Subprograms

COS 301 — Programming Languages

Topics

- Fundamentals of Subprograms
- Design Issues
- Parameter-Passing Methods
- Function Parameters
- Local Referencing Environments
- Overloaded Subprograms and Operators
- Generic Subprograms
- Coroutines

Subprograms

- Subprograms — functions, procedures, subroutines
- Fundamental to all programming languages:
  - Abstraction
  - Separation of concerns
  - Top-down design
- Behavior: closely related to dynamic memory management and the stack

Abstraction

- Process abstraction
  - Only abstraction available in early languages
  - Only data structure available was array, e.g.
- Data abstraction — 80s → present
  - records, abstract data types, packages
  - objects

Terminology

- Functions vs other subroutines
  - **Function:** returns a value, no side effects
  - **Procedure**: executed for its side effects
  - At machine level: no distinction
- Some languages — syntactically distinguish the two:
  - FORTRAN: functions, subroutines
  - Ada, Pascal: functions, procedures
- C-like languages — no syntactic distinction
  - Functions return values generally
  - Those that do not: **void functions**:

```
void foo(int i) {…}
```

Subroutine calls

- Functions:
  - ⇒ **r-values**
  - can appear in expressions
  - e.g.:

```
x = (b*b - sqrt(4*a*c))/2*a
```

- Procedures:
  - invoked as a separate statement
  - e.g.:

```
strcpy(s1, s2);
```

Assumptions

- Subroutine has single **entry point**
  - Exception: **coroutines**
  - Some languages allow multiple entry points (e.g., FORTRAN)
- Calling program suspended during subroutine execution
  - I.e., uses machine language "jump sub" & "return sub" instructions
  - Exception: concurrent programs, **threads**
  - Some languages → support for concurrency: StarLisp, Concurrent Euclid, Java, Fortran, …
- Control returns to caller when subroutine exits

Basic definitions

- Subroutine **definition** — interface + actions
- **Interface** is the abstraction — “API”
- Subroutine **call** — invokes subprogram
- **Formal parameter:** variable listed in subroutine header, used in subroutine
- **Argument:** actual value or address **passed** to parameter in the call statement
- Subprogram **header**: includes name, kind of subprogram (sometimes), formal parameters
- **Signature/protocol** of a subprogram: the **parameter profile**, i.e., number, order, type of parms + **return type**
  - **Declaration:** protocol, but not body
  - **Definition:** protocol + body

Topics

- Fundamentals of Subprograms
- Design Issues
- Parameter-Passing Methods
- Function Parameters
- Local Referencing Environments
- Overloaded Subprograms and Operators
- Generic Subprograms
- Coroutines

Subroutine design issues

- Local vars — static or dynamic?
- Nested subprogram definitions allowed?
- Parameter passing method(s)?
- Type checking for actual/formal parameters?
- Subprograms as parameters?
- If subprograms as parameters, nested subprograms: what is referencing environment?
- Overloading of subprograms allowed (polymorphism)?
- Generic functions allowed?

Function design issues

- Side effects allowed?
  - If not, is this enforced?
  - E.g., disallow call-by-reference
  - E.g., `in` and `out` parameters in Ada

Function design issues

- Types of return values allowed?
  - Imperative languages — often restrict types
    - C: any type <u>except</u> arrays, functions
    - C++: like C, but allows user-defined types to be returned
  - Ada: any type can be returned (<u>except</u> subprograms — which aren't types)
  - Java and C#: methods return any type (<u>except</u> methods, which aren't a type)
  - Python, Ruby, Lisp: methods are first-class objects → any class or method can be returned
  - Javascript, Lisp: (generic, other) functions & methods can be returned
  - Lua, Lisp: functions can return multiple values

Subprogram headers

- Fortran: parameter types defined on separate line:

```
subroutine avg(a,b,c)
real a, b, c
…
real function avg(a,b)
real a, b
```

- C:

```
void avg(float a, float b, float c);
float avg(float a, float b);
```

Subprogram headers

- Ada

```
procedure Avg(A, B: in Integer; C: out Integer)

function Avg(a,b: in Integer) returns Integer
```

Subprogram headers

- Python: can use in code

```
def makeAvg(n):
    if n==3 :
       def newAvg(a,b,c):
           return(a+b+c)/3
    else:
       def newAvg(a,b):
           return(a+b)/2
    return newAvg

foo = makeAvg(2)
foo(20,30) ⟹ 25
```

