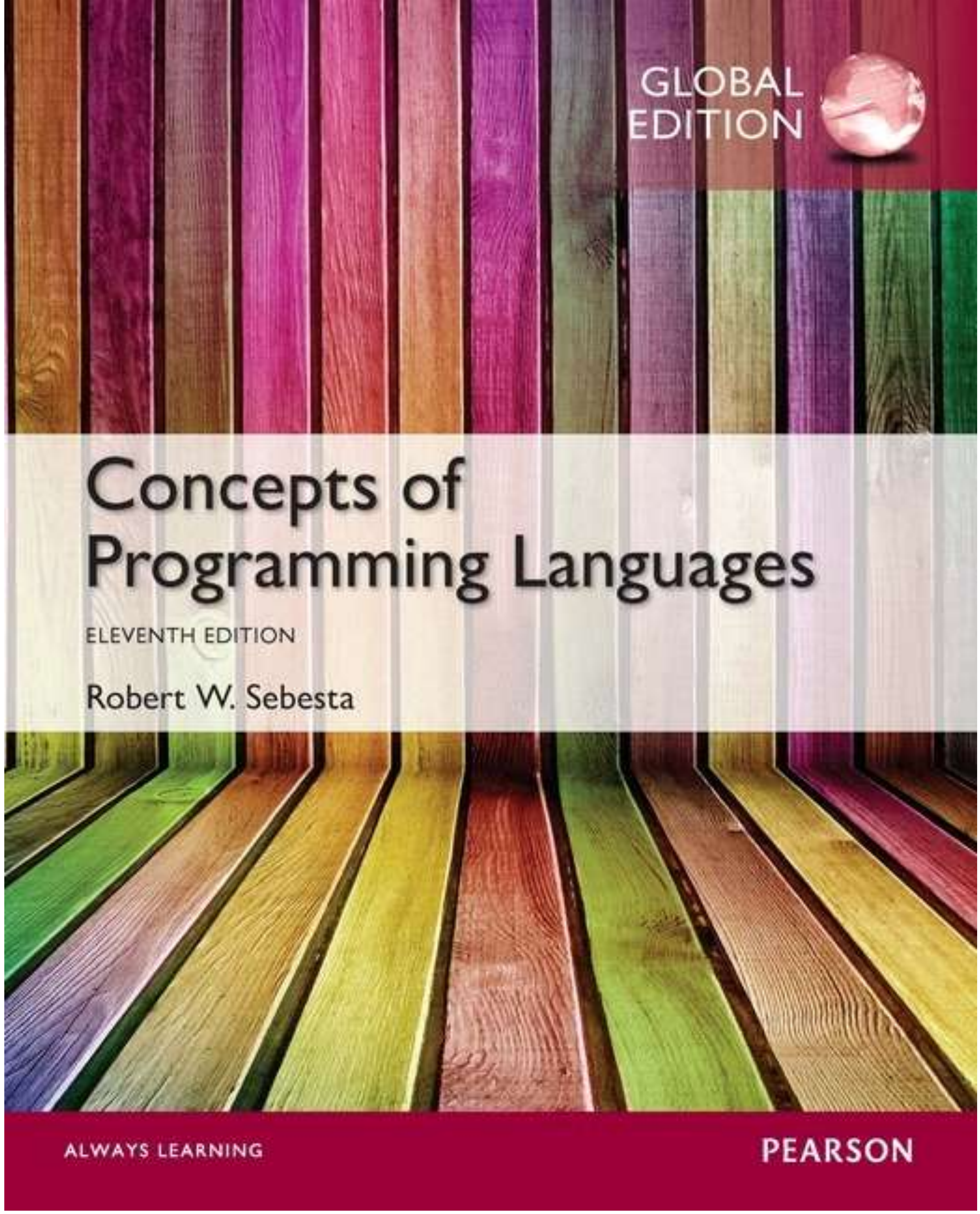
Chapter 5

Names, Bindings, and Scopes



Chapter 5 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
- Variables are characterized by attributes
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

- Design issues for names:
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Names (continued)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C# and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Names (continued)

- Special characters
  - PHP: all variable names must begin with dollar signs
  - Perl: all variable names begin with special characters, which specify the variable’s type
  - Ruby: variable names that begin with `@` are instance variables; those that begin with `@@` are class variables

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

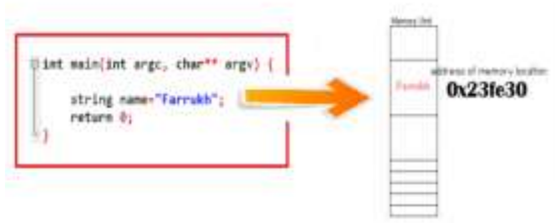
Names (continued)

- Special words
  - An aid to readability; used to delimit or separate statement clauses
  - A *keyword* is a word that is special only in certain contexts
  - A *reserved word* is a special word that cannot be used as a user-defined name
  - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

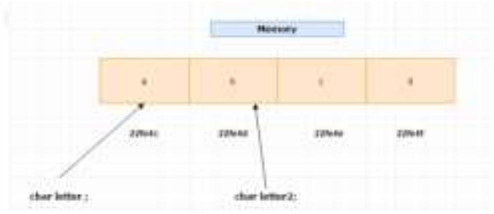
Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Variables Attributes



```
int main(int argc, char** argv) {

    string name="Farrukh";
    return 0;
}
```



- Name – not all variables have them
- Address – the memory address with which it is associated
  - A variable may have different addresses at different times during execution
  - A variable may have different addresses at different places in a program
  - If two variable names can be used to access the same memory location, they are called aliases
  - Aliases are created via pointers, reference variables, C and C++ unions
  - Aliases are harmful to readability (program readers must remember all of them)

Variables Attributes (continued)

- *Type* – determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
- *Value* – the contents of the location with which the variable is associated
  - The l-value of a variable is its address
  - The r-value of a variable is its value
- *Abstract memory cell* – the physical cell or collection of cells associated with a variable

The Concept of Binding

A *binding* is an association between an entity and an attribute, such as between a variable and its type or value, or between an operation and a symbol

- *Binding time* is the time at which a binding takes place.

Possible Binding Times

- Language design time -- bind operator symbols to operations
- Language implementation time-- bind floating point type to a representation
- Compile time -- bind a variable to a type in C or Java
- Load time -- bind a C or C++ `static` variable to a memory cell)
- Runtime -- bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Type Binding

- How is a type specified?
- When does the binding take place?
- If static, the type may be specified by either an explicit or an implicit declaration

Explicit/Implicit Declaration

- An *explicit declaration* is a program statement used for declaring the types of variables
- An *implicit declaration* is a default mechanism for specifying types of variables through default conventions, rather than declaration statements
- Basic, Perl, Ruby, JavaScript, and PHP provide implicit declarations
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Explicit/Implicit Declaration (continued)

- Some languages use type inferencing to determine types of variables (context)
  - C# – a variable can be declared with **`var`** and an initial value. The initial value sets the type
  - Visual Basic 9.0+, ML, Haskell, and F# use type inferencing. The context of the appearance of a variable determines its type

Dynamic Type Binding

- Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited))
- Specified through an assignment statement e.g., JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult

Variable Attributes (continued)

- Storage Bindings & Lifetime
  - Allocation – getting a cell from some pool of available cells
  - Deallocation – putting a cell back into the pool
- The lifetime of a variable is the time during which it is bound to a particular memory cell

Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables in functions
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)

Categories of Variables by Lifetimes

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are *elaborated*.

  (A declaration is elaborated when the executable code associated with it is executed)
- If scalar, all attributes except address are statically bound
  - local variables in C subprograms (not declared **`static`**) and Java methods
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Categories of Variables by Lifetimes

- *Explicit heap-dynamic* -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- Referenced only through pointers or references, e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java
- Advantage: provides for dynamic storage management
- Disadvantage: inefficient and unreliable

Categories of Variables by Lifetimes

- *Implicit heap-dynamic*--Allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
- The *local variables* of a program unit are those that are declared in that unit
- The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there
- *Global variables* are a special category of nonlocal variables
- The scope rules of a language determine how references to names are associated with variables

Static Scope

- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*
- Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python)

Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name

Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Example in C:

```
void sub() {
  int count;
  while (...) {
    int count;
    count++;
    ...
  }
  …
}
```

- Note: legal in C and C++, but not in Java and C# – too error-prone

The `LET` Construct

- Most functional languages include some form of `let` construct
- A let construct has two parts
  - The first part binds names to values
  - The second part uses the names defined in the first part
- In Scheme:

```
(LET (
  (name_1 expression_1)
  …
  (name_n expression_n)
)
```

The LET Construct (continued)

- In ML:

```
let
  val name₁ = expression₁
  …
  val nameₙ = expressionₙ
in
 expression
end;
```

- In F#:
  - First part: `let` left_side = expression
  - (left_side is either a name or a tuple pattern)
  - All that follows is the second part

Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - However, a variable still must be declared before it can be used

Declaration Order (continued)

- In C++, Java, and C#, variables can be declared in `for` statements
  - The scope of such variables is restricted to the `for` construct

Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
  - These languages allow variable declarations to appear outside function definitions

- C and C++have both declarations (just attributes) and definitions (attributes and storage)
  - A declaration outside a function definition specifies that it is defined in another file

Global Scope (continued)

- PHP
  - Programs are embedded in HTML markup documents, in any number of fragments, some statements and some function definitions
  - The scope of a variable (implicitly) declared in a function is local to the function
  - The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
    - Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`

Global Scope (continued)

- Python
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

Evaluation of Static Scoping

- Works well in many situations
- Problems:
  - In most cases, too much access is possible
  - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Scope Example

```
function big() {        big calls sub1
    function sub1()     sub1 calls sub2
       var x = 7;       sub2 uses x
    function sub2() {
      var y = x;
    }
   var x = 3;
 }
```

- Static scoping
  - Reference to `x` in `sub2` is to `big`'s `x`
- Dynamic scoping
  - Reference to `x` in `sub2` is to `sub1`'s `x`

Scope Example

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - *Disadvantages:*
    1. While a subprogram is executing, its variables are visible to all subprograms it calls
    2. Impossible to statically type check
    3. Poor readability– it is not possible to statically determine the type of a variable

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a `static` variable in a C or C++ function

Referencing Environments

- The *referencing environment* of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Named Constants

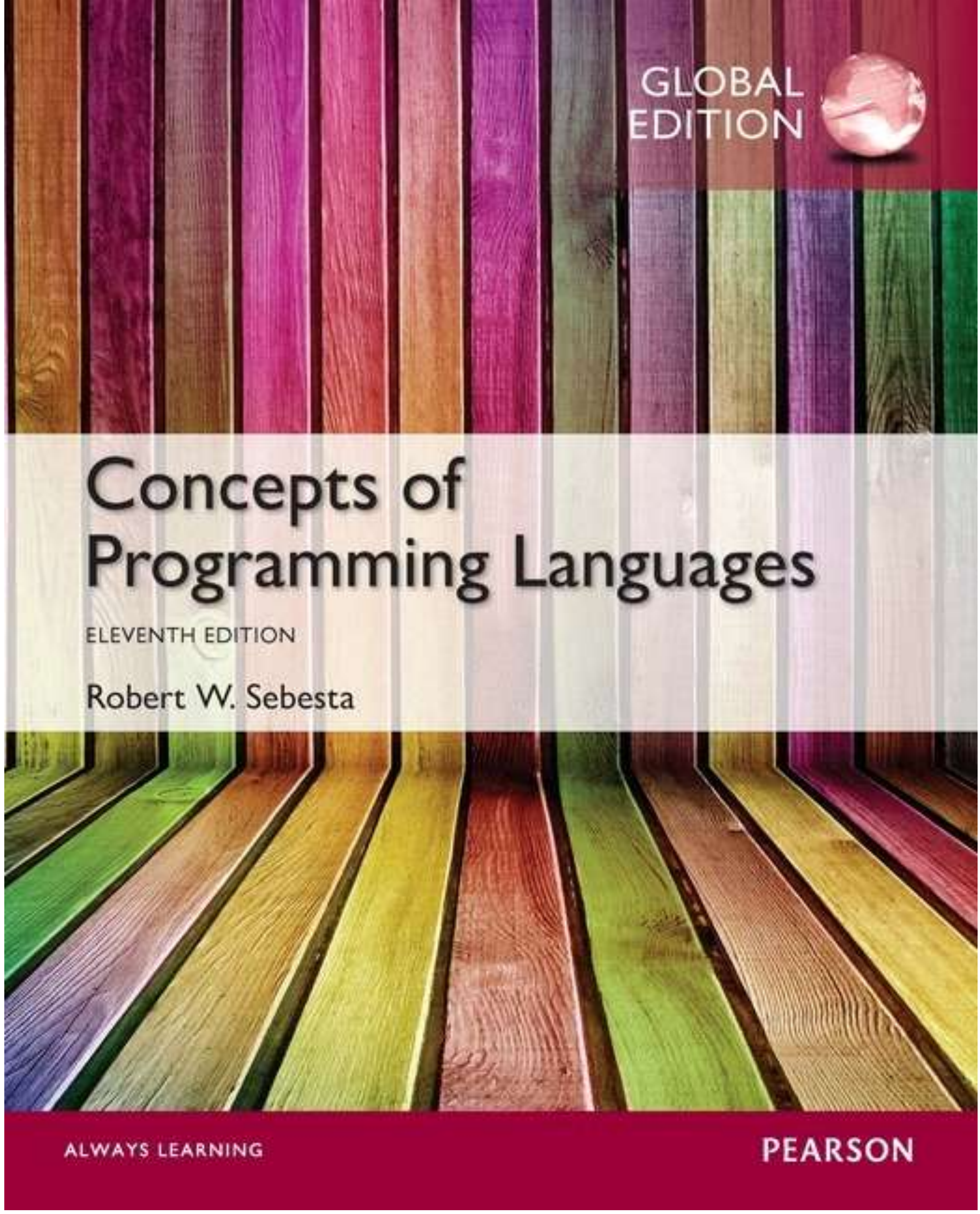
- A *named constant* is a variable that is bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - C++ and Java: expressions of any kind, dynamically bound
  - C# has two kinds, **`readonly`** and **`const`**
    - the values of **`const`** named constants are bound at compile time
    - The values of **`readonly`** named constants are dynamically bound

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors

Chapter 6

Data Types



Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- Enumeration Types
- Array Types
- Associative Arrays
- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types
- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

Introduction

- A *data type* defines a collection of data objects and a set of predefined operations on those objects
- A *descriptor* is the collection of the attributes of a variable
- An *object* represents an instance of a user-defined (abstract data) type
- One design issue for all data types: What operations are defined and how are they specified?

Primitive Data Types

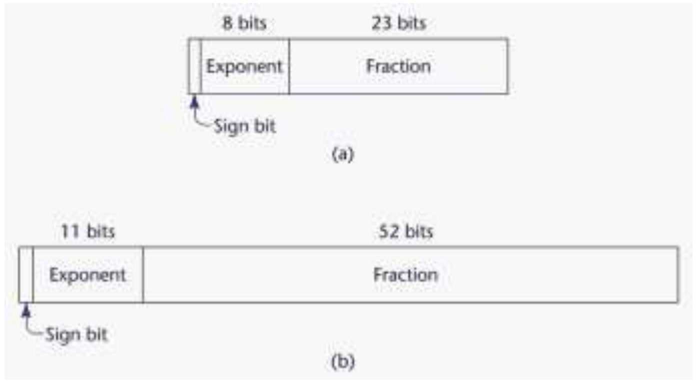
- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

Primitive Data Types: Integer

- Almost always an exact reflection of the hardware so the mapping is trivial
- There may be as many as eight different integer types in a language
- Java’s signed integer sizes: **`byte`**, **`short`**, **`int`**, **`long`**

Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., **`float`** and **`double`**; sometimes more
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Primitive Data Types: Complex

- Some languages support a complex type, e.g., C99, Fortran, and Python
- Each value consists of two floats, the real part and the imaginary part
- Literal form (in Python):

  `(7 + 3j)`, where `7` is the real part and `3` is the imaginary part

Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits, in coded form (BCD)
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements, one for “true” and one for “false”
- Could be implemented as bits, but often as bytes
  - Advantage: readability

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode (UCS-2)
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode
- 32-bit Unicode (UCS-4)
  - Supported by Fortran, starting with 2003

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?

Character String Types Operations

- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Character String Type in Certain Languages

- C and C++
  - Not primitive
  - Use **`char`** arrays and a library of functions that provide operations
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Fortran and Python
  - Primitive type with assignment and several operations
- Java
  - Primitive via the `String` class
- Perl, JavaScript, Ruby, and PHP
  - Provide built-in pattern matching, using regular expressions

Character String Length Options

- Static: COBOL, Java's `String` class
- *Limited Dynamic Length*: C and C++
  - In these languages, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript

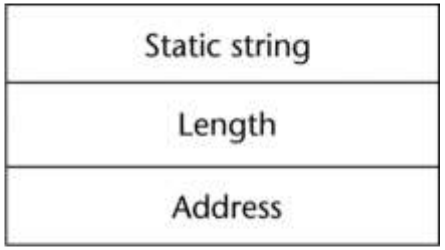
Character String Type Evaluation

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide--why not have them?
- Dynamic length is nice, but is it worth the expense?

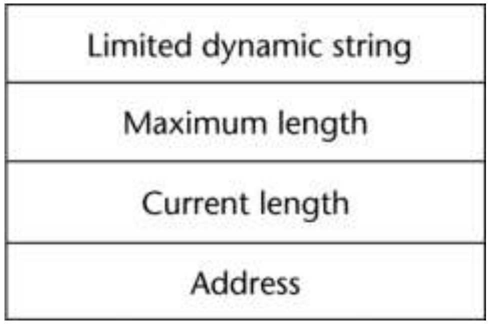
Character String Implementation

- Static length: compile-time descriptor
- Limited dynamic length: may need a run-time descriptor for length (but not in C and C++)
- Dynamic length: need run-time descriptor; allocation/deallocation is the biggest implementation problem

Compile- and Run-Time Descriptors



Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings

User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of primitive ordinal types in Java
  - `integer`
  - `char`
  - `boolean`

Enumeration Types

- All possible values, which are named constants, are provided in the definition
- C# example

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- Design issues
  - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
  - Are enumeration values coerced to integer?
  - Any other type coerced to an enumeration type?

