Conversions

Coercion in Expression

- 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
- Coercion -> Implicit type conversion that is initiated by the compiler
- Type conversions explicitly requested by the programmer are referred to as explicit conversions, or casts, not coercions
- For overloaded operators in a language that uses static type binding, the compiler chooses the correct type of operation on the basis of the types of the operands

Type Conversions

- When the two operands of an operator are not of the same type and that is legal in the language, the compiler must choose one of them to be coerced and supply the code for that coercion
- Issues
  - Reliability problems -> Reduce the benefit of type checking
  - Whether programmers should be concerned with this category of errors or whether the compiler should detect them

```
int a;
float b, c, d;
. . .
d = b * a;    // “a” is a typo. “c” must be there
```

Type Conversions

- Ada allows very few mixed type operands in expressions
- It does not allow mixing of integer and floating-point operands in an expression, with one exception
  - Exponentiation operator, **, can take either a floating-point or an integer type for the first operand and an integer type for the second operand
- C-based languages have integer types that are smaller than the int (32-bits) Type -> Byte (8-bits) and Short (16-Bits)
- Operands of all of these types are coerced to int whenever virtually any operator is applied to them
- Data can be stored in variables of these types, it cannot be manipulated before conversion to a larger type

Type Conversions

```
byte a, b, c;
...
a = b + c;
```

- Values of b and c are coerced to int and an int addition is performed
- Sum is converted to byte and put in “a”
- Given the large size of the memories of contemporary computers, there is little incentive to use byte and short, unless a large number of them must be stored

Type Conversions

Explicit Type Conversions

- Most languages provide some capability for doing explicit conversions, both widening and narrowing
- Warning messages are produced when an explicit narrowing conversion results in a significant change to the value of the object being converted
- Explicit type conversions are called casts
- To specify a cast, the desired type is placed in parentheses just before the expression to be converted

(int) angle

- Reason for having parantheses -> Two-word type names. Eg: *long int (at least 4 Bytes, usually 8 Bytes)*

Type Conversions

Errors in Expressions

- Number of errors can occur during expression evaluation
- If the language requires type checking, either static or dynamic, then operand type errors cannot occur
- Other kinds of errors are due to the limitations of computer arithmetic and the inherent limitations of arithmetic
- Most common error occurs when the result of an operation cannot be represented in the memory cell where it must be stored
- Overflow or underflow, depending on whether the result was too large or too small
- One limitation of arithmetic is that division by zero is disallowed
- Floating-point overflow, underflow, and division by zero are examples of run-time errors -> Exceptions

Relational and Boolean Expressions

- Addition to Arithmetic Expressions, Programming Languages support Relational and Boolean Expressions

Relational Expressions

- Operator that compares the values of its two operands
- Relational expression has two operands and one relational operator. Eg: a > b
- Value of a relational expression is Boolean, except when Boolean is not a type included in the language
- Relational operators always have lower precedence than the arithmetic operators
  - a + 1 > 2 * b -> Arithmetic expressions are evaluated first

Boolean Expressions

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

  const int true = 1

  const int false = 0

- Boolean Operators -> AND, OR, NOT operations, and sometime Exclusive OR and Equivalence
- Boolean operators usually take only Boolean operands (Boolean variables, Boolean literals or relational expressions) and produce Boolean values
- C-based languages assign a higher precedence to AND than OR *(AND -> Multiplication; OR -> Addition)*

Boolean Expressions

- Arithmetic expressions can be the operands of relational expressions

$$100 + 5 > 5 - 15$$

- Relational expressions can be the operands of Boolean expressions

$$a > b > c$$

- Three categories of operators must be placed in different precedence levels, relative to each other

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

Boolean Expressions

0 -> False; 1 -> True

Perl and Ruby -> *&&* and *and* -> AND; *||* and *or* -> OR

- One difference between *&&* and *and* (and *||* and *or*) is that the spelled versions (*and* and *or*) have lower precedence
- Also, *and* and *or* have equal precedence, but *&&* has higher precedence than *||*
- C-based languages -> More than 40 non-arithmetic operators and at least 14 different levels of precedence
    - Evidence of the richness of the collections of operators and the complexity of expressions possible in these languages

Boolean Expressions

- Readability dictates that a language should include a Boolean type rather than simply using numeric types in Boolean expressions

```
bool b = True;
```

```
int b = 1;
```

- Some error detection is lost in the use of numeric types for Boolean operands because any numeric expression, whether intended or not, is a legal operand to a Boolean operator

```
int b = 10;
```

```
b + 5 > 5 + 10;
```

- In some imperative languages, any non-Boolean expression used as an operand of a Boolean operator is detected as an error

Short-Circuit Evaluation

- Result is determined without evaluating all of the operands and/or operators

(13 * a) * (b / 13 - 1) // a = 0

- In arithmetic expressions, this shortcut is not easily detected during execution