Subprogram headers

- Lisp:

```
> (defun foo (n)
        (if (= n 3)
         (defun avg (a b c)
           (/ (+ a b c) 3.0))
         (defun avg (a b)
           (/ (+ a b) 2.0))))
FOO
> (setq bar (foo 3))
AVG
> (apply bar '(3 2 100))
35.0
```

Subprogram headers

- Scheme:

```
(define makeAvg
  (lambda (n)
    (if (= n 3)
        (lambda (a b c)
           (/ (+ a b c) 3.0))
        (lambda (a b)
           (/ (+ a b) 2.0)))))
;Value: makeavg

(define avg (makeAvg 3))
;Value: avg

(avg 1 5 12)
;Value: 6.
```

Topics

- Fundamentals of Subprograms
- Design Issues
- Parameter-Passing Methods
- Function Parameters
- Local Referencing Environments
- Overloaded Subprograms and Operators
- Generic Subprograms
- Coroutines

Parameters: Access to Data

- Two ways subprograms can access data:
  - non-local variables
  - parameters
- **Parameters:** more flexible, cleaner
  - use non-local variables →
    - subprogram is restricted to using those names
    - limits environments in which it can be used
  - parameters →
    - provide local names for data from caller
    - can be used in more contexts, regardless of caller's names
    - needed for, e.g., recursion

Access to functions

- Some languages: parameters can hold function/subprogram names
- So can specify functionality as well as data

Actual and formal

- Parameters in subprogram headers = **formal parameters** (or just parameters)
  - Local name for actual values/variables passed
  - Storage usually only bound during subroutine activation
- Parameters in subprogram call = **arguments** (or actual parameters)
  - Arguments bound to formal parameters during subprogram activation or…
  - …value from arguments → formal parameters at start

Arguments ⇔ formal parameters

- How to determine which argument ⇒ which parameter?
  - **Positional parameters**
  - **Keyword parameters**
- Pros and cons:
  - Positional parms: easy to specify, no special syntax, no need to know parameter names
  - Keyword parms: flexible, no need to know order, can provide only some arguments

Example: Python

- Keyword parameters (Python)

```
def listsum(length=my_length, list=my_array,
sum=my_sum):
```

- Mixed positional/kw parameters (Python)

```
def listsum(my_length, list=my_array, sum=my_sum):
```

  - After first keyword parameters, all others must be keyword
  - Can call positional parameter by name, as well!

```
listsum(20, my_array = your_array, sum =20)
listsum(sum=20, my_length=20)
```

Some exceptions

- Perl
  - no formal parameters declared
  - **parameter array:** `@_`
- Smalltalk — unusual infix notation for method names

```
array at: index + offset put: Bag new
array at: 1 put: self
x < 4 ifTrue: ['Yes'] ifFalse: [‘No']
```

- Basically the same for Objective-C

```
[array at: index+offset put: [Bag new]];
[array at: 1    put: [Bag new]];
```

Default values

- Some languages: Default values, optional parameters
  - E.g., C++, Python, Ruby, Ada, Lisp, PHP, VB…
  - Ex.: Python

```
def day_of_week(date, first_day = "Sunday")
```

- Syntax rules can be complex:
  - C++: default parameters last, since positionally placed
  - Some languages with keywords + positional: any omitted parameter must be "keyworded"
  - Lisp: only **&optional** and **&key** parms can have defaults

Variable parameter lists

- C#:
  - methods can accept a variable number of parameters
  - have to be same type
  - the formal parameter is an array preceded by **`params`**
  - Example:

```
public void DisplayList(params int[] list){
   foreach (int next in list){
          Console.WriteLine(“Next value {0}”,
                                 next);}}
```

Variable parameter lists

- C++, C:
  - slightly odd syntax "…"
  - requires some macro/library support: special type (`va_list`), macros to get next arg, etc.

```
void foo(int n, …) {
     va_list params;
     va_start(params, n);
     for (i=0; i<n; i++) {
          … va_arg(params, int)…
      }
     va_end(params);
  }
```

Variable parameter lists

- Ruby:
  - Extra args sent as elements of array to param specified w/ “*”:
  - Kind of complicated:

```
def some_method(a, b, c=5, *p, q)
end
some_method(25,35,45) - a=25, b=35, c=5,
p=[], q=45
some_method(25,35,45,55) - a=25, b=35, c=45,
p=[], q=55
some_method(25,35,45,55,65) - a=25, b=35,
c=45, p=[55], q=65
some_method(25,35,45,55,65,75) - a=25, b=35,
c=45, p=[55,65], q=75
```


Ruby example from http://www.skorks.com/2009/08/method-arguments-in-ruby/.