Evaluation of Enumerated Type

- Aid to readability, e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
  - operations (don't allow colors to be added)
  - No enumeration variable can be assigned a value outside its defined range
  - C# and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Array Types

- An array is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements

  array_name (index_value_list) $\rightarrow$ an element
- Index Syntax
  - Fortran and Ada use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
  - Most other languages use brackets

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Java: integer types only
- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking

Subscript Binding and Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
  - Advantage: space efficiency

Subscript Binding and Array Categories (continued)

- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

Subscript Binding and Array Categories (continued)

- Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby

Array Initialization

- C-based languages
  - `int list [] = {1, 3, 5, 7}`
  - `char *names [] = {"Mike", "Fred", "Mary Lou"};`
- Python
  - List comprehensions

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
    puts [0, 9, 36, 81] in list
```

Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)
- Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
- Ruby also provides array catenation

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- F# and C# support rectangular arrays and jagged arrays

Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations

Slice Examples

- Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`vector (3:6)` is a three-element array

`mat[0][0:2]` is the first and second element of the first row of `mat`

- Ruby supports slices with the `slice` method

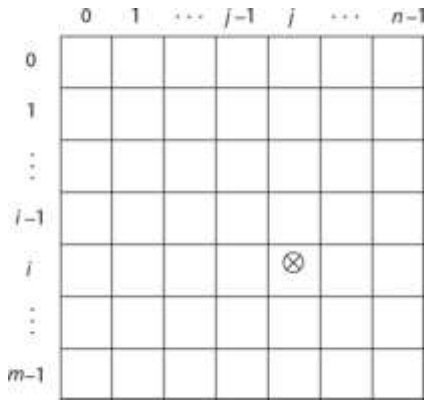
`list.slice(2, 2)` returns the third and fourth elements of `list`

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:

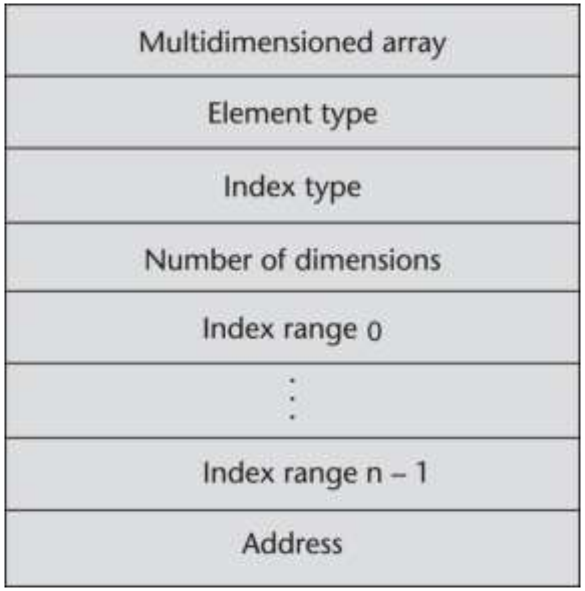
  address(list[k]) = address (list[lower_bound])

  + ((k-lower_bound) * element_size)



Accessing Multi-dimensioned Arrays

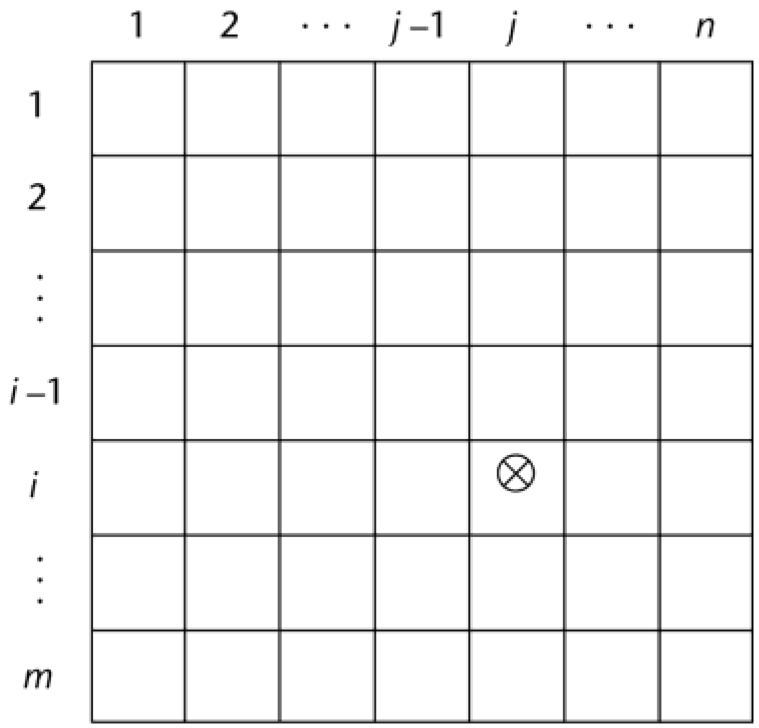
- Two common ways:
  - Row major order (by rows) – used in most languages
  - Column major order (by columns) – used in Fortran
  - A compile-time descriptor for a multidimensional array



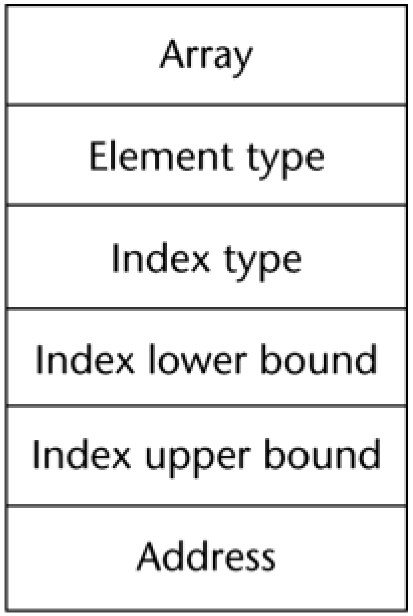
Multidimensioned array
Element type
Index type
Number of dimensions
Index range 0
⋮
Index range n – 1
Address

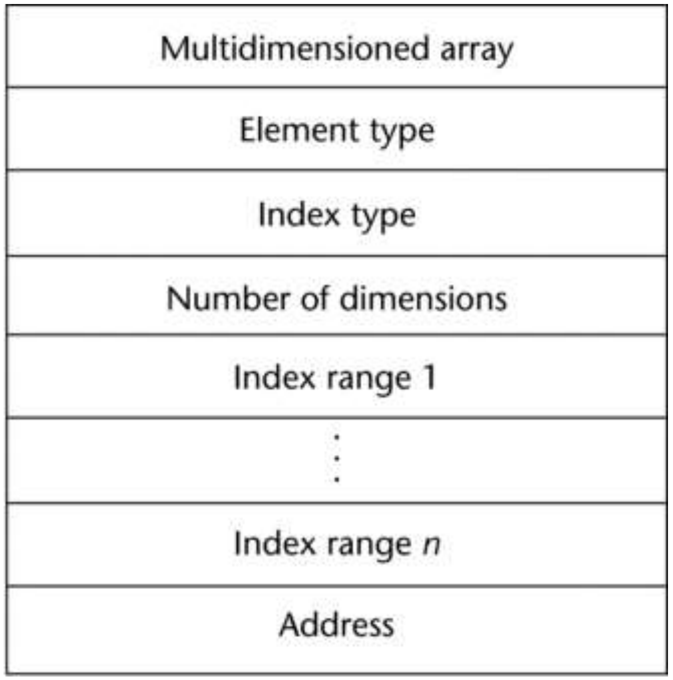
Locating an Element in a Multi-dimensioned Array

- General format
  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I - row_lb) * n) + (j - col_lb)) * element_size



Compile-Time Descriptors





Single-dimensioned array

Multidimensional array

Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
  - User-defined keys must be stored
- Design issues:
  - What is the form of references to elements?
  - Is the size static or dynamic?
- Built-in type in Perl, Python, Ruby, and Lua
  - In Lua, they are supported by tables

Associative Arrays in Perl

- Names begin with `%`; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" =>
   65, …);
```

- Subscripting is done using braces and keys

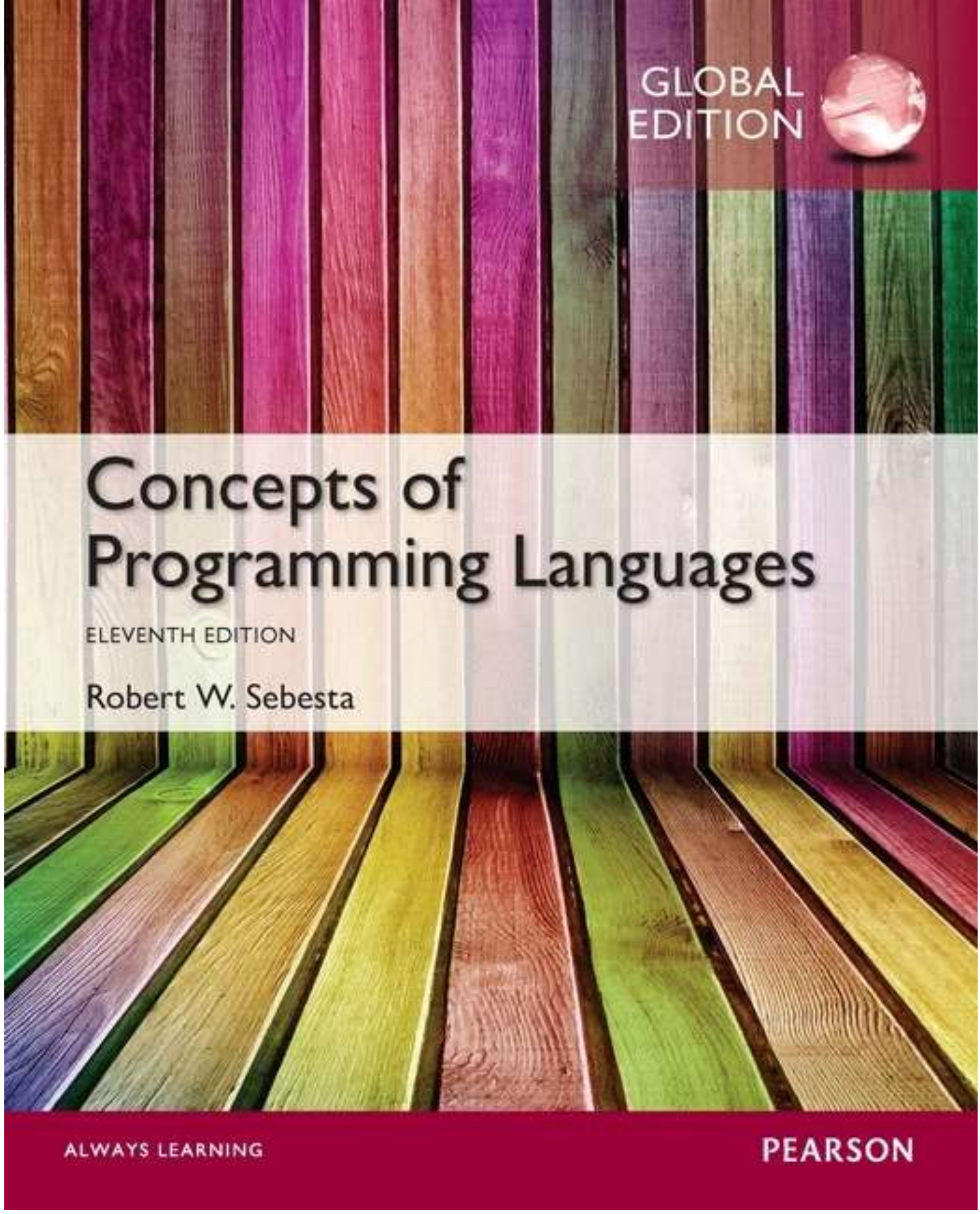
```
$hi_temps{"Wed"} = 83;
```

  – Elements can be removed with **`delete`**

```
   delete $hi_temps{"Tue"};
```

Chapter 6.2

Data Types



Chapter 6.2 Topics

- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types
- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

Record Types

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

Definition of Records in COBOL

- COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

References to Records

- Record field references

  1. COBOL

  field_name `OF` record_name_1 `OF` ... `OF` record_name_n

  2. Others (dot notation)

  record_name_1.record_name_2. ... record_name_n.field_name

- Fully qualified references must include all record names

- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL

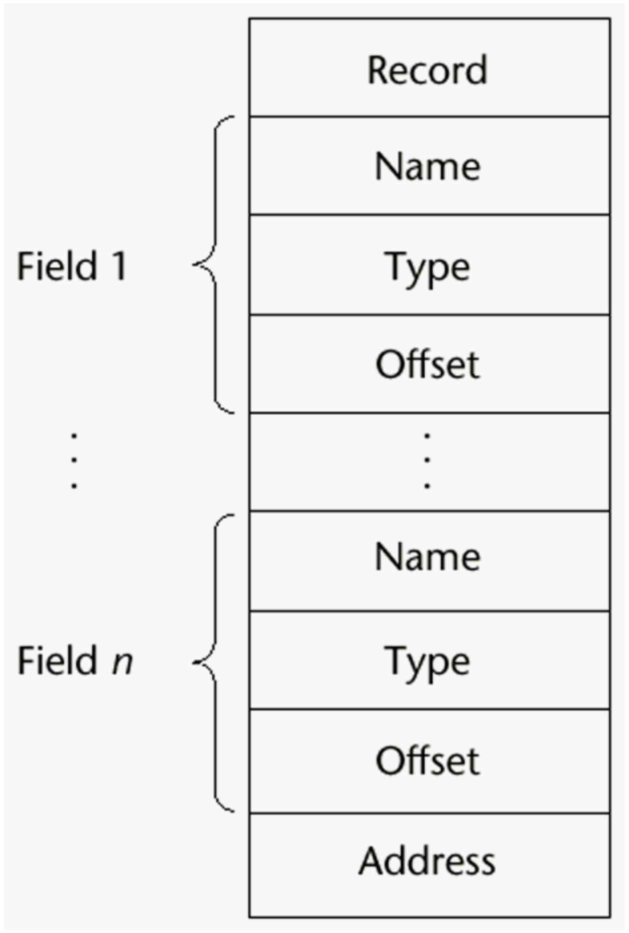
  `FIRST`, `FIRST OF EMP-NAME`, and `FIRST` of EMP-REC are elliptical references to the employee's first name

Evaluation and Comparison to Arrays

- Records are used when collection of data values is heterogeneous
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Tuple Types

- A tuple is a data type that is similar to a record, except that the elements are not named
- Used in Python, ML, and F# to allow functions to return multiple values
  - Python
    - Closely related to its lists, but immutable
    - Create with a tuple literal

      `myTuple = (3, 5.8, 'apple')`

      Referenced with subscripts (begin at `1`)

    Catenation with `+` and deleted with **`del`**

Tuple Types (continued)

- ML

```
val myTuple = (3, 5.8, 'apple');
```

  - Access as follows:

`#1(myTuple)` is the first element

  - A new tuple type can be defined

```
type intReal = int * real;
```

- F#

```
let tup = (3, 5, 7)
let a, b, c = tup
```

This assigns a tuple to a tuple pattern `(a, b, c)`

List Types

- Lists in Lisp and Scheme are delimited by parentheses and use no commas

  `(A B C D)` and `(A (B C) D)`

- Data and code have the same form

  As data, `(A B C)` is literally what it is

  As code, `(A B C)` is the function `A` applied to the parameters `B` and `C`

- The interpreter needs to know which a list is, so if it is data, we quote it with an apostrophe

  `'(A B C)` is data

List Types (continued)

- List Operations in Scheme
  - `CAR` returns the first element of its list parameter

    `(CAR '(A B C))` returns `A`
  - `CDR` returns the remainder of its list parameter after the first element has been removed

    `(CDR '(A B C))` returns `(B C)`
  - `CONS` puts its first parameter into its second parameter, a list, to make a new list

    `(CONS 'A (B C))` returns `(A B C)`
  - `LIST` returns a new list of its parameters

    `(LIST 'A 'B '(C D))` returns `(A B (C D))`

List Types (continued)

- List Operations in ML
  - Lists are written in brackets and the elements are separated by commas
  - List elements must be of the same type
  - The Scheme `CONS` function is a binary operator in ML, `::`

    `3 :: [5, 7, 9]` evaluates to `[3, 5, 7, 9]`
  - The Scheme `CAR` and `CDR` functions are named `hd` and `tl`, respectively

List Types (continued)

- F# Lists
  - Like those of ML, except elements are separated by semicolons and `hd` and `tl` are methods of the `List` class
- Python Lists
  - The list data type also serves as Python's arrays
  - Unlike Scheme, Common Lisp, ML, and F#, Python's lists are mutable
  - Elements can be of any type
  - Create a list with an assignment

```
myList = [3, 5.8, "grape"]
```

List Types (continued)

- Python Lists (continued)
  - List elements are referenced with subscripting, with indices beginning at zero

    `x = myList[1]` Sets `x` to `5.8`
  - List elements can be deleted with `del`

    `del myList[1]`
  - List Comprehensions – derived from set notation

    `[x * x for x in range(6) if x % 3 == 0]`

    `range(12)` creates `[0, 1, 2, 3, 4, 5, 6]`

    Constructed list: `[0, 9, 36]`

List Types (continued)

- Haskell’s List Comprehensions
  - The original

```
[n * n | n <- [1..10]]
```

- F#’s List Comprehensions

```
let myArray = [|for i in 1 .. 5 -> [i * i) |]
```

- Both C# and Java supports lists through their generic heap-dynamic collection classes, `List` and `ArrayList`, respectively

Unions Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
- Design issue
  - Should type checking be required?