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

- This shortcut can be easily discovered during execution

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

- If evaluation is not short-circuit, and if key is not in list, the program will terminate with a subscript out-of-range exception

Short-Circuit Evaluation

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

- Suppose, if the number of iteration is equal to the listlen, then the condition (index < listlen) will fail
  - Suppose if the programming language facilitates short-circuit evaluation then the second condition (list[index] != key) will not be executed and hence the execution will successfully come out of the loop
  - However, if the programming language does not facilitate short-circuit evaluation, then the condition (list[index] != key) will try to access the element at position “index” in the array “list”. This throws the exception “out-of-range”

Short-Circuit Evaluation

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

- 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
- Ada:
  - Short-circuit operators -> *and then* and *or else*
  - Non-short-circuit operators -> *and* and *or*
- *C:*
  - Usual AND and OR operators, && and ||, respectively, are short-circuit
  - Bitwise AND and OR operators, & and |, respectively, that can be used on Boolean-valued operands and are not short-circuit

Assignment Statements

- Assignment statement is one of the central constructs in imperative languages
- Provides the mechanism by which the user can dynamically change the bindings of values to variables

```
int a = 5;              // Static Binding
int a = b + 10 – 5      // Dynamic Binding
```

Simple Assignments

- Assigment Operator vs Equality Relational Operator

= or := vs ==

Conditional Targets

Perl -> ($flag ? $count1 : $count2) = 0;

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

Assignment Statements

Compound Assignment Operators

- Destination variable also appearing as the first operand in the expression on the right side

```
a = a + b
```

- Syntax of these assignment operators is the catenation of the desired binary operator to the = operator

```
a += b
```

Unary Assignment Operator

- Combine increment and decrement operations with assignment

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

Assignment Statements

- When two unary operators apply to the same operand, the association is right to left

  - count ++ => - (count ++)

Assignment as an Expression

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

- Disadvantage of allowing assignment statements to be operands in expressions is that it provides yet another kind of expression side effect
- This type of side effect can lead to expressions that are difficult to read and understand

```
a = b + (c = d / b) - 1   =>   Assign d / b to c
                               Assign b + c to temp
                               Assign temp - 1 to a
```

- Multiple-target assignments, sum = count = 0;
- Loss of error detection in the C design of the assignment operation that frequently leads to program errors

  if (x == y) ≠ if (x = y)

Assignment Statements

Multiple Assignments

- Multiple-target, multiple-source assignment statements

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

- If the values of two variables must be interchanged

```
($first, $second) = ($second, $first);
```

- This correctly interchanges the values of $first and $second, without the use of a temporary variable (at least one created and managed by the programmer)

Mixed-Mode Assignment

- Frequently, assignment statements also are mixed mode
- Design question is: Does the type of the expression have to be the same as the type of the variable being assigned, or can coercion be used in some cases of type mismatch?

```
int a = 5, b = 10;
float c = a / b;
```

- Fortran, C, C++, and Perl use coercion rules for mixed-mode assignment that are similar to those they use for mixed-mode expressions
  - Many of the possible type mixes are legal, with coercion freely applied
- C++, Java and C# allow mixed-mode assignment only if the required coercion is widening. Eg: int (32-bit) value can be assigned to a float (32-bit) variable, but not vice versa

Miscellaneous

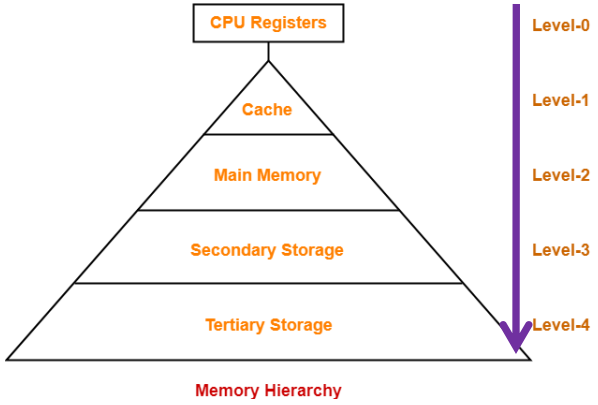
```
#include <stdio.h>
void main()
{
    int a = 5, b = 10;
    float c = a / b;
    printf("%f", c);
}
```

O/P: 0.000000

```
#include <stdio.h>
void main()
{
    int a = 5, b = 10;
    float c = (float) a / (float) b;
    printf("%f", c);
}
```

O/P: 0.500000

Miscellaneous



Cost-per-bit ↓
Access Time ↑

Main Memory or RAM *(DRAM -> Capacitor)*

File Manager

Direct Memory Access

Secondary Memory *(Hard Disk)*

Register 1

Register 2

Central Processing Unit *(Arithmetic Logic Operation)*

Cache Memory *(SRAM -> Flip Flop)*

Register n

Tertiary Memory *(Magnetic Tape)*

Internal Memory -> Registers, Cache Memory