Variable parameter lists

- Python:
  - `*args` (variable #, → tuple), `**kwargs` (keywords, → dictionary)
  - Example:

```
def myfunc2(*args, **kwargs):
    for a in args:
        print a
    for k,v in kwargs.iteritems():
        print "%s = %s" % (k, v)

myfunc2(1, 2, 3, banan=123)
1
2
3
banan = 123
```


Python example from http://stackoverflow.com/questions/919680/can-a-variable-number-of-arguments-be-passed-to-a-function

Variable parameter lists

- Lua:
  - formal parameter with "…" → map (table)
  - Example:

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

- Lisp: &rest parameter; can mix with positional, &key parms (in complex, perhaps implementation-dependent ways)

```
(defun foo (bar &rest baz) (print bar) (print baz))
(foo 3 4 5 6) ⇒

3
(4 5 6)
```


Lua example from https://www.lua.org/pil/5.2.html

Ruby Blocks

- Ruby provides built-in iterators that can be used to process the elements of arrays; e.g., each and find
  - Iterators are implemented with blocks, which can also be defined by applications
- Blocks can have formal parameters (specified between vertical bars)
  - they are executed when the method executes a `yield` statement

Ruby Blocks

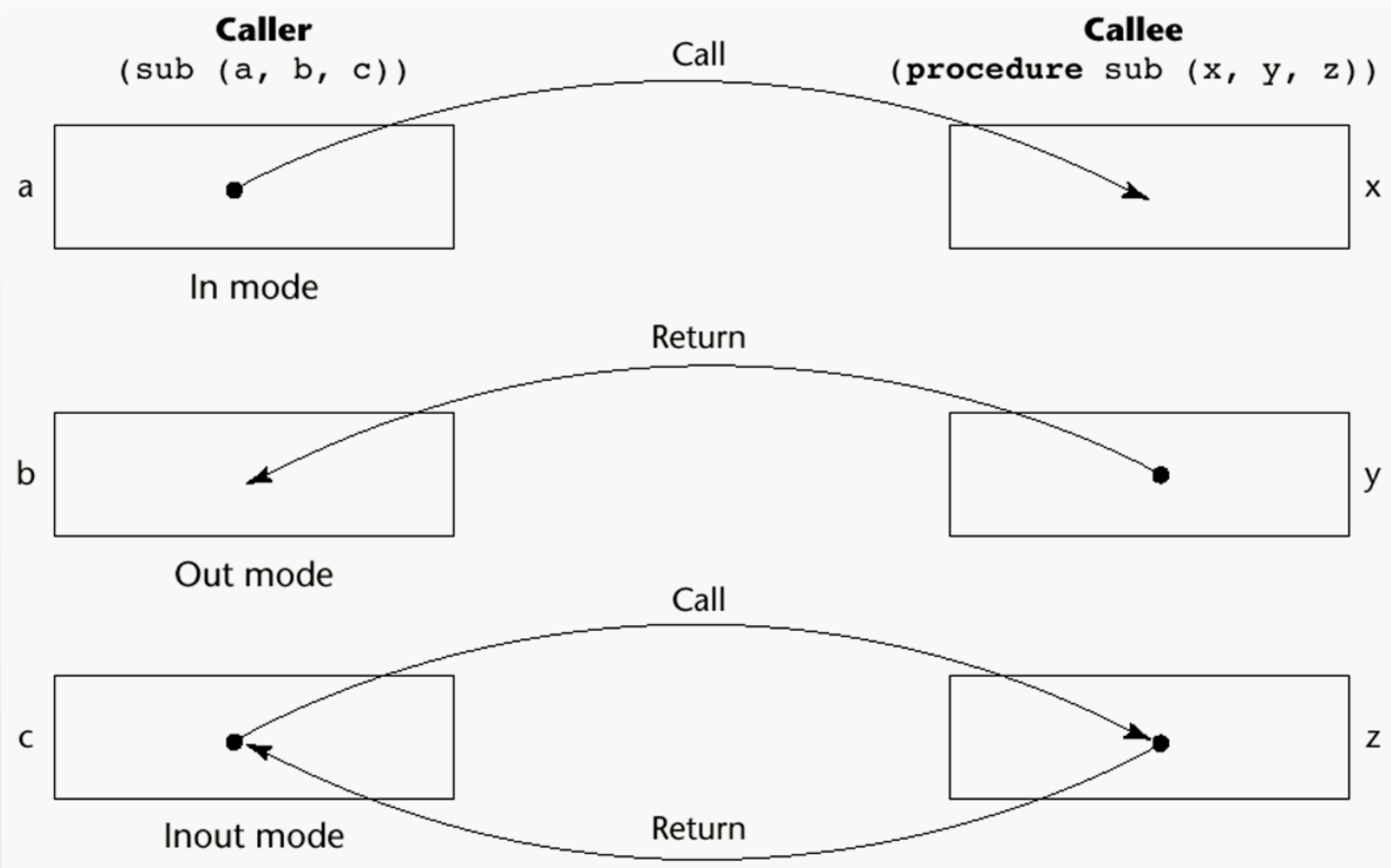
```
def fibonacci(last)
  first, second = 1, 1
  while first <= last
    yield first
    first,second = second,first + second
  end
end

puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "}
puts
```