Discriminated vs. Free Unions

- C and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*
- Type checking of unions require that each union include a type indicator called a *discriminant*
  - Supported by ML, Haskell, and F#

Unions in F#

- Defined with a type statement using OR

```
type intReal =
     | IntValue of int
     | RealValue of float;;
```

`intReal` is the new type

`IntValue` and `RealValue` are constructors

To create a value of type `intReal`:

```
let ir1 = IntValue 17;;
let ir2 = RealValue 3.4;;
```

Unions in F# (continued)

- Accessing the value of a union is done with pattern matching

```
match pattern with
    | expression_list_1 -> expression_1
    | …
    | expression_list_n -> expression_n
```

  - Pattern can be any data type
  - The expression list can have wild cards (_)

Unions in F# (continued)

Example:

```
let a = 7;;
let b = "grape";;
let x = match (a, b) with
      | 4, "apple" -> apple
      | _, "grape" -> grape
      | _ -> fruit;;
```

Unions in F# (continued)

To display the type of the `intReal` union:

```
let printType value =
    match value with
          | IntVale value -> printfn "int"
          | RealValue value -> printfn "float";;
```

If `ir1` and `ir2` are defined as previously,

`printType ir1` **returns** `int`

`printType ir2` **returns** `float`

Evaluation of Unions

- Free unions are unsafe
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language

Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)

Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

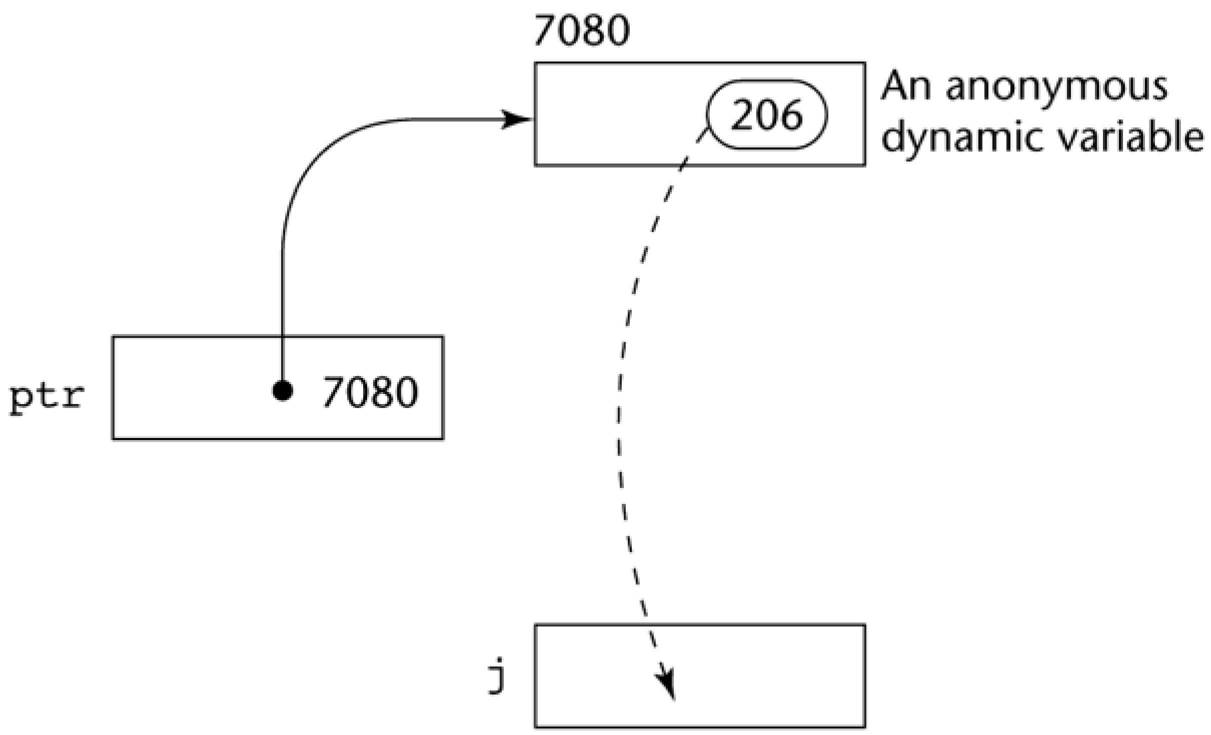
Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable’s value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer’s value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *

    ```
    j = *ptr
    ```

    sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - Pointer `p1` is set to point to a newly created heap-dynamic variable
    - Pointer `p1` is later set to point to another newly created heap-dynamic variable
    - The process of losing heap-dynamic variables is called *memory leakage*

Pointers in C and C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when or where it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (`void *`)

  `void *` can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++

Evaluation of Pointers

- Dangling pointers and dangling objects are problems as is heap management
- Pointers are like `goto`'s--they widen the range of cells that can be accessed by a variable
- Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Representations of Pointers

- Large computers use single values
- Intel microprocessors use segment and offset

Dangling Pointer Problem

- *Tombstone*: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - Costly in time and space

. *Locks-and-keys*: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

Heap Management

- A very complex run-time process
- Single-size cells vs. variable-size cells
- Two approaches to reclaim garbage
  - Reference counters (*eager approach*): reclamation is gradual
  - Mark-sweep (*lazy approach*): reclamation occurs when the list of variable space becomes empty

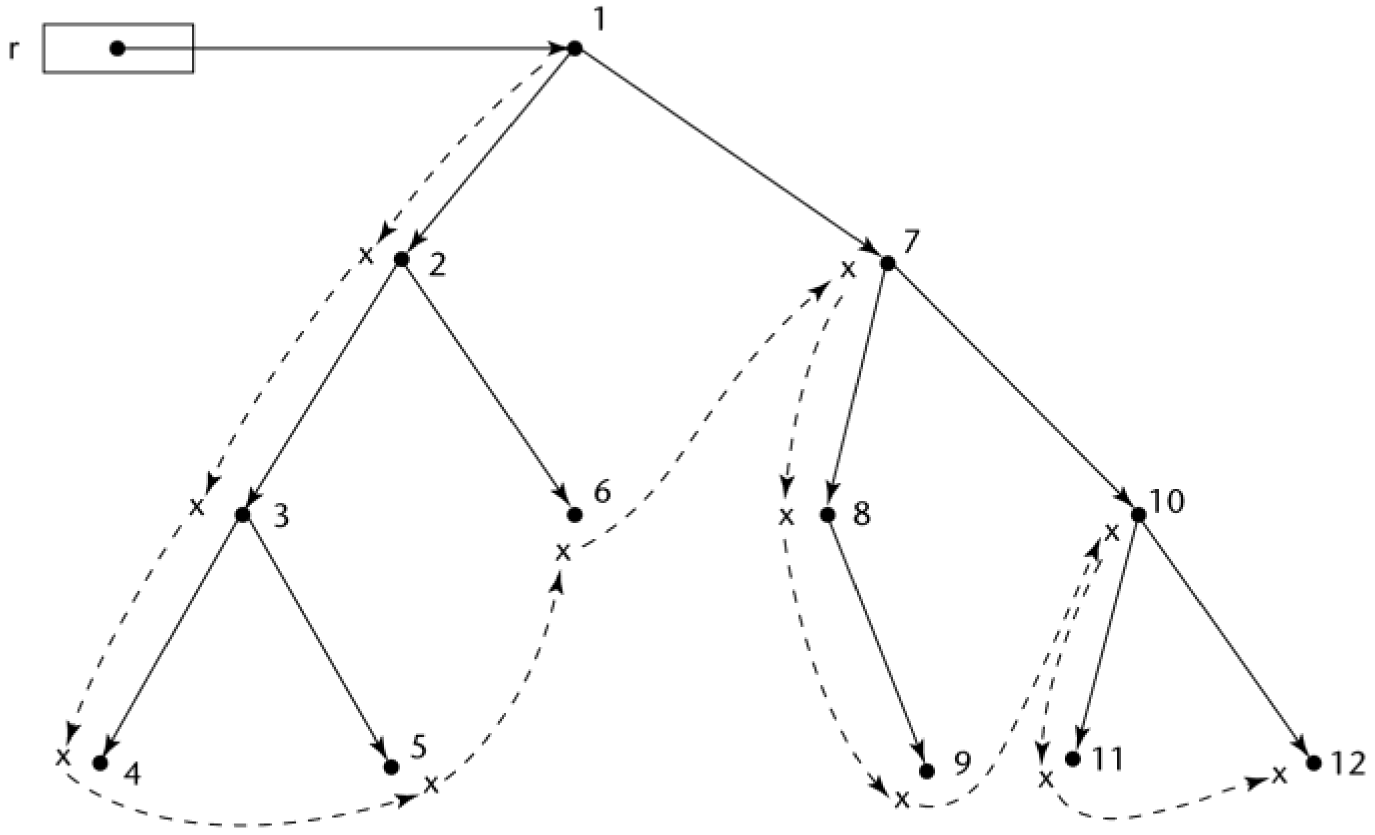
Reference Counter

- Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - *Disadvantages*: space required, execution time required, complications for cells connected circularly
  - *Advantage*: it is intrinsically incremental, so significant delays in the application execution are avoided

Mark-Sweep

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep

Marking Algorithm



Dashed lines show the order of node_marking

Variable-Size Cells

- All the difficulties of single-size cells plus more
- Required by most programming languages
- If mark-sweep is used, additional problems occur
  - The initial setting of the indicators of all cells in the heap is difficult
  - The marking process in nontrivial
  - Maintaining the list of available space is another source of overhead

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

Language examples:

- C and C++ are not: parameter type checking can be avoided; unions are not type checked
- Java and C# are, almost (because of explicit type casting)
- ML and F# are

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus ML and F#)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Name Type Equivalence

- *Name type equivalence* means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not equivalent with integer types
  - Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

- *Structure type equivalence* means that two variables have equivalent types if their types have identical structures
- More flexible, but harder to implement

Type Equivalence (continued)

- Consider the problem of two structured types:
  - Are two record types equivalent if they are structurally the same but use different field names?
  - Are two array types equivalent if they are the same except that the subscripts are different? (e.g. `[1..10]` and `[0..9]`)
  - Are two enumeration types equivalent if their components are spelled differently?
  - With structural type equivalence, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Theory and Data Types

- Type theory is a broad area of study in mathematics, logic, computer science, and philosophy
- Two branches of type theory in computer science:
  - Practical – data types in commercial languages
  - Abstract – typed lambda calculus
- A type system is a set of types and the rules that govern their use in programs

Theory and Data Types (continued)

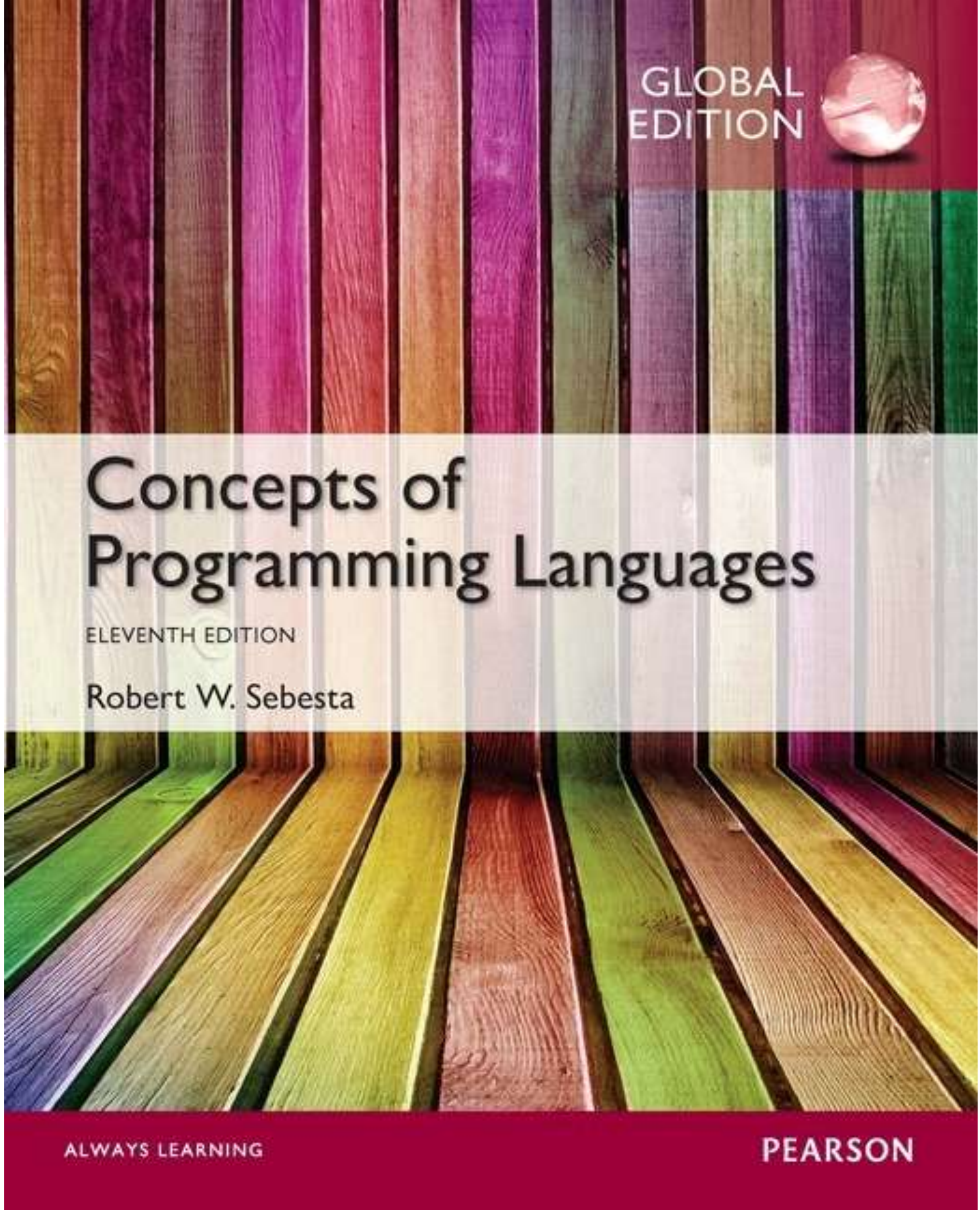
- Formal model of a type system is a set of types and a collection of functions that define the type rules
  - Either an attribute grammar or a type map could be used for the functions
  - Finite mappings – model arrays and functions
  - Cartesian products – model tuples and records
  - Set unions – model union types
  - Subsets – model subtypes

Summary

- The data types of a language are a large part of what determines that language’s style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management

Chapter 7

Expressions and Assignment Statements



Chapter 7 Topics

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Introduction

- Expressions are the fundamental means of specifying computations in a programming language
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation
- Essence of imperative languages is dominant role of assignment statements

Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of operators, operands, parentheses, and function calls

Arithmetic Expressions: Design Issues

- Design issues for arithmetic expressions
  - Operator precedence rules?
  - Operator associativity rules?
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?

Arithmetic Expressions: Operators

- A unary operator has one operand
- A binary operator has two operands
- A ternary operator has three operands

Arithmetic Expressions: Operator Precedence Rules

- The *operator precedence rules* for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated
- Typical precedence levels
  - parentheses
  - unary operators
  - ** (if the language supports it)
  - *, /
  - +, -

Arithmetic Expressions: Operator Associativity Rule

- The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated
- Typical associativity rules
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., in FORTRAN)
- APL is different; all operators have equal precedence and all operators associate right to left
- Precedence and associativity rules can be overriden with parentheses

Expressions in Ruby and Scheme

- Ruby
  - All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as methods
  - One result of this is that these operators can all be overriden by application programs
- Scheme (and Common Lisp)
  - All arithmetic and logic operations are by explicitly called subprograms
  - `a + b * c` is coded as `(+ a (* b c))`

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:

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

  - Evaluates as if written as follows:

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

Arithmetic Expressions: Operand Evaluation Order

- *Operand evaluation order*
  1. Variables: fetch the value from memory
  2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
  3. Parenthesized expressions: evaluate all operands and operators first
  4. The most interesting case is when an operand is a function call

Arithmetic Expressions: Potentials for Side Effects

- *Functional side effects:* when a function changes a two-way parameter or a non-local variable
- Problem with functional side effects:
  - When a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(&a);
```

Functional Side Effects

- Two possible solutions to the problem
  1. Write the language definition to disallow functional side effects
     - No two-way parameters in functions
     - No non-local references in functions
     - **Advantage:** it works!
     - **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

Referential Transparency

- 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

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

If `fun` has no side effects, `result1 = result2`

Otherwise, not, and referential transparency is violated

Referential Transparency (continued)

- Advantage of referential transparency
  - Semantics of a program is much easier to understand if it has referential transparency
- Because they do not have variables, programs in pure functional languages are referentially transparent
  - Functions cannot have state, which would be stored in local variables
  - If a function uses an outside value, it must be a constant (there are no variables). So, the value of a function depends only on its parameters

Overloaded Operators

- Use of an operator for more than one purpose is called *operator overloading*
- Some are common (e.g., + for **`int`** and **`float`**)
- Some are potential trouble (e.g., * in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability

Overloaded Operators (continued)

- C++, C#, and F# allow user-defined overloaded operators
  - When sensibly used, such operators can be an aid to readability (avoid method calls, expressions appear natural)
  - Potential problems:
    - Users can define nonsense operations
    - Readability may suffer, even when the operators make sense

Type Conversions

- A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type e.g., `float` to `int`
- A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type
  e.g., `int` to `float`

Type Conversions: Mixed Mode

- A *mixed-mode expression* is one that has operands of different types
- A *coercion* is an implicit type conversion
- Disadvantage of coercions:
  - They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- In ML and F#, there are no coercions in expressions

Explicit Type Conversions

- Called *casting* in C-based languages
- Examples
  - C: `(int)angle`
  - F#: `float(sum)`

Note that F#'s syntax is similar to that of function calls

Errors in Expressions

- Causes
  - Inherent limitations of arithmetic
    e.g., division by zero
  - Limitations of computer arithmetic
    e.g. overflow
- Often ignored by the run-time system

Relational and Boolean Expressions

- Relational Expressions
  - Use relational operators and operands of various types
  - Evaluate to some Boolean representation
  - Operator symbols used vary somewhat among languages (`!=`, `/=`, `~=`, `.NE.`, `<>`, `#`)
- JavaScript and PHP have two additional relational operator, `===` and `!==`
  - Similar to their cousins, `==` and `!=`, except that they do not coerce their operands
  - Ruby uses `==` for equality relation operator that uses coercions and `eql?` for those that do not

Relational and Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Example operators
- C89 has no Boolean type--it uses `int` type with 0 for false and nonzero for true
- One odd characteristic of C's expressions: `a < b < c` is a legal expression, but the result is not what you might expect:
  - Left operator is evaluated, producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., `c`)

Short Circuit Evaluation

- An expression in which the result is determined without evaluating all of the operands and/or operators
- Example: `(13 * a) * (b / 13 - 1)`

  If `a` is zero, there is no need to evaluate `(b /13 - 1)`
- Problem with non-short-circuit evaluation

```
index = 0;
while (index <= length) && (LIST[index] != value)
        index++;
```

  - When `index=length`, `LIST[index]` will cause an indexing problem (assuming `LIST` is `length - 1` long)

Short Circuit Evaluation (continued)

- C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (`&&` and `||`), but also provide bitwise Boolean operators that are not short circuit (`&` and `|`)
- All logic operators in Ruby, Perl, ML, F#, and Python are short-circuit evaluated
- Short-circuit evaluation exposes the potential problem of side effects in expressions
  e.g. `(a > b) || (b++ / 3)`

Assignment Statements

- The general syntax

  `<target_var> <assign_operator> <expression>`
- The assignment operator

  `=` Fortran, BASIC, the C-based languages

  `:=` Ada
- `=` can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use `==` as the relational operator)

Assignment Statements: Conditional Targets

- Conditional targets (Perl)

```
($flag ? $total : $subtotal) = 0
```

Which is equivalent to

```
if ($flag){
  $total = 0
} else {
  $subtotal = 0
}
```

Assignment Statements: Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by C and the C-based languaes
  - Example