Parameter passing methods

- **Semantic models** — effects of assignments to formal parameters
- **Implementation models** — techniques of achieving desired semantic model

Semantic models of parameter passing



Conceptual models of transfer

- Actual values can be copied — to caller, callee or both
- Or provide a reference or an access path rather than copying values

Pass-by-value (in mode)

- Value of actual parameter → formal parameter
- Changes formal parameter → no effect on actual parameter
- Implementation:
  - Usually: copy argument to stack
  - Could provide reference or access path
    - not recommended
    - enforcing write protection is not easy
- Disadvantages:
  - additional storage required
  - copy operation can be costly for large arguments

Pass-by-result (out mode)

- No value transmitted to the subprogram
- Formal parameter is local variable
- Subprogram done: parameter value → argument
- Physical copy ⟹ requires extra time, space
- Potential problem: **sub(p1, p1)**
  - whichever formal parameter is copied back will represent the current value of p1
  - Order determines value

Out mode example: C#

- What happens?

```
void fixer(out int x; out int y){
        x = 42;
        y = 33;
}
// what happens with this code?
f.fixer(out a, out a);
```

Out mode example: C#

- What happens?

```
void DoIt(out int x, int index){
   x = 17;
   index = 42;
}
...
sub = 21;
f.DoIt(list[sub], sub);
```

- Depends on when arg addresses are assigned
  - If prior to call, then list[21] = 17
  - If after, then list[42] = 17

Pass-by-reference (in-out mode)

- Pass reference to argument (usually just its address)
- Sometimes called pass-by-sharing
- Advantage: efficiency
  - no copying
  - no duplicated storage
- Disadvantages
  - Creates aliases $\Longrightarrow$ potential unwanted side effects

Distinguishing ref & value parameters

- Language can support ref & value parameters for same types
  - If so: have to make distinction explicit
  - E.g., Pascal:
    - pass-by-value is default:

      ```
      procedure foo(x,y: integer) ...
      ```

    - pass-by-reference:

      ```
      procedure swap(var x,y: integer)
      ...
      ```

Distinguishing ref & value parameters

- Some languages — ref for some, value for others
  - E.g., C: ref for arrays
  - Array "decays" to pointer, so can just use array name
  - E.g.,

```
void foo(int a[]);
```

or

```
void foo(int *a);

int b[100];

foo(b);    or    foo(&b)
```

E.g., swap function

- This won't work in C

```
void swap (int a, int b) {
      int temp = a;
      a = b;
      b = temp;
}
```

- This will:

```
void swap (int *a, int *b) {
      int temp = *a;
      *a = *b;
      *b = temp;
}
```

- To call:

```
 swap (&x, &y)
```

Swap in Java

- Same reasoning in Java

```
void swap (Object a, Object b) {
  Object temp = a;
  a = b;
  b = temp;
}
```

- But you can swap array elements

```
void swap (Object [] A, int i, int j) {
  int temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

Reference parameters must be l-values

- Since an address is passed — can't (usually) pass a literal value as a reference parameter

```
swap (a, b)  //OK
swap(a+1, b) // Not OK
swap(x[j],x[j+1]) // OK
```

- Fortran: all parameters are reference
  - Some early compilers had an interesting bug

```
Subroutine inc(j)
  j = j + 1
End Subroutine
```

  - Calling inc(1) ⟹ the constant "1" would have value of 2 for rest of program!

Using r-values as arguments

- Some languages (e.g., Fortran, Visual Basic) allow non l-values as arguments for reference parameter
- Solution: create temporary variable, pass that address
- On exit: temp variable is destroyed

Pass-by-value-result (in-out mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called **pass-by-copy** — copy-in/copy-out
- Formal parameters have local storage
- Disadvantages: same as pass-by-result & pass by value
- Advantages: same as pass-by-reference

Why use pass-by-value-result?

- Identical to pass-by-reference except when aliasing is involved
- A swap in Ada syntax :

```
Procedure swap3(a : in out Integer,
                b : in out Integer) is
temp : Integer
Begin
     temp := a;
     a := b;
     b := temp;
end swap3;
```

```
a = 3;
b = 2;
swap3(a,b)
```

Now a = 2, b = 3

Pass-by-name

- Pass parameters by **textual substitution**
- Behaves as if textually-substituted for every occurrence of the parameter in the function body — very much like a **macro**
- If argument is a variable name: like call by reference

```
procedure swap(a, b);
 integer a, b;
 begin
     integer t;
     t:= a;
     a := b;
     b := t;
 end;
```

Call swap(i,j):

1. t := i
2. i := j
3. j := t

Pass-by-name

- Cool thing: argument can be an expression
- Expression evaluated each time it's encountered
- Can change variables ⇒ different results each time
- E.g., Jensen's device

Jensen's Device

```
real procedure SIGMA(x, i, n);
 value n;      // x, i called by name
 real x; integer i, n;
 begin
   real s;
   s := 0;
   for i := 1 step 1 until n do
      s := s + x;
   SIGMA := s;
 end;
```

Jensen's Device

```
real procedure SIGMA(x, i, n);
 value n;     // x, i called by name
 real x; integer i, n;
 begin
   real s;
   s := 0;
   for i := 1 step 1 until n do
      s := s + x;
   SIGMA := s;
 end;
```

1. Suppose call is SIGMA(a,b,c) — what is returned?
2. Suppose call is SIGMA(X[i],i,m), where m = max index of X?
3. Suppose call is SIGMA(x[i]*y[i],i,n)?
4. Suppose call is SIGMA(1/i, i, n)?

Jensen's Device

```
real procedure SIGMA(x, i, n);
 value n;     // x, i called by name
 real x; integer i, n;
 begin
   real s;
   s := 0;
   for i := 1 step 1 until n do
      s := s + x;
   SIGMA := s;
 end;
```

- Suppose call is SIGMA(a,b,c):
  - s := s + a
  - does this c times (n := c by value)
  - ⟹ returns a*c

Jensen's Device

```
real procedure SIGMA(x, i, n);
 value n;     // x, i called by name
 real x; integer i, n;
 begin
   real s;
   s := 0;
   for i := 1 step 1 until n do
      s := s + x;
   SIGMA := s;
 end;
```

- Suppose call is SIGMA(X[i],i,m), where m = max index of X:
  - s := s + X[i]
  - does this m times
  - returns s := X[1] + X[2] + … + X[m]

Jensen's Device

```
real procedure SIGMA(x, i, n);
 value n;      // x, i called by name
 real x; integer i, n;
 begin
   real s;
   s := 0;
   for i := 1 step 1 until n do
      s := s + x;
   SIGMA := s;
 end;
```

- Suppose call is SIGMA(x[i]*y[i],i,n):
  - s := s + x[i]*y[i]
  - does this n times
  - returns s := x[1]*y[1] + y[2]*y[2] +… + x[n]*y[n]

Jensen's Device