```
a = a + b
```

can be written as

```
a += b
```

Assignment Statements: Unary Assignment Operators

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment
- Examples

  `sum = ++count` (`count` incremented, then assigned to `sum`)

  `sum = count++` (`count` assigned to `sum`, then incremented

  `count++` (`count` incremented)

  `-count++` (`count` incremented then negated)

Assignment as an Expression

- In the C-based languages, Perl, and JavaScript, the assignment statement produces a result and can be used as an operand

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

`ch = getchar()` is carried out; the result (assigned to `ch`) is used as a conditional value for the `while` statement

- Disadvantage: another kind of expression side effect

Multiple Assignments

- Perl, Ruby, and Lua allow multiple-target multiple-source assignments

```
($first, $second, $third) = (20, 30, 40);
```

Also, the following is legal and performs an interchange:

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

Assignment in Functional Languages

- Identifiers in functional languages are only names of values
- ML
  - Names are bound to values with **`val`**

    ```
    val fruit = apples + oranges;
    ```

  - If another val for fruit follows, it is a new and different name
- F#
  - F#'s **`let`** is like ML's **`val`**, except **`let`** also creates a new scope

Mixed-Mode Assignment

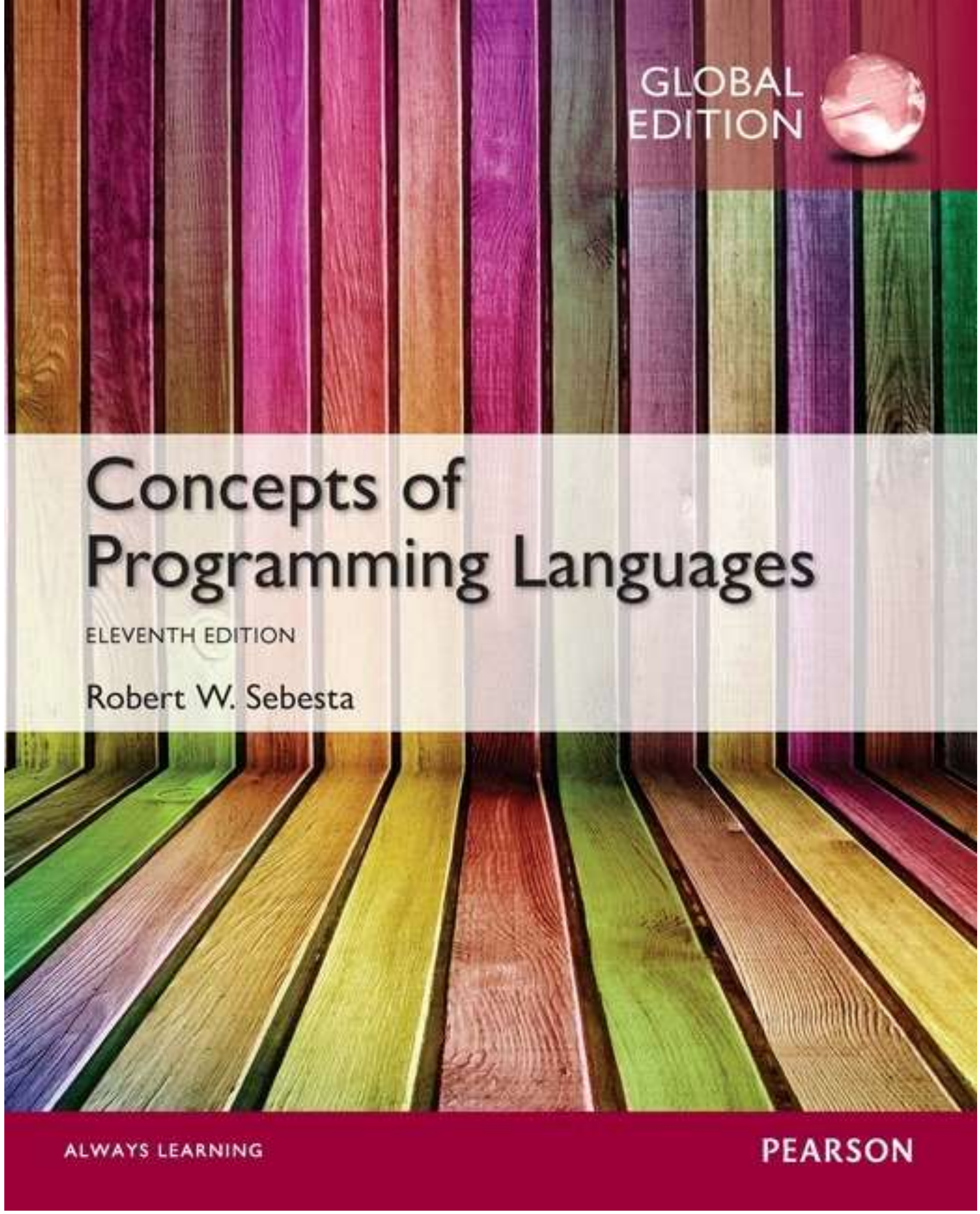
- Assignment statements can also be mixed-mode
- In Fortran, C, Perl, and C++, any numeric type value can be assigned to any numeric type variable
- In Java and C#, only widening assignment coercions are done
- In Ada, there is no assignment coercion

Summary

- Expressions
- Operator precedence and associativity
- Operator overloading
- Mixed-type expressions
- Various forms of assignment

Chapter 8

Statement-Level Control Structures



Chapter 8 Topics

- Introduction
- Selection Statements
- Iterative Statements
- Unconditional Branching
- Guarded Commands
- Conclusions

Control Statements: Evolution

- FORTRAN I control statements were based directly on IBM 704 hardware
- Much research and argument in the 1960s about the issue
  - One important result: It was proven that all algorithms represented by flowcharts can be coded with only two-way selection and pretest logical loops

Control Structure

- A *control structure* is a control statement and the statements whose execution it controls
- Design question
  - Should a control structure have multiple entries?

Selection Statements

- A *selection statement* provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Two-Way Selection Statements

- General form:

```
if control_expression
    then clause
    else clause
```

- Design Issues:
  - What is the form and type of the control expression?
  - How are the **`then`** and **`else`** clauses specified?
  - How should the meaning of nested selectors be specified?

The Control Expression

- If the then reserved word or some other syntactic marker is not used to introduce the then clause, the control expression is placed in parentheses
- In C89, C99, Python, and C++, the control expression can be arithmetic
- In most other languages, the control expression must be Boolean

Clause Form

- In many contemporary languages, the then and else clauses can be single statements or compound statements
- In Perl, all clauses must be delimited by braces (they must be compound)
- In Python and Ruby, clauses are statement sequences
- Python uses indentation to define clauses

```
if x > y :
  x = y
  print " x was greater than y"
```

Nesting Selectors

- Java example

```
    if (sum == 0)
      if (count == 0)
           result = 0;
  else result = 1;
```

- Which `if` gets the `else`?
- Java's static semantics rule: `else` matches with the nearest previous `if`

Nesting Selectors (continued)

- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
 if (count == 0)
     result = 0;
}
else result = 1;
```

- The above solution is used in C, C++, and C#

Nesting Selectors (continued)

- Statement sequences as clauses: Ruby

```
if sum == 0 then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Nesting Selectors (continued)

- Python

```
if sum == 0 :
  if count == 0 :
    result = 0
  else :
    result = 1
```

Selector Expressions

- In ML, F#, and Lisp, the selector is an expression; in F#:

```
let y =
  if x > 0 then x
  else 2 * x
```

  - If the `if` expression returns a value, there must be an else clause (the expression could produce a unit type, which has no value). The types of the values returned by then and else clauses must be the same.

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
- Design Issues:
  1. What is the form and type of the control expression?
  2. How are the selectable segments specified?
  3. Is execution flow through the structure restricted to include just a single selectable segment?
  4. How are case values specified?
  5. What is done about unrepresented expression values?

Multiple-Way Selection: Examples

- C, C++, Java, and JavaScript

```
switch (expression) {
   case const_expr_1: stmt_1;
   …
   case const_expr_n: stmt_n;
   [default: stmt_n+1]
}
```

Multiple-Way Selection: Examples

- Design choices for C's **`switch`** statement
  1. Control expression can be only an integer type
  2. Selectable segments can be statement sequences, blocks, or compound statements
  3. Any number of segments can be executed in one execution of the construct (*there is no implicit branch at the end of selectable segments*)
  4. **`default`** clause is for unrepresented values (if there is no **`default`**, the whole statement does nothing)

Multiple-Way Selection: Examples

- C#
  - Differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (`goto` or `break`)
  - Also, in C# the control expression and the case constants can be strings

Multiple-Way Selection: Examples

- Ruby has two forms of case statements–we'll cover only one

```
leap = case
    when year % 400 == 0 then true
    when year % 100 == 0 then false
    else year % 4 == 0
    end
```

Implementing Multiple Selectors

- Approaches:
  - Multiple conditional branches
  - Store case values in a table and use a linear search of the table
  - When there are more than ten cases, a hash table of case values can be used
  - If the number of cases is small and more than half of the whole range of case values are represented, an array whose indices are the case values and whose values are the case labels can be used

Multiple-Way Selection Using `if`

- Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses, for example in Python:

```
if count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

Multiple-Way Selection Using `if`

- The Python example can be written as a Ruby `case`

```
case
  when count < 10 then bag1 = true
  when count < 100 then bag2 = true
  when count < 1000 then bag3 = true
end
```

Scheme's Multiple Selector

- General form of a call to `COND`:

```
(COND
  (predicate₁ expression₁)
   …
  (predicateₙ expressionₙ)
  [(ELSE expressionₙ₊₁)]
)
```

  - The `ELSE` clause is optional; `ELSE` is a synonym for true
  - Each predicate-expression pair is a parameter
  - Semantics: The value of the evaluation of `COND` is the value of the expression associated with the first predicate expression that is true

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- General design issues for iteration control statements:
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

Counter-Controlled Loops

- A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values
- Design Issues:
  1. What are the type and scope of the loop variable?
  2. Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
  3. Should the loop parameters be evaluated only once, or once for every iteration?

Counter-Controlled Loops: Examples

- C-based languages

  `for ([expr_1] ; [expr_2] ; [expr_3]) statement`

  – The expressions can be whole statements, or even statement sequences, with the statements separated by commas
  - The value of a multiple-statement expression is the value of the last statement in the expression
  - If the second expression is absent, it is an infinite loop
- Design choices:
  - There is no explicit loop variable
  - Everything can be changed in the loop
  - The first expression is evaluated once, but the other two are evaluated with each iteration
  - It is legal to branch into the body of a for loop in C

Counter-Controlled Loops: Examples

- C++ differs from C in two ways:
  1. The control expression can also be Boolean
  2. The initial expression can include variable definitions (scope is from the definition to the end of the loop body)
- Java and C#
  - Differs from C++ in that the control expression must be Boolean

Counter-Controlled Loops: Examples

- Python

  for loop_variable **in** object:

   - loop body

  [**else**:

   - else clause]

  - The object is often a range, which is either a list of values in brackets (`[2, 4, 6]`), or a call to the range function (**range**`(5)`, which returns `0, 1, 2, 3, 4`

  - The loop variable takes on the values specified in the given range, one for each iteration

  - The else clause, which is optional, is executed if the loop terminates normally

Counter-Controlled Loops: Examples

- F#
  - Because counters require variables, and functional languages do not have variables, counter-controlled loops must be simulated with recursive functions

```
let rec forLoop loopBody reps =
  if reps <= 0 then ()
  else
    loopBody()
    forLoop loopBody, (reps - 1)
```

  - This defines the recursive function `forLoop` with the parameters `loopBody` (a function that defines the loop's body) and the number of repetitions
  - `()` means do nothing and return nothing

Logically-Controlled Loops

- Repetition control is based on a Boolean expression
- Design issues:
  - Pretest or posttest?
  - Should the logically controlled loop be a special case of the counting loop statement or a separate statement?

Logically-Controlled Loops: Examples

- C and C++ have both pretest and posttest forms, in which the control expression can be arithmetic:

```
while (control_expr)        do
    loop body                   loop body
                            while (control_expr)
```

  - In both C and C++ it is legal to branch into the body of a logically-controlled loop

- Java is like C and C++, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no `goto`

Logically-Controlled Loops: Examples

- F#
  - As with counter-controlled loops, logically-controlled loops can be simulated with recursive functions

```
let rec whileLoop test body =
   if test() then
     body()
     whileLoop test body
   else ()
```

  - This defines the recursive function `whileLoop` with parameters `test` and `body`, both functions. `test` defines the control expression

User-Located Loop Control Mechanisms

- Sometimes it is convenient for the programmers to decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (e.g., **`break`**)
- Design issues for nested loops
  1. Should the conditional be part of the exit?
  2. Should control be transferable out of more than one loop?

User-Located Loop Control Mechanisms

- C , C++, Python, Ruby, and C# have unconditional unlabeled exits (**`break`**)
- Java and Perl have unconditional labeled exits (**`break`** in Java, **`last`** in Perl)
- C, C++, and Python have an unlabeled control statement, **`continue`**, that skips the remainder of the current iteration, but does not exit the loop
- Java and Perl have labeled versions of **`continue`**

Iteration Based on Data Structures

- The number of elements in a data structure controls loop iteration
- Control mechanism is a call to an *iterator* function that returns the next element in some chosen order, if there is one; else loop is terminate
- C's **`for`** can be used to build a user-defined iterator:

```
for (p=root; p==NULL; traverse(p)){
   ...
}
```

Iteration Based on Data Structures (continued)

- PHP
  - `current` points at one element of the array
  - `next` moves `current` to the next element
  - `reset` moves `current` to the first element
- Java 5.0 (uses **`for`**, although it is called foreach)

  For arrays and any other class that implements the `Iterable` interface, e.g., `ArrayList`

```
for (String myElement : myList) { … }
```

Iteration Based on Data Structures (continued)

- C# and F# (and the other .NET languages) have generic library classes, like Java 5.0 (for arrays, lists, stacks, and queues). Can iterate over these with the **`foreach`** statement. User-defined collections can implement the `IEnumerator` interface and also use **`foreach`**.

```
List<String> names = new List<String>();
 names.Add("Bob");
 names.Add("Carol");
 names.Add("Ted");
 foreach (Strings name in names)
     Console.WriteLine ("Name: {0}", name);
```

Iteration Based on Data Structures (continued)

- Ruby *blocks* are sequences of code, delimited by either braces or **`do`** and **`end`**
  - Blocks can be used with methods to create iterators
  - Predefined iterator methods (`times`, `each`, `upto`):

```
3.times {puts "Hey!"}
list.each {|value| puts value}
```

(`list` is an array; `value` is a block parameter)

```
1.upto(5) {|x| print x, " "}
```

Iterators are implemented with blocks, which can also be defined by applications

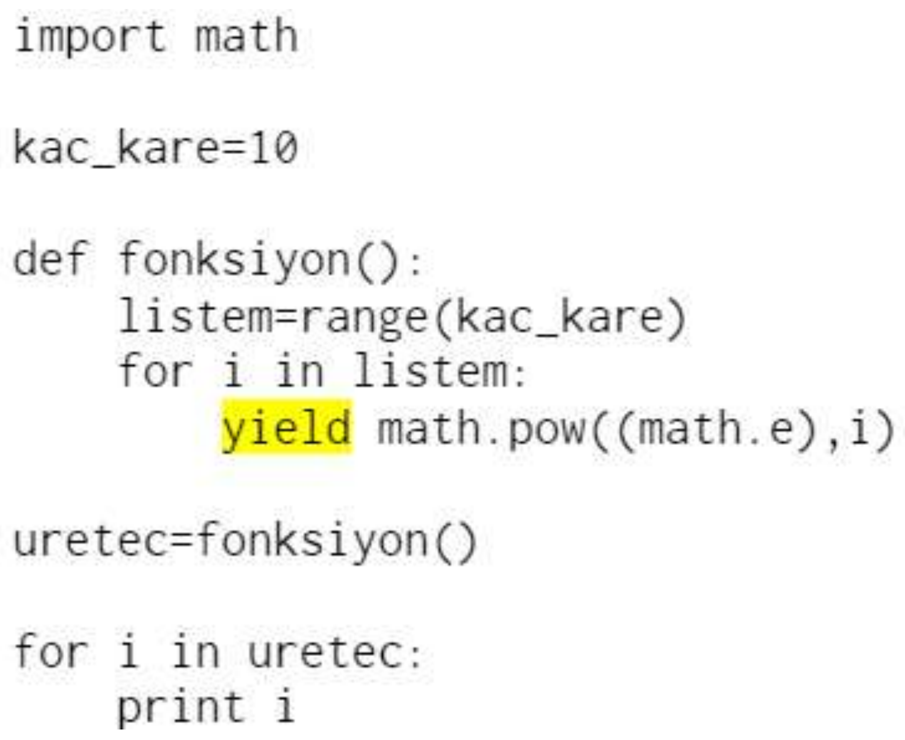
Iteration Based on Data Structures (continued)

- Ruby blocks are attached methods calls; they can have parameters (in vertical bars); they are executed when the method executes a **`yield`** statement

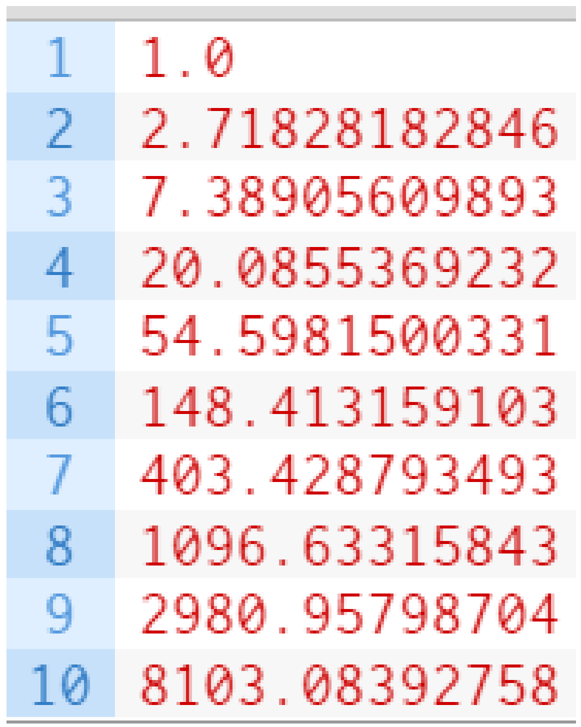
```
def fibonacci(last)
  first, second = 1, 1
  while first <= last
    yield first
    first, second = second, first + second
  end
end
puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "}
puts
```

  – Ruby has a **`for`** statement, but Ruby converts them to `upto` method calls

Python yield example

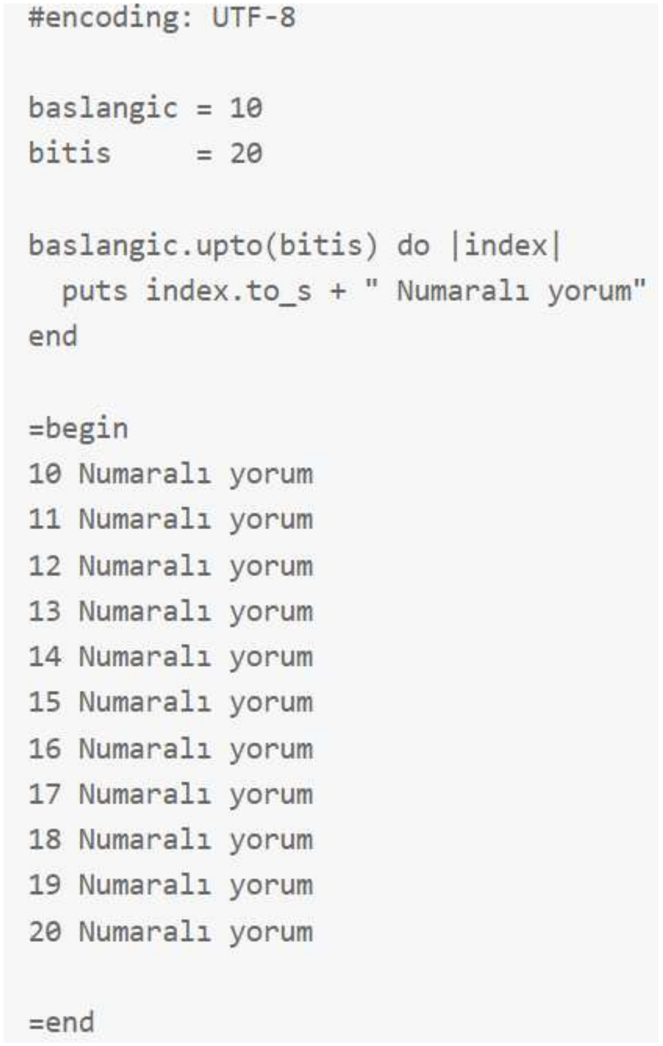


```
import math

kac_kare=10

def fonksiyon():
    listem=range(kac_kare)
    for i in listem:
        yield math.pow((math.e),i)

uretec=fonksiyon()

for i in uretec:
    print i
```



```
1.0
2.71828182846
7.38905609893
20.0855369232
54.5981500331
148.413159103
403.428793493
1096.63315843
2980.95798704
8103.08392758
```

Ruby upto Example



```
#encoding: UTF-8

baslangic = 10
bitis     = 20

baslangic.upto(bitis) do |index|
  puts index.to_s + " Numaralı yorum"
end

=begin
10 Numaralı yorum
11 Numaralı yorum
12 Numaralı yorum
13 Numaralı yorum
14 Numaralı yorum
15 Numaralı yorum
16 Numaralı yorum
17 Numaralı yorum
18 Numaralı yorum
19 Numaralı yorum
20 Numaralı yorum

=end
```

Unconditional Branching

- Transfers execution control to a specified place in the program
- Represented one of the most heated debates in 1960’s and 1970’s
- Major concern: Readability
- Some languages do not support **`goto`** statement (e.g., Java)
- C# offers **`goto`** statement (can be used in **`switch`** statements)
- Loop exit statements are restricted and somewhat camouflaged **`goto`**’s

Guarded Commands

- Designed by Dijkstra
- Purpose: to support a new programming methodology that supported verification (correctness) during development
- Basis for two linguistic mechanisms for concurrent programming (in CSP)
- Basic Idea: if the order of evaluation is not important, the program should not specify one

Selection Guarded Command

- Form

```
if <Boolean expr> -> <statement>
[] <Boolean expr> -> <statement>
  ...
[] <Boolean expr> -> <statement>
fi
```

- Semantics: when construct is reached,
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically
  - If none are true, it is a runtime error

Loop Guarded Command

- **Form**

```
do <Boolean> -> <statement>
 [] <Boolean> -> <statement>
  ...
 []  <Boolean> -> <statement>
od
```

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - If none are true, exit loop

Guarded Commands: Rationale

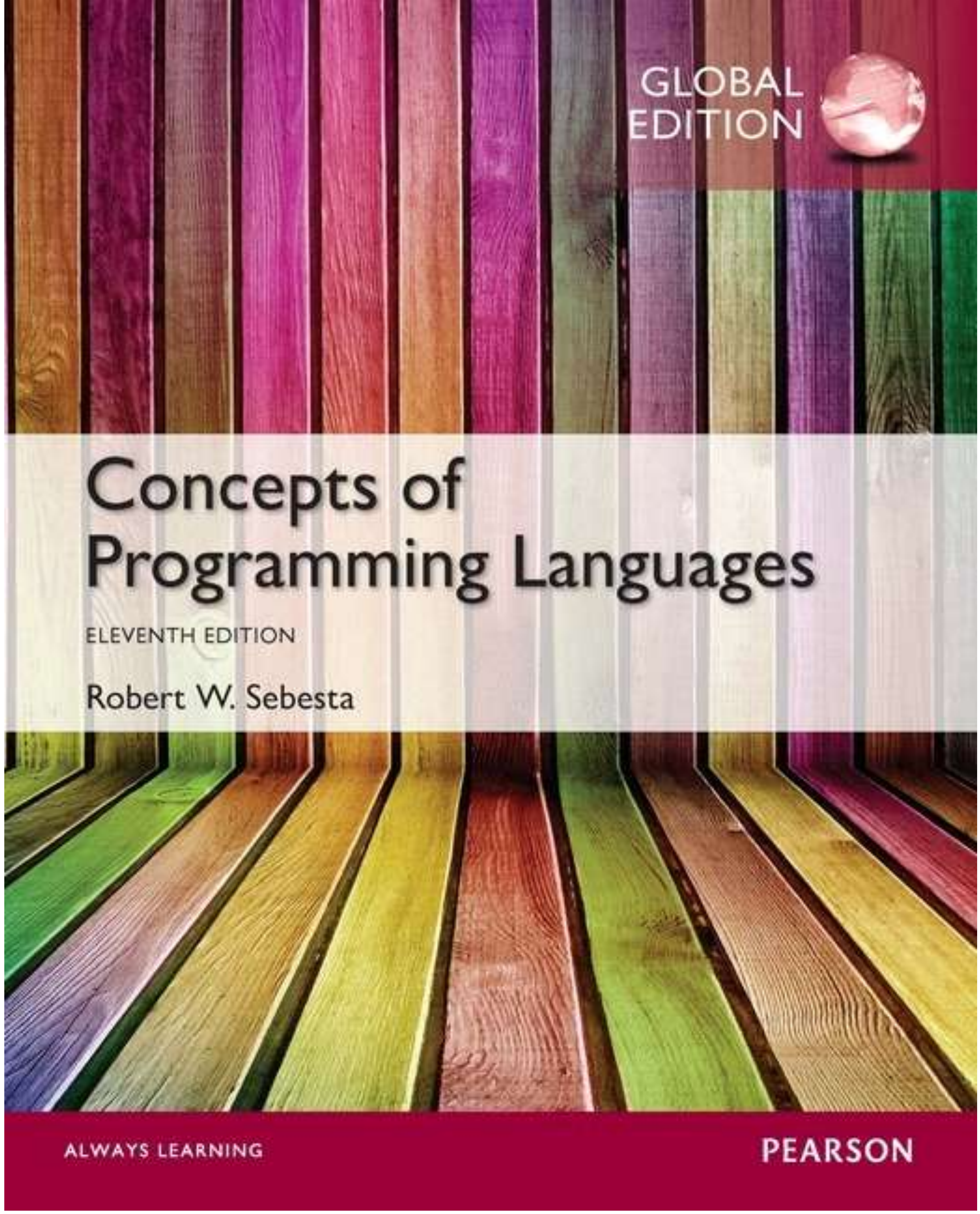
- Connection between control statements and program verification is intimate
- Verification is impossible with `goto` statements
- Verification is possible with only selection and logical pretest loops
- Verification is relatively simple with only guarded commands

Conclusions

- Variety of statement-level structures
- Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability
- Functional and logic programming languages use quite different control structures

Chapter 9

Subprograms



Chapter 9 Topics

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Calling Subprograms Indirectly
- Design Issues for Functions
- Overloaded Subprograms
- Generic Subprograms
- User-Defined Overloaded Operators
- Closures
- Coroutines

Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
    - Discussed in this chapter
  - Data abstraction
    - Emphasized in the1980s
    - Discussed at length in Chapter 11

Fundamentals of Subprograms

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram’s execution terminates

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
  - In Python, function definitions are executable; in all other languages, they are non-executable
  - In Ruby, function definitions can appear either in or outside of class definitions. If outside, they are methods of `Object`. They can be called without an object, like a function
  - In Lua, all functions are anonymous
- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The *parameter profile* (aka *signature*) of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return type

Basic Definitions (continued)

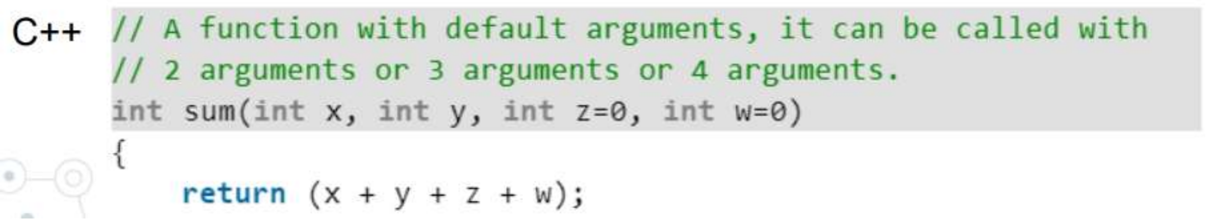
- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram
- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

- Positional
  - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective
- Keyword
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*: User must know the formal parameter’s names

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, PHP), formal parameters can have default values (if no actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated (no keyword parameters)
- Variable numbers of parameters
  - C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **`params`**
  - In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.



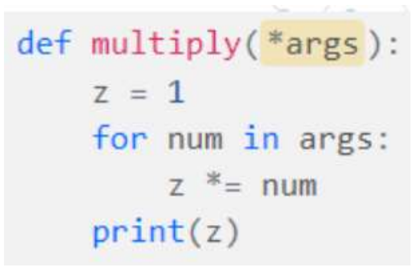
C++

```
// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
```

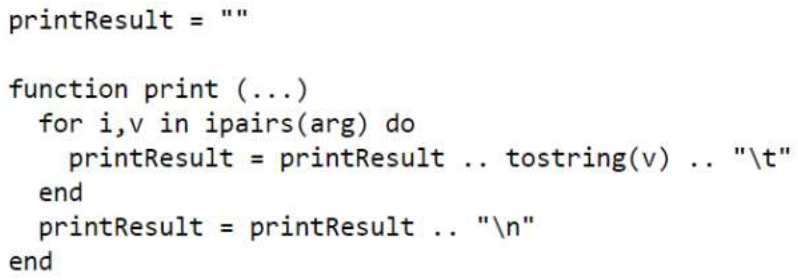
Variable Numbers of Parameters (continued)

- In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk

- In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a **`for`** statement or with a multiple assignment from the three periods



```
def multiply(*args):
    z = 1
    for num in args:
        z *= num
    print(z)
```



```
printResult = ""

function print (...)
  for i,v in ipairs(arg) do
    printResult = printResult .. tostring(v) .. "\t"
  end
  printResult = printResult .. "\n"
end
```

Procedures and Functions

- There are two categories of subprograms
  - *Procedures* are collection of statements that define parameterized computations
  - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
    - They are expected to produce no side effects
    - In practice, program functions have side effects

Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?
- Can subprograms be overloaded?
- Can subprogram be generic?
- If the language allows nested subprograms, are closures supported?

Local Referencing Environments

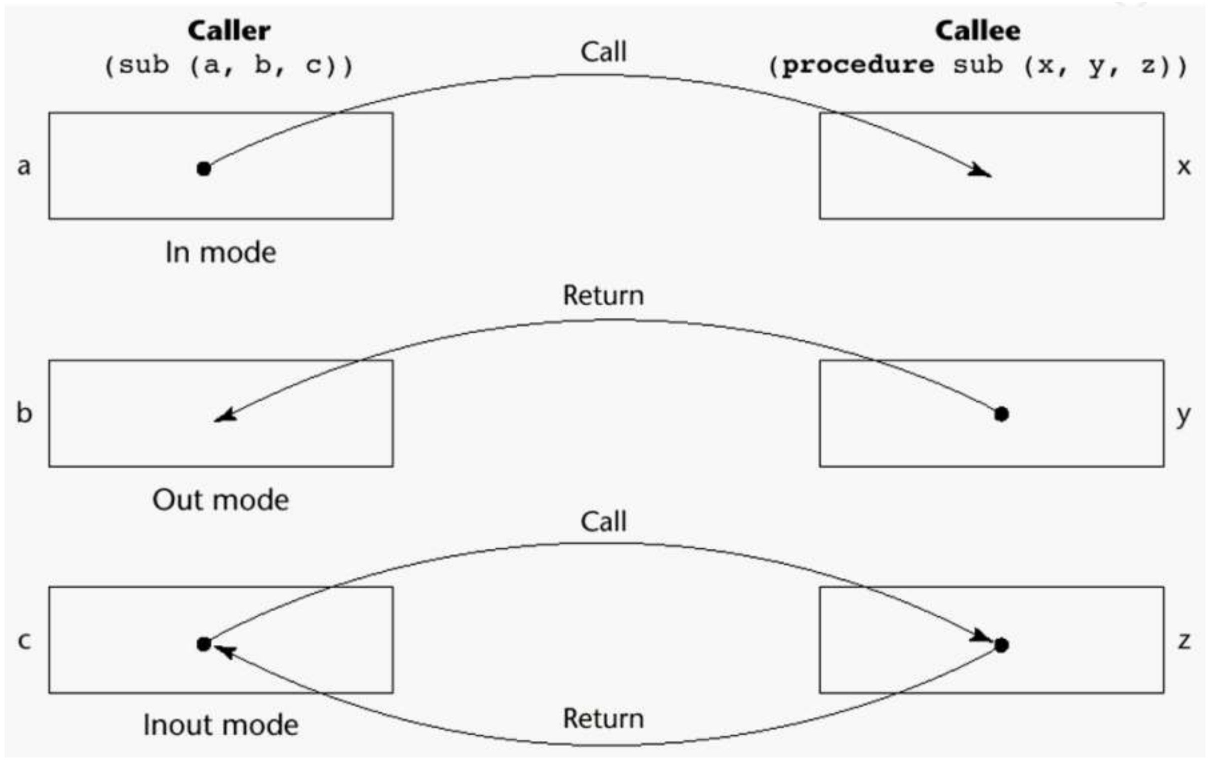
- Local variables can be stack-dynamic
  - Advantages
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages
    - Allocation/de-allocation, initialization time
    - Indirect addressing
    - Subprograms cannot be history sensitive
- Local variables can be static
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

Local Referencing Environments: Examples

- In most contemporary languages, locals are stack dynamic
- In C-based languages, locals are by default stack dynamic, but can be declared `static`
- The methods of C++, Java, Python, and C# only have stack dynamic locals
- In Lua, all implicitly declared variables are global; local variables are declared with `local` and are stack dynamic

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode



Conceptual Models of Transfer

- Physically move a value
- Move an access path to a value

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move
  - Require extra storage location and copy operation
- Potential problems:
  - `sub(p1, p1);` whichever formal parameter is copied back will represent the current value of `p1`
  - `sub(list[sub], sub);` Compute address of list[sub] at the beginning of the subprogram or end?

Pass-by-Value-Result (inout Mode)

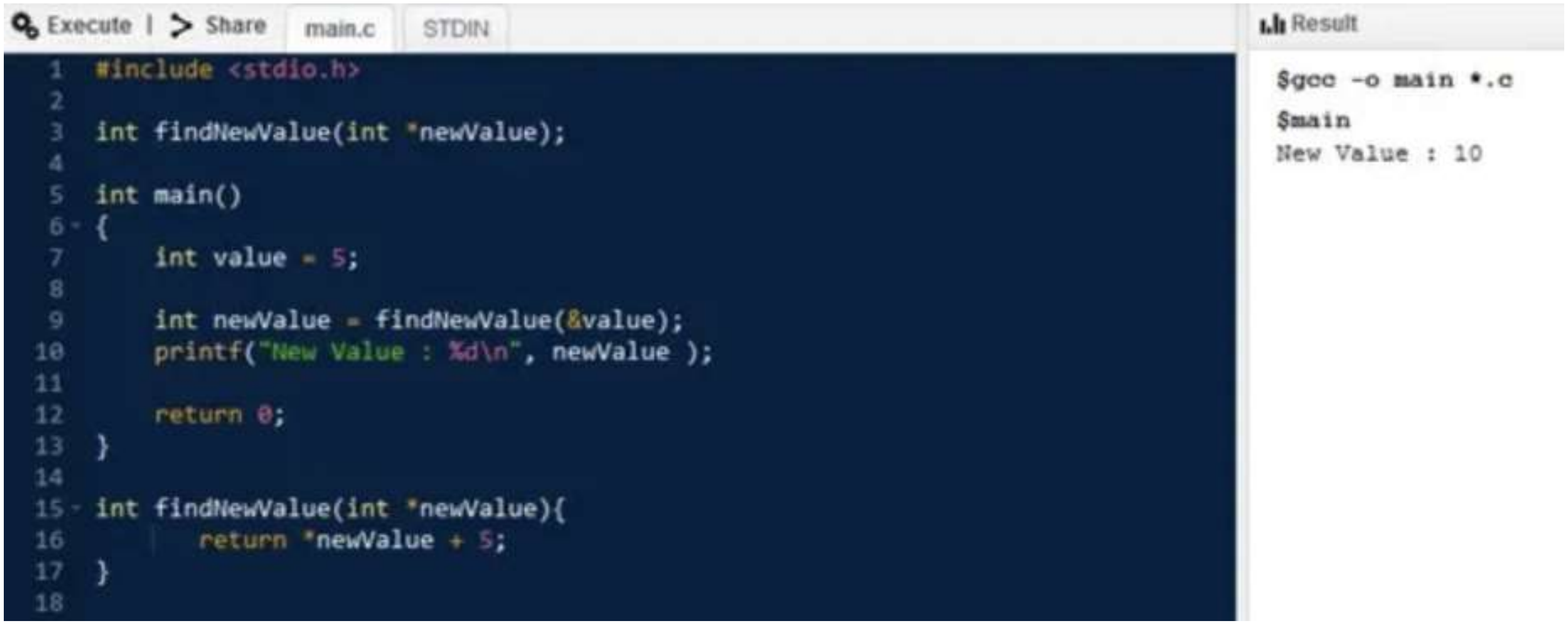
- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

```
fun(total, total);  fun(list[i], list[j];  fun(list[i], i);
```