```
real procedure SIGMA(x, i, n);
 value n;     // x, i called by name
 real x; integer i, n;
 begin
   real s;
   s := 0;
   for i := 1 step 1 until n do
      s := s + x;
   SIGMA := s;
 end;
```

- Suppose call is SIGMA(1/i, i, n); —
  - s := s + 1/i
  - does this n times
  - returns s := 1 + 1/2 + 1/3 + … + 1/n

Pass-by-name

- Implementation of pass-by-name for expressions:
  - Can't assign to them ⇒ compile-time error
  - Don't want to just copy the expression's calculation *n* times
  - Instead, use a **thunk**
- Thunk: subroutine created by compiler encapsulating the expression
  - From Algol
  - Bind thunk call to formal parameter
  - Called each time it's encountered
- Example of **late binding:** evaluation delayed until its occurrence in the body is actually executed
- Dropped by successors (Pascal, Modula, Ada) due to semantic complexity
- Associated with **lazy evaluation** in functional languages e.g., Haskell, somewhat in Scheme (but not Lisp)

Pass-by-name problems

- Complexity, (un)readability
- Unexpected results — e.g., can't write general-purpose swap procedure
- From above:

```
procedure swap(a, b);
    integer a, b;
    begin
        integer t;
        t:= a;
        a := b;
        b := t;
    end;
```

- swap (i , j) — works fine

→ t := i
→ i:= j
→ j := t

Pass-by-name problems

```
procedure swap(a, b);
   integer a, b;
   begin
      integer t;
      t:= a;
      a := b;
      b := t;
   end;
```

- swap(i, A[i]) — doesn't work! a:=b changes i's value
  - → t := i
  - → i:= A[i]
  - → A[i] := t,

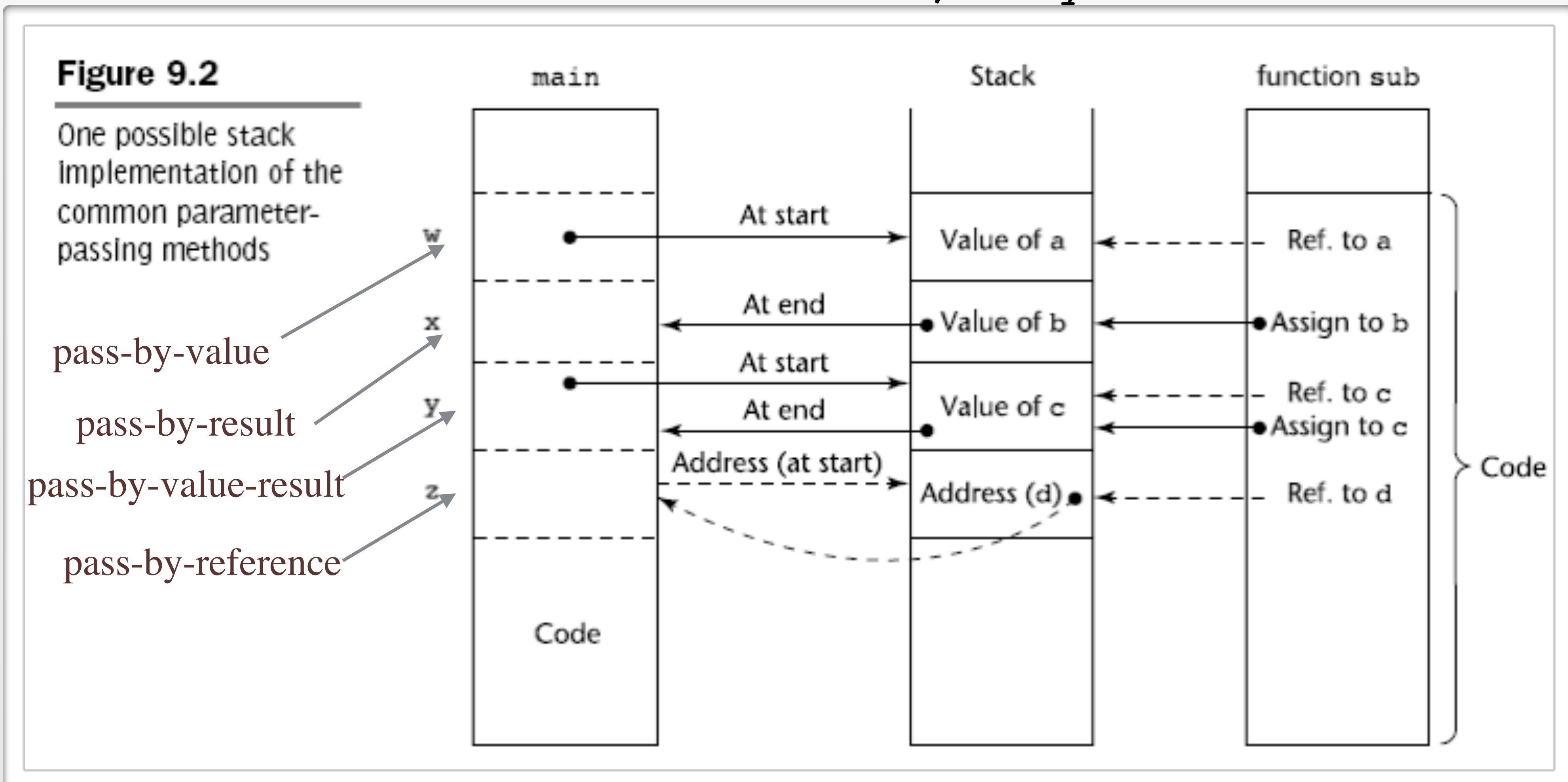
  but really A[A[i]] := t

Implementing parameter-passing methods

- Most languages: via run-time stack
- Local variables (including formal parameters) — addresses are relative to top-of-stack
- Pass-by-reference — simplest: only address placed on stack
- Possible subtle error with pass-by-reference and pass-by-value-result:
  - if argument is a constant, its address placed on stack
  - it's possible to change the *actual* constant via the address

Stack Implementation

```
void sub(int a, int b, int c, int d)
. . .
Main()
  sub(w,x,y,z)   //pass w by val, x by result, y by
                 // value-result, z by ref
```



Parameter passing examples

- C
  - Everything is actually passed by value — including structs
  - Arrays seemingly act as if they are passed by reference
    - This is because an array variable is basically a pointer to the start of the array
    - Thus, attempting to pass by value (where **int X[10]** is the array):
      - void foo(int* A);   void foo(int A[]);   void foo(int A[10]);
      - foo(X);   foo(*X);   foo(&X);
        {int* ptr; ptr = &X[0]; foo(ptr);}
  - Aside: check out www.cdecl.org

Parameter passing examples

- C++:
  - A special type called *reference type* for pass-by-reference
  - E.g.:
    - `int& foo = bar;`
  - References are implicitly dereferenced — so cannot do pointer arithmetic as in C
  - Can have const reference
  - Cannot assign to a reference (can't "reseat" it)

Parameter passing examples

- Java
  - Technically, all parameters are passed by value
  - Most variables (declared to contain objects) are actually *references*, though
  - Formal parameter gets copy of reference — i.e., it points to the same object as the argument
  - Thus, even though it's called by value, can change the argument via the parameter!

Parameter passing examples

- Ada:
  - Semantic modes of parameter passing: in, out, and in out
  - Default: in
  - Parameters declared out: can be assigned, not referenced
  - Parameters declared in: can be referenced, but not assigned
  - Parameters declared in out: can be referenced and assigned

Parameter passing examples

- FORTRAN:
  - Original: all passed by reference
  - Fortran 95
    - Parameters can be declared to be in, out, or inout mode using Intent

```
subroutine a(b,c)
   real, intent(in) :: b
   real, intent(inout) :: c
```

    - Otherwise pass by reference

Parameter passing examples

- C#
  - Default method: pass-by-value
  - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with **`ref`**

```
void foo(int a, ref int b);
…
foo(x, ref y);
```

Parameter passing examples

- PHP:
  - Pass-by-value by default:

```
function foo($bar) {…}
```

  - Use & before variable name for pass-by-reference:

```
function foo(&$bar) {…}
```

Parameter passing examples

- Python and Ruby: **pass-by-assignment**
  - Every variable = reference to an object
  - Acts like pass-by-reference
  - But argument reference is copied → parameter reference
  - Can change what object parameter points to, but if reassign parameter, argument reference unchanged (unlike, e.g., &foo parameters in C++, double pointers in C, etc.)
  - In other words, pretty much like Java’s pass-by-value of a reference!

Parameter passing examples

- Perl:
  - Arguments ⟹ `@_`
  - The things in `@_` are references, which may not be expected
  - Can explicitly pass a reference via `\$foo`
  - Difference:

```
sub foo {
  my ( @bar, $baz) = @_;
     print @bar;
}
my @a = qw(1 2 3 4);
my $b = 0;
&foo(@a, $b);    ⟹ 12340
&foo(\@a, $b);   ⟹ 1234
```

Example after www.perlmonks.org

Parameter passing examples

- Lisp:
  - Pass by value
  - But has references to objects (like Java, e.g.) and other structured things (e.g., cons cells)
  - So works much like Python and Ruby and Java

Type checking parameters

- Important for reliability
  - FORTRAN 77 and original C: none
  - Pascal, FORTRAN 90, Java, and Ada: always required
- C
  - Functions can be declared without types in headers:

```
double sin(x){
    double x;  /* no type checking */
…}
```

  - Or by prototypes with types

```
double sin( double x) {…}
```

  - The semantics of this code differ for each call

```
int ival; double dval;
dval = sin(ival)  /* not coerced with 1st def */
```

Type checking parameters

- C99 and C++ require formal parameters in prototype form
  - But type checks can be avoided by replacing last parameter with an ellipsis

    ```
    int printf(const char* fmt_string, …);
    ```

  - …or by using void pointers

    ```
    int foo(void *a);
    ```

- Python, Ruby, PHP, Javascript, Lisp, etc.
  - NO type checking

Multidimensional arrays as parameters

- Recall address function for array elements:

$$A = B + (I - L)S$$

- Single-dimensional array passed to subroutine → only need to know B, S, and L for parameter
- Multidimensional array:
  - Need to know at least all the subscripts (upper bounds) except the first (for row-major order)
  - E.g., `int A[10,20]`– need to know how many elements/row:

$$\begin{aligned} A_r &= B + (I_r - L_r)S_r \\ A_{ele} &= A_r + (I_c - L_c)S_{ele} \\ S_r &= S_{ele} \times (U_c - L_c + 1) \end{aligned}$$

  - So maximum column index is needed

Multidimensional arrays: C

- All but first subscript required in formal parameter:

```
void fn(int matrix [][10])
```

- Don't need lower bound: it's always 0
- Decreases flexibility → can't handle different-sized arrays on different invocations
- A solution:
  - Pass array as pointer, also pass sizes of other dimensions as parameters
  - It's up to the user to provide the mapping function, e.g.:

```
void fn(int *matptr, int nr, int nc){
…
*(matptr + (row*nc*SizeOf(int)) + col*SizeOf(int)) = x;
…}
```

Multidimensional arrays: Ada

- Multidimensional arrays not a problem in Ada
- Two types of arrays, constrained and unconstrained
  - **Constrained arrays** – size is part of the array's type
  - **Unconstrained arrays** - declared size is part of the object declaration, not type decl
  - If parameter: size of array changes with argument

```
type mat_type is array (Integer range <>) of float;

function matsum(mat : in mat_type) return Float is
   sum: Float := 0.0;
   begin
     for row in mat'range(1) loop
       for col in mat'range(2) loop
          sum := sum + mat(row, col);
       end loop;
     end loop;
     return sum;
end matsum;
```

Multidimensional arrays: Fortran

- Array formal parameters — declaration after header
- Single-dimensional arrays: subscript irrelevant
- Multidimensional arrays:
  - Sizes sent via parameters
  - Parameters used in the declaration of the array parameter
  - The size variables are used in storage mapping function

```
subroutine foo(x,y,z,n)
 implicit none
 integer :: n
 real(8) :: x(n,n), y(n), z(n,n,n)
...
```


Example after http://nf.nci.org.au/training/FortranBasic/slides/slides.032.html

Multidimensional arrays: Java

- Similar to Ada
- Arrays are objects
- All single-dimensional — but elements can be arrays (and thus, arrays can be jagged)
- Array has associated named constant (**length** in Java, **Length** in C#) — set to array length when object created

```
float matsum(float mat[][]) {
    float sum = 0.0;
    for (int r=0; r < mat.length; r++){
        for (int c=0; c < mat[row].length; c++){
            sum += sum + mat[r, c];
        }
    }
  return sum;}
```

Parameter passing design

- Efficiency:
  - Pass-by-reference is more efficient (space, time)
  - Easy two-way transfer of information
- Safety:
  - Limited access to variables best $\Longrightarrow$ one-way transfer
  - in/out parameters (pass-by-value-result) also okay
- Obviously tradeoff between safety, efficiency

Topics

- Fundamentals of Subprograms
- Design Issues
- Parameter-Passing Methods
- Function Parameters
- Local Referencing Environments
- Overloaded Subprograms and Operators
- Generic Subprograms
- Coroutines

Subprograms as parameters

- Useful/