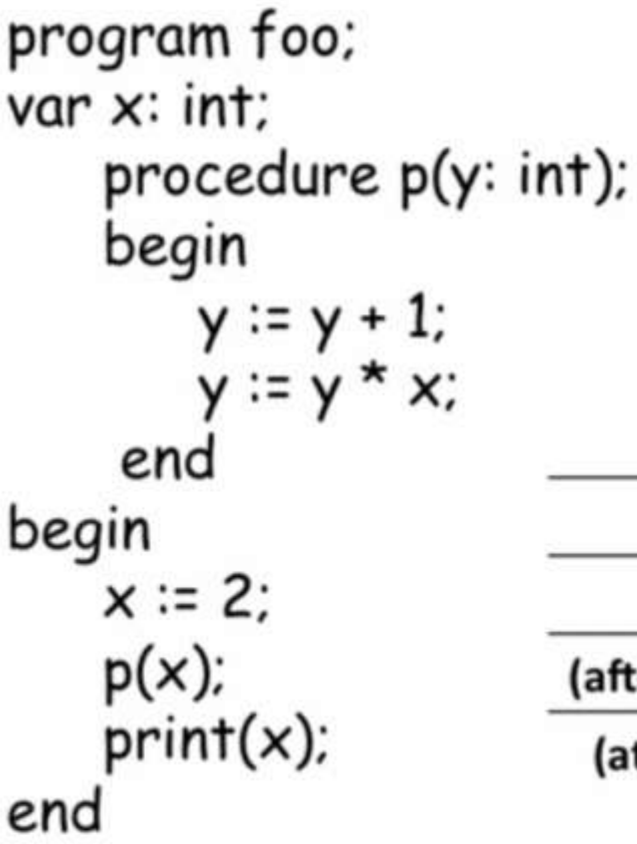
Pass-by-Reference



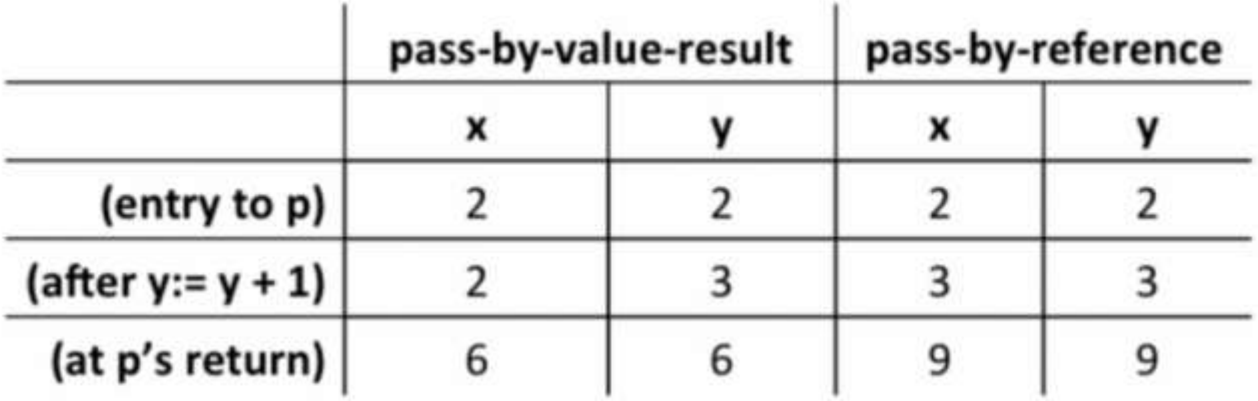
```c
#include <stdio.h>

int findNewValue(int *newValue);

int main()
{
    int value = 5;

    int newValue = findNewValue(&value);
    printf("New Value : %d\n", newValue );

    return 0;
}

int findNewValue(int *newValue){
    return *newValue + 5;
}
```

```
$gcc -o main *.c
$main
New Value : 10
```

Pass-by-Reference X Pass-by-Value-Result



```
program foo;
var x: int;
    procedure p(y: int);
    begin
        y := y + 1;
        y := y * x;
     end
begin
    x := 2;
    p(x);
    print(x);
end
```



	pass-by-value-result		pass-by-reference	
	x	**y**	**x**	**y**
(entry to p)	2	2	2	2
(after y:= y + 1)	2	3	3	3
(at p's return)	6	6	9	9

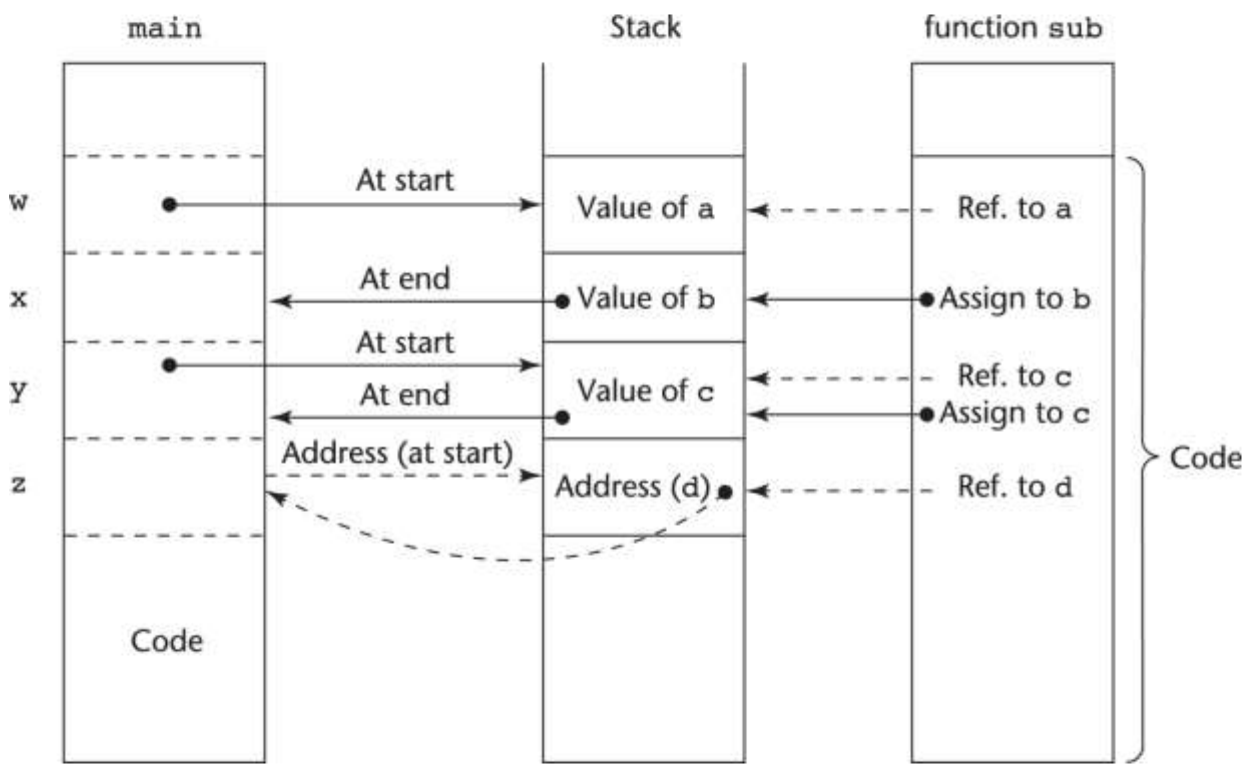
Pass-by-Name (Inout Mode)

- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding
- Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated

Implementing Parameter-Passing Methods

- In most languages parameter communication takes place thru the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack

Implementing Parameter-Passing Methods



Function header: **`void`** `sub(`**`int`** `a,` **`int`** `b,` **`int`** `c,` **`int`** `d)`
Function call in main: `sub(w, x, y, z)`
(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)

Parameter Passing Methods of Major Languages

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- Java
  - All parameters are passed are passed by value
  - Object parameters are passed by reference

Parameter Passing Methods of Major Languages (continued)

- Fortran 95+
  - Parameters can be declared to be in, out, or inout mode
- C#
  - Default method: pass-by-value
  - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with `ref`
- PHP: very similar to C#, except that either the actual or the formal parameter can specify ref
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`
- Python and Ruby use pass-by-assignment (all data values are objects); the actual is assigned to the formal

Type Checking Parameters

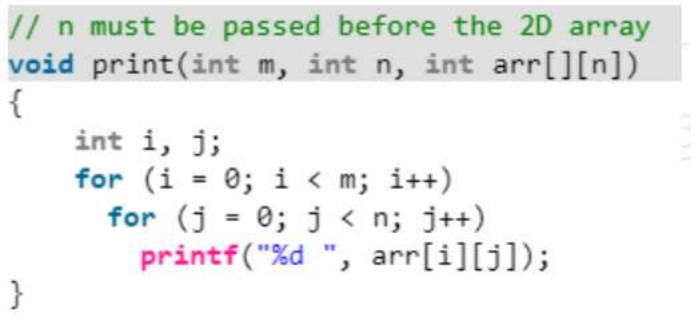
- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal and Java: it is always required
- ANSI C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

Multidimensional Arrays as Parameters

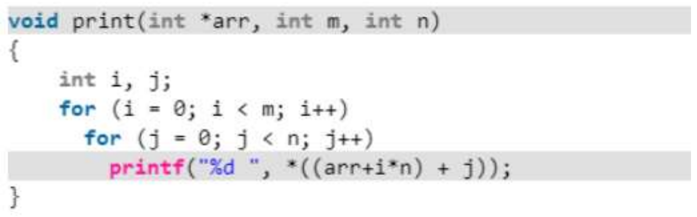
- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function

Multidimensional Arrays as Parameters: C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
- Disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the other dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters



```
// n must be passed before the 2D array
void print(int m, int n, int arr[][n])
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", arr[i][j]);
}
```



```
void print(int *arr, int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", *((arr+i*n) + j));
}
```

Multidimensional Arrays as Parameters: Java and C#

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (`length` in Java, `Length` in C#) that is set to the length of the array when the array object is created

Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

Parameters that are Subprogram Names: Referencing Environment

- *Shallow binding*: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- *Deep binding*: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

```
function sub1() {
  var x;
  x = 1;
  function sub2() {  // this defines sub2
    print(x);
    };
  function sub3() {
    var x;
    x = 3;
    sub4(sub2);  // this passes sub2
    };
  function sub4(subx) {
    var x;
    x = 4;
    subx();  // this enacts sub2
    };
  sub3();
};
```

shallow - output is 4
deep - output is 1
ad hoc - output is 3

Calling Subprograms Indirectly

- Usually when there are several possible subprograms to be called and the correct one on a particular run of the program is not know until execution (e.g., event handling and GUIs)
- In C and C++, such calls are made through function pointers

Calling Subprograms Indirectly (continued)

- In C#, method pointers are implemented as objects called *delegates*
  - A delegate declaration:

    ```
    public delegate int Change(int x);
    ```

    – This delegate type, named `Change`, can be instantiated with any method that takes an `int` parameter and returns an `int` value

    A method: `static int fun1(int x) { … }`

    Instantiate: `Change chgfun1 = new Change(fun1);`

    Can be called with: `chgfun1(12);`
  - A delegate can store more than one address, which is called a *multicast delegate*

Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
  - Lua allows functions to return multiple values

Overloaded Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

Generic Subprograms

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations
- Overloaded subprograms provide *ad hoc polymorphism*
- *Subtype polymorphism* means that a variable of type T can access any object of type T or any type derived from T (OOP languages)
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*
  – A cheap compile-time substitute for dynamic binding

Generic Subprograms (continued)

- C++
  - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
  - Generic subprograms are preceded by a `template` clause that lists the generic variables, which can be type names or class names

```
template <class Type>
  Type max(Type first, Type second) {
  return first > second ? first : second;
  }
```

Generic Subprograms (continued)

- Java 5.0
  - Differences between generics in Java 5.0 and those of C++:

    1. Generic parameters in Java 5.0 must be classes
    2. Java 5.0 generic methods are instantiated just once as truly generic methods
    3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
    4. Wildcard types of generic parameters

Generic Subprograms (continued)

- Java 5.0 (continued)

```
public static <T> T doIt(T[] list) { … }
```

  – The parameter is an array of generic elements (`T` is the name of the type)

  – A call:

```
doIt<String>(myList);
```

Generic parameters can have bounds:

```
public static <T extends Comparable> T
   doIt(T[] list) { … }
```

The generic type must be of a class that implements the `Comparable` interface

Generic Subprograms (continued)

- Java 5.0 (continued)
  - Wildcard types

    `Collection<?>` is a wildcard type for collection classes

```
void printCollection(Collection<?> c) {
    for (Object e: c) {
      System.out.println(e);
    }
 }
```

  - Works for any collection class

Generic Subprograms (continued)

- C# 2005
  - Supports generic methods that are similar to those of Java 5.0
  - One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
  - Another – C# 2005 does not support wildcards

Generic Subprograms (continued)

- F#
  - Infers a generic type if it cannot determine the type of a parameter or the return type of a function – *automatic generalization*
  - Such types are denoted with an apostrophe and a single letter, e.g., `'a`
  - Functions can be defined to have generic parameters

```
let printPair (x: 'a) (y: 'a) =
      printfn "%A %A" x y
```

  - `%A` is a format code for any type
  - These parameters are not type constrained

Generic Subprograms (continued)

- F# (continued)
  - If the parameters of a function are used with arithmetic operators, they are type constrained, even if the parameters are specified to be generic
  - Because of type inferencing and the lack of type coercions, F# generic functions are far less useful than those of C++, Java 5.0+, and C# 2005+

User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby
- A Python example

```
def __add__ (self, second) :
  return Complex(self.real + second.real,
                 self.imag + second.imag)
```

Use: To compute `x + y`, `x.__add__(y)`

Closures

- A *closure* is a subprogram and the referencing environment where it was defined
  - The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
  - A static-scoped language that does not permit nested subprograms doesn’t need closures
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
  - To support closures, an implementation may need to provide unlimited extent to some variables (because a subprogram may access a nonlocal variable that is normally no longer alive)

Closures (continued)

- A JavaScript closure:

```
function makeAdder(x) {
  return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) +
               "<br />");
document.write("add 5 to 20: " + add5(20) +
               "<br />");
```

  – The closure is the anonymous function returned by `makeAdder`

Closures (continued)

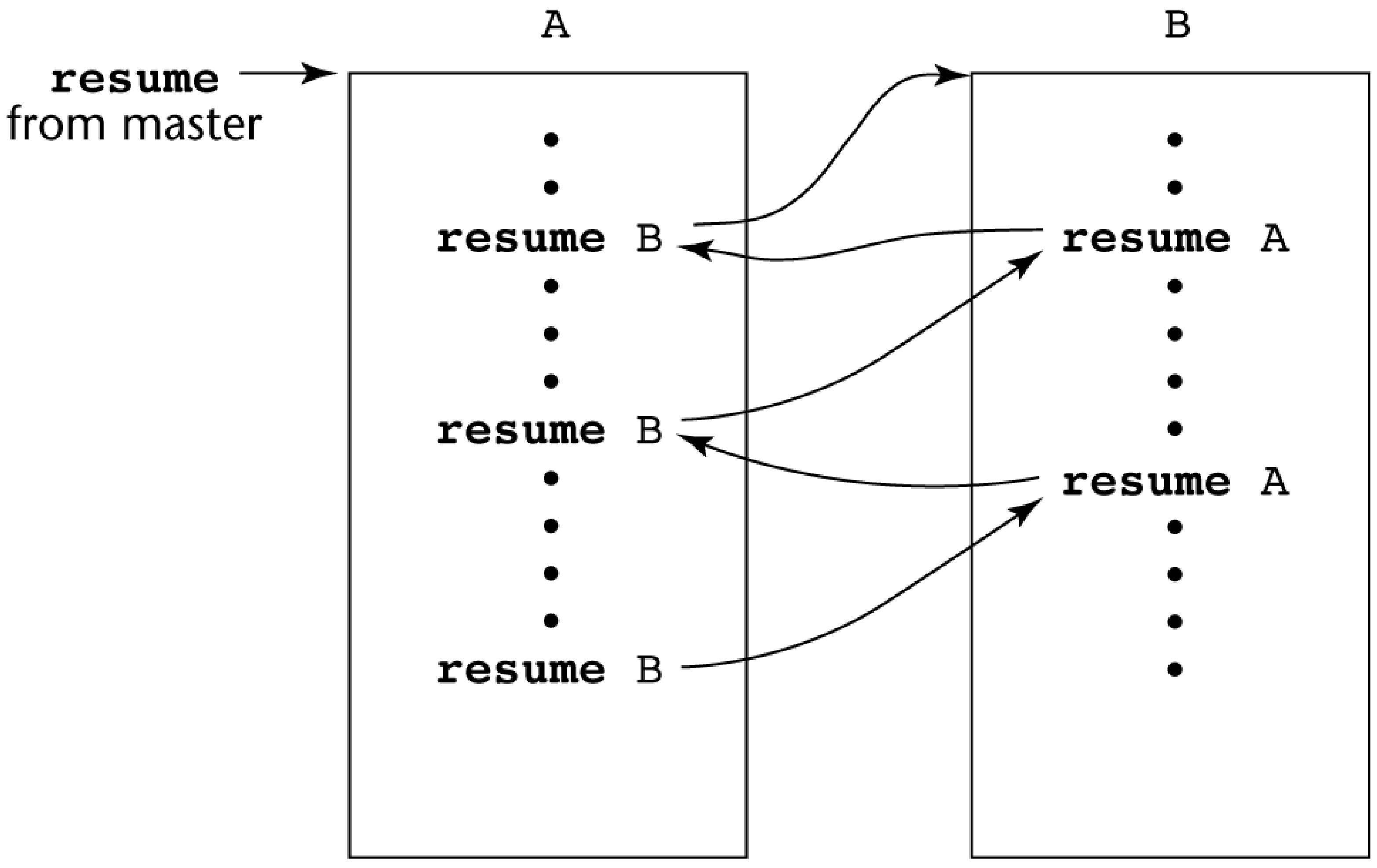
- C#
  - We can write the same closure in C# using a nested anonymous delegate
  - `Func<int, int>` (the return type) specifies a delegate that takes an **`int`** as a parameter and returns and **`int`**

```
static Func<int, int> makeAdder(int x) {
   return delegate(int y) {return x + y;};
}
...
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

Coroutines

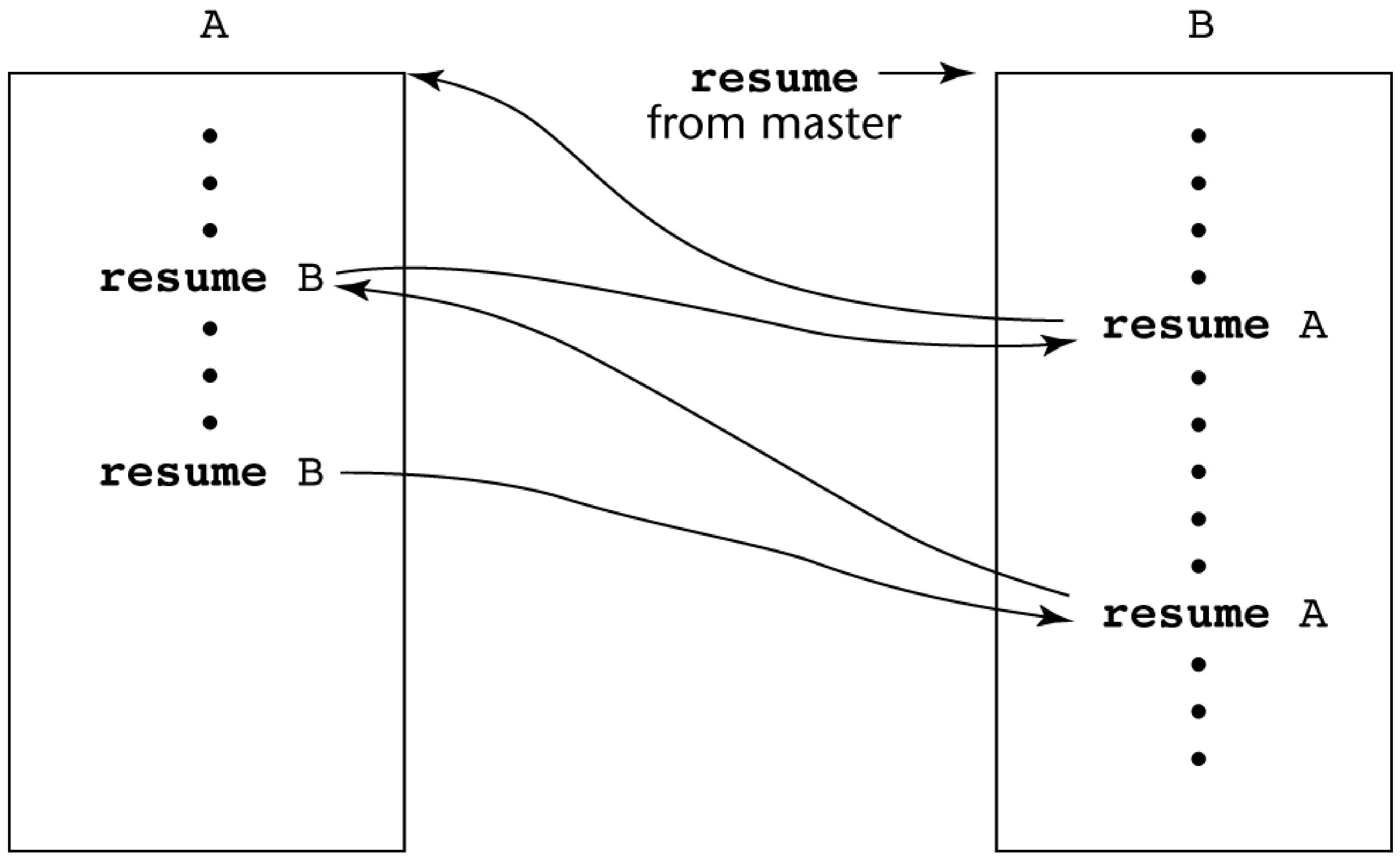
- A *coroutine* is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- Also called *symmetric control:* caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

Coroutines Illustrated: Possible Execution Controls



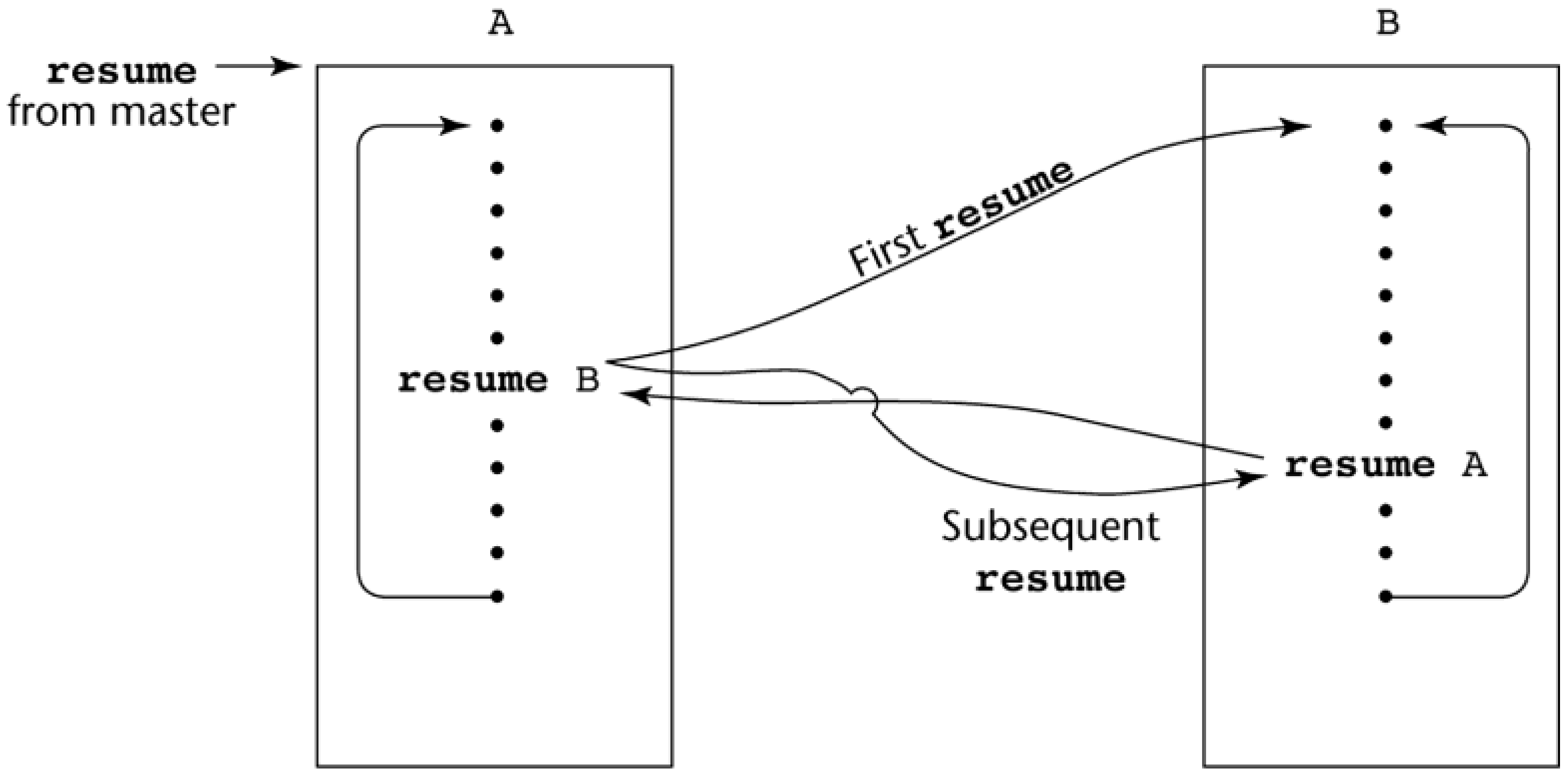
(a)

Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops

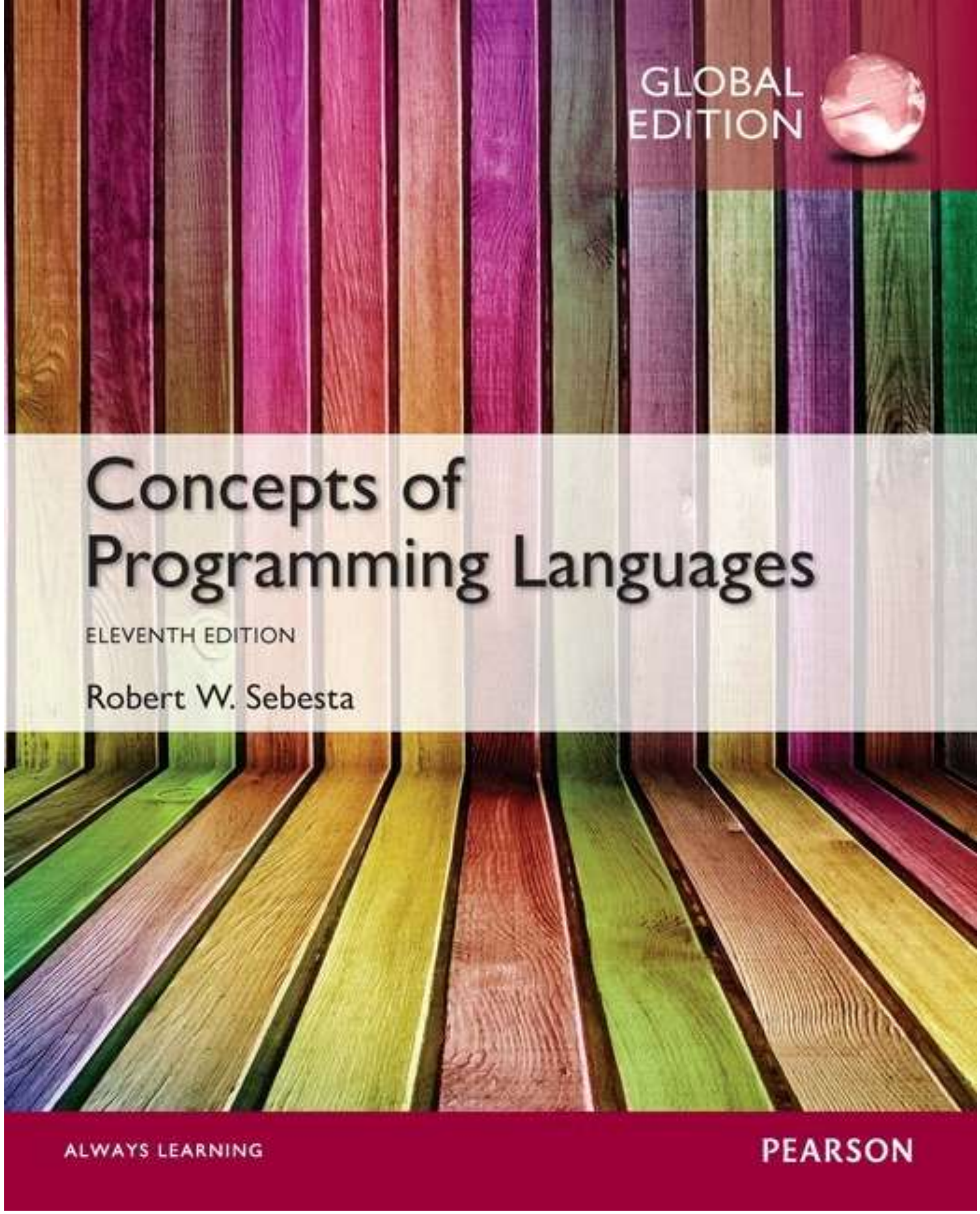


Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A closure is a subprogram and its ref. environment
- A coroutine is a special subprogram with multiple entries

Chapter 10

Implementing Subprograms



GLOBAL EDITION

Concepts of Programming Languages

ELEVENTH EDITION

Robert W. Sebesta

ALWAYS LEARNING

PEARSON

Chapter 10 Topics

- The General Semantics of Calls and Returns
- Implementing “Simple” Subprograms
- Implementing Subprograms with Stack-Dynamic Local Variables
- Nested Subprograms
- Blocks
- Implementing Dynamic Scoping

The General Semantics of Calls and Returns

- The subprogram call and return operations of a language are together called its *subprogram linkage*
- General semantics of calls to a subprogram
  - Parameter passing methods
  - Stack-dynamic allocation of local variables
  - Save the execution status of calling program
  - Transfer of control and arrange for the return
  - If subprogram nesting is supported, access to nonlocal variables must be arranged

The General Semantics of Calls and Returns

- General semantics of subprogram returns:
  - In mode and inout mode parameters must have their values returned
  - Deallocation of stack-dynamic locals
  - Restore the execution status
  - Return control to the caller

Implementing "Simple" Subprograms

- Call Semantics:
  - Save the execution status of the caller
  - Pass the parameters
  - Pass the return address to the called
  - Transfer control to the called

Implementing "Simple" Subprograms (continued)

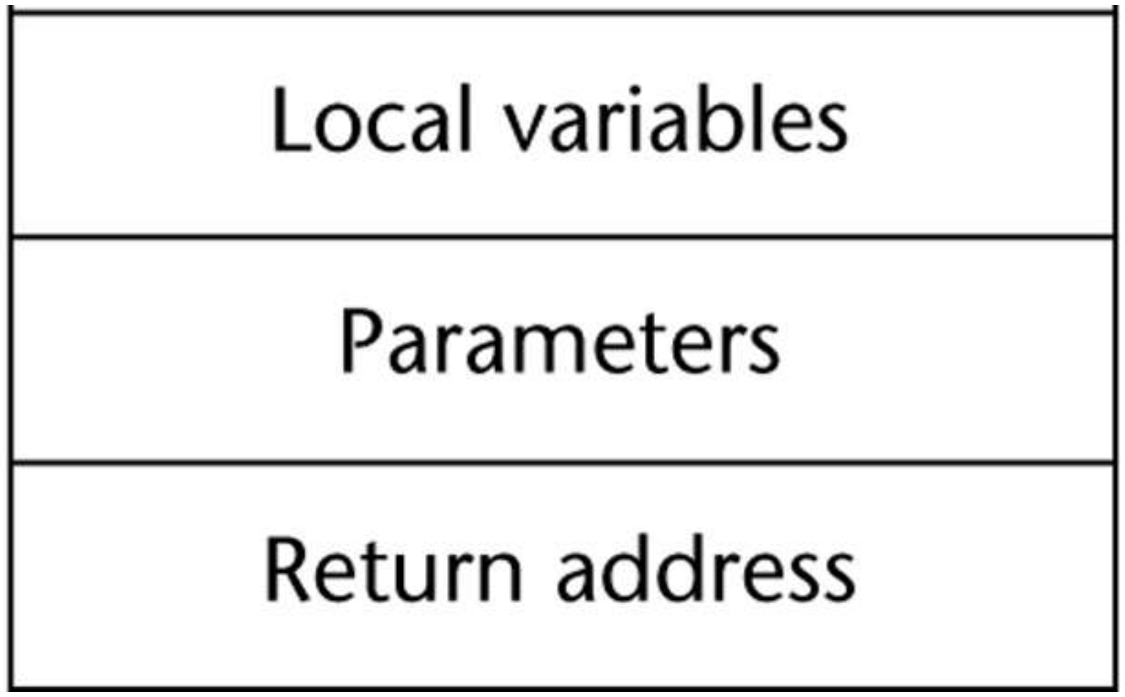
- Return Semantics:
  - If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters
  - If it is a function, move the functional value to a place the caller can get it
  - Restore the execution status of the caller
  - Transfer control back to the caller

- Required storage:
  - Status information, parameters, return address, return value for functions, temporaries

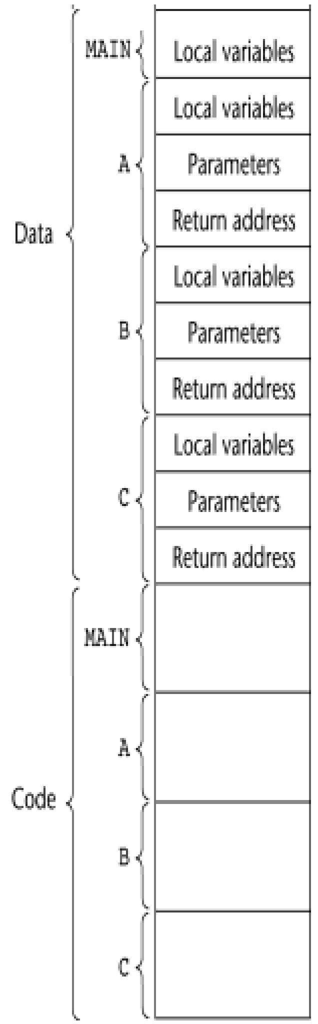
Implementing “Simple” Subprograms (continued)

- Two separate parts: the actual code and the non–code part (local variables and data that can change)
- The format, or layout, of the non–code part of an executing subprogram is called an *activation record*
- An *activation record instance* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

An Activation Record for "Simple" Subprograms

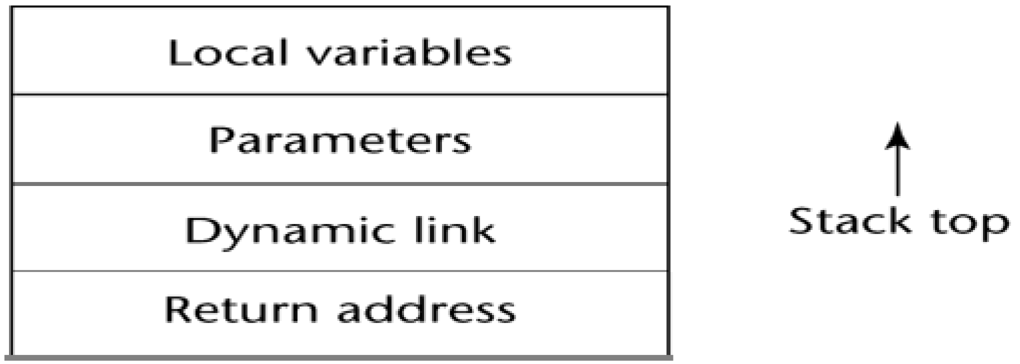


Code and Activation Records of a Program with “Simple” Subprograms



Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
  - The compiler must generate code to cause implicit allocation and deallocation of local variables
  - Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

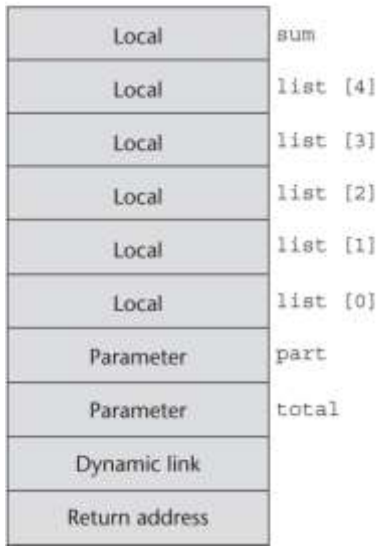


Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- The activation record format is static, but its size may be dynamic
- The *dynamic link* points to the top of an instance of the activation record of the caller
- An activation record instance is dynamically created when a subprogram is called
- Activation record instances reside on the run-time stack
- The *Environment Pointer* (EP) must be maintained by the run-time system. It always points at the base of the activation record instance of the currently executing program unit

An Example: C Function

```
void sub(float total, int part)
{
   int list[5];
   float sum;
   …
}
```



Revised Semantic Call/Return Actions

- Caller Actions:
  - Create an activation record instance
  - Save the execution status of the current program unit
  - Compute and pass the parameters
  - Pass the return address to the called
  - Transfer control to the called
- Prologue actions of the called:
  - Save the old EP in the stack as the dynamic link and create the new value
  - Allocate local variables

Revised Semantic Call/Return Actions (continued)

- Epilogue actions of the called:
  - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
  - If the subprogram is a function, its value is moved to a place accessible to the caller
  - Restore the stack pointer by setting it to the value of the current EP-1 and set the EP to the old dynamic link
  - Restore the execution status of the caller
  - Transfer control back to the caller

An Example Without Recursion

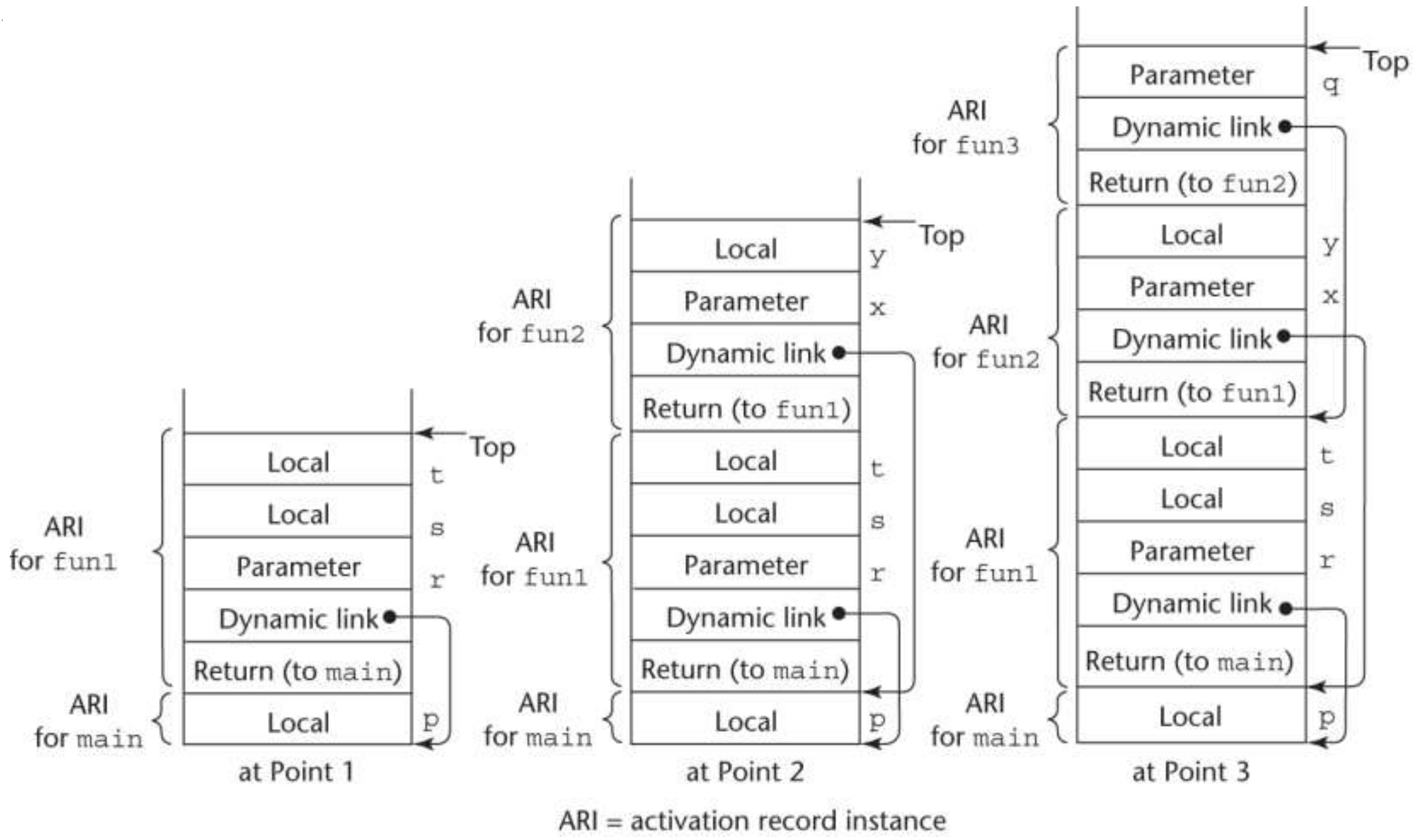
```
void fun1(float r) {
    int s, t;
    ...
    fun2(s);
    ...
}
void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}
void fun3(int q) {
    ...
}
void main() {
    float p;
    ...
    fun1(p);
    ...
}
```

`main` calls `fun1`

`fun1` calls `fun2`

`fun2` calls `fun3`

An Example Without Recursion



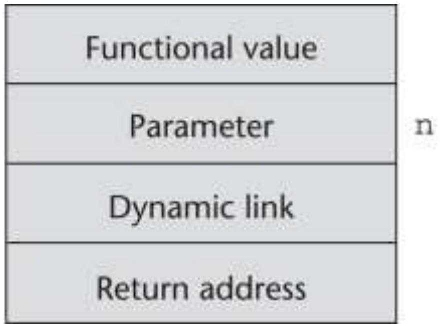
Dynamic Chain and Local Offset

- The collection of dynamic links in the stack at a given time is called the *dynamic chain*, or *call chain*
- Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the *local_offset*
- The local_offset of a local variable can be determined by the compiler at compile time

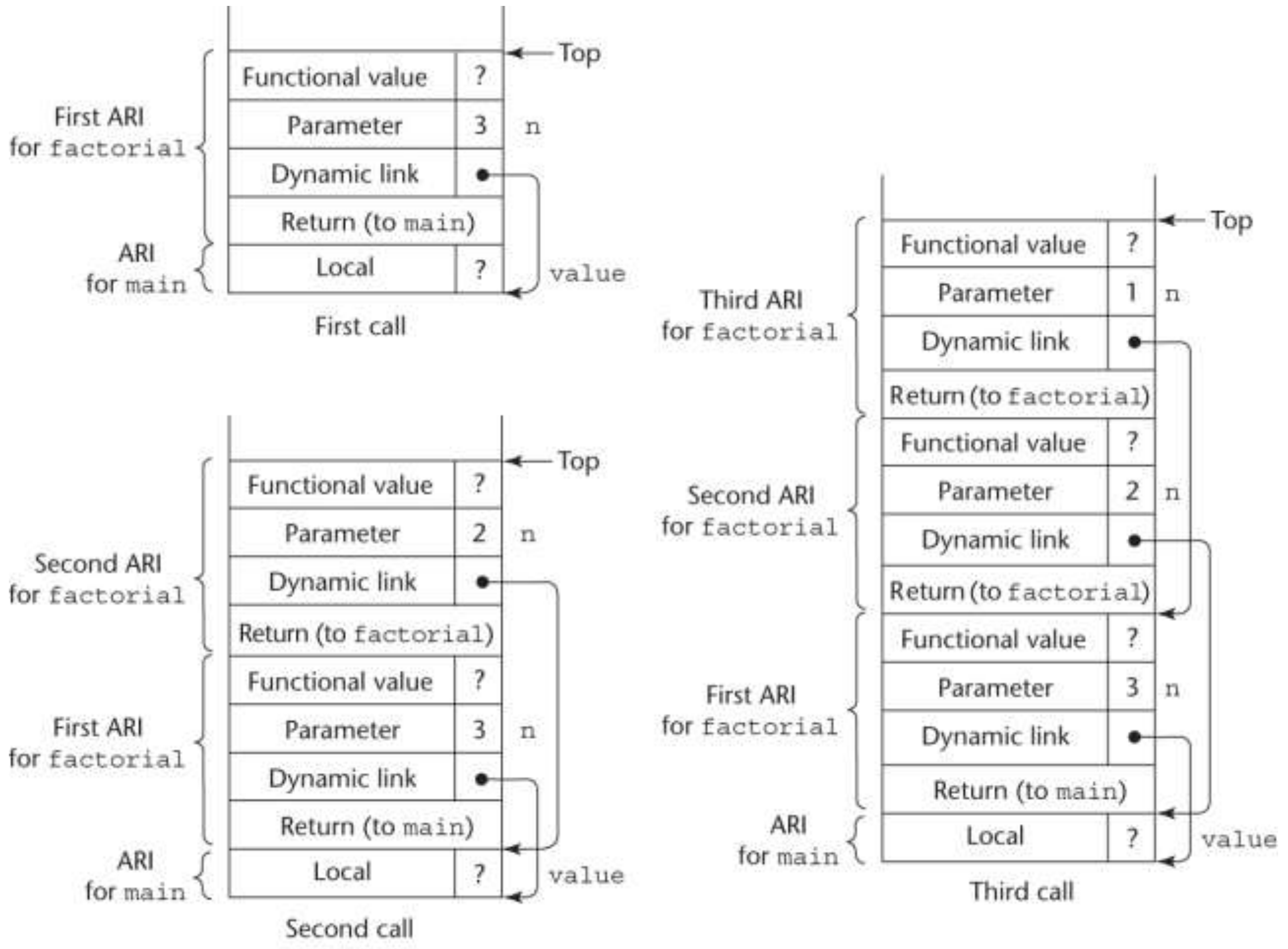
An Example With Recursion

- The activation record used in the previous example supports recursion

```
 int factorial (int n) {
      <-----------------------------1
   if (n <= 1) return 1;
   else return (n * factorial(n - 1));
      <-----------------------------2
}
void main() {
   int value;
   value = factorial(3);
      <-----------------------------3
}
```

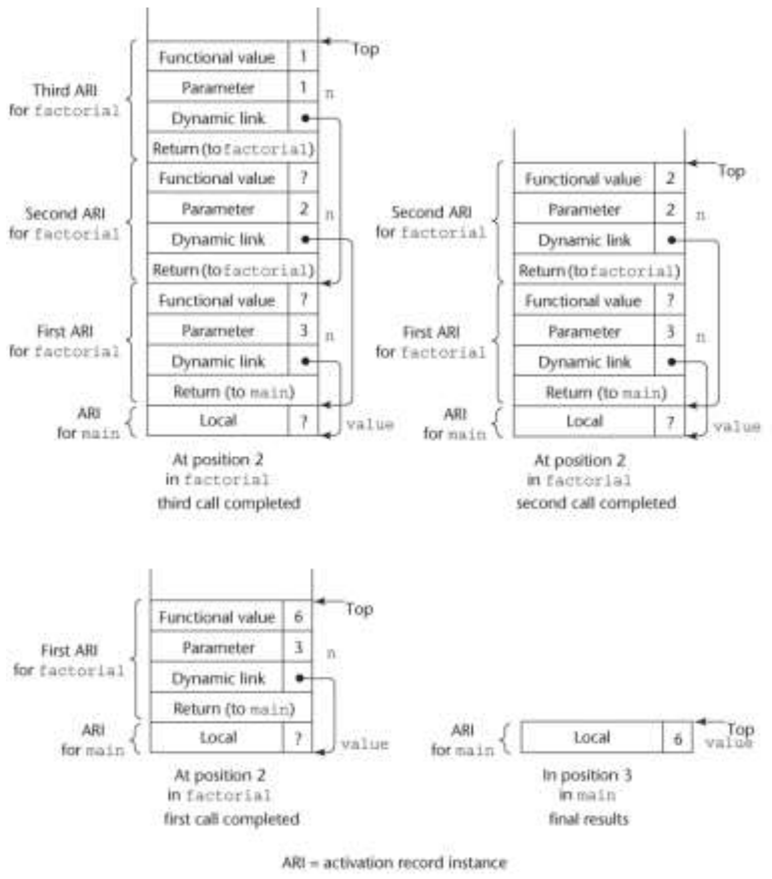


Stacks for calls to `factorial`



ARI = activation record instance

Stacks for returns from `factorial`



Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95+, Ada, Python, JavaScript, Ruby, and Lua) use stack-dynamic local variables and allow subprograms to be nested
- All variables that can be non-locally accessed reside in some activation record instance in the stack
- The process of locating a non-local reference:
  1. Find the correct activation record instance
  2. Determine the correct offset within that activation record instance

Locating a Non-local Reference

- Finding the offset is easy
- Finding the correct activation record instance
  - Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made

Static Scoping

- A *static chain* is a chain of static links that connects certain activation record instances

- The *static link* in an activation record instance for subprogram A points to one of the activation record instances of A's static parent

- The static chain from an activation record instance connects it to all of its static ancestors

- *Static_depth* is an integer associated with a static scope whose value is the depth of nesting of that scope

Static Scoping (continued)

- The *chain_offset* or *nesting_depth* of a nonlocal reference is the difference between the static_depth of the reference and that of the scope when it is declared

- A reference to a variable can be represented by the pair:

  (chain_offset, local_offset),

  where local_offset is the offset in the activation record of the variable being referenced

Example JavaScript Program

```
function main(){
 var x;
 function bigsub() {
  var a, b, c;
  function sub1 {
   var a, d;
   function main(){
 var x;
 function bigsub() {
  var a, b, c;
  function sub1 {
   var a, d;
    a = b + c; <------------------------------1
   ...
  }  // end of sub1
  function sub2(x) {
   var b, e;
```

Example JavaScript Program (continued)

```
    function sub3() {
       var c, e;
      ...
       sub1();
       ...
      e = b + a; ←----------------------------------2
     } // end of sub3 ...
    sub3();
    ...
     a = d + e; ←-----------------------------------3
   } // end of sub2
   ...
   sub2(7);
   ...
  } // end of bigsub
  ...
  bigsub();
  ...
} // end of main
```

Example JavaScript Program (continued)

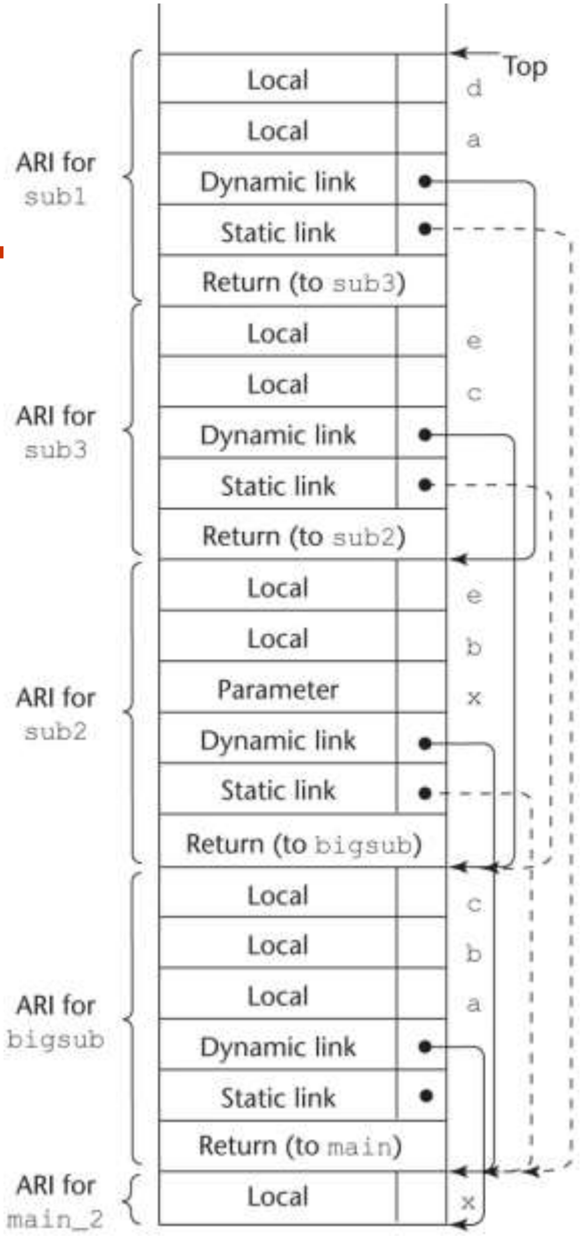
- Call sequence for `main`

  `main` calls `bigsub`
  `bigsub` calls `sub2`
  `sub2` calls `sub3`
  `sub3` calls `sub1`

Stack Contents at Position 1

- Static Chain Maintenance

- At the call,
  - The activation record instance must be built
  - The dynamic link is just the old stack top pointer
  - The static link must point to the most recent ari of the static parent
    - Two methods:
      1. Search the dynamic chain
      2. Treat subprogram calls and definitions like variable references and definitions



ARI = activation record instance

Evaluation of Static Chains

- Problems:
  1. A nonlocal areference is slow if the nesting depth is large
  2. Time-critical code is difficult:
     a. Costs of nonlocal references are difficult to determine
     b. Code changes can change the nesting depth, and therefore the cost

Blocks

- Blocks are user-specified local scopes for variables
- An example in C

```
{int temp;
 temp = list [upper];
 list [upper] = list [lower];
 list [lower] = temp
}
```

- The lifetime of `temp` in the above example begins when control enters the block
- An advantage of using a local variable like `temp` is that it cannot interfere with any other variable with the same name

Implementing Blocks

- Two Methods:
  1. Treat blocks as parameter-less subprograms that are always called from the same location
     - Every block has an activation record; an instance is created every time the block is executed
  2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

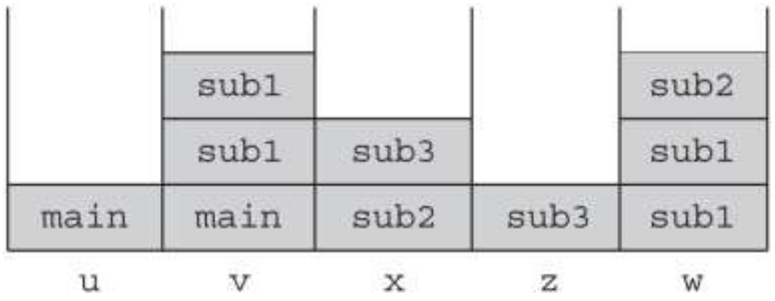
Implementing Dynamic Scoping

- *Deep Access*: non-local references are found by searching the activation record instances on the dynamic chain
  - Length of the chain cannot be statically determined
  - Every activation record instance must have variable names
- *Shallow Access*: put locals in a central place
  - One stack for each variable name
  - Central table with an entry for each variable name

Using Shallow Access to Implement Dynamic Scoping

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```

```
main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3
```



(The names in the stack cells indicate the program units of the variable declaration.)

Summary

- Subprogram linkage semantics requires many action by the implementation
- Simple subprograms have relatively basic actions
- Stack-dynamic languages are more complex
- Subprograms with stack-dynamic local variables and nested subprograms have two components
  - actual code
  - activation record

Summary (continued)

- Activation record instances contain formal parameters and local variables among other things
- Static chains are the primary method of implementing accesses to non-local variables in static-scoped languages with nested subprograms
- Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or thru some central variable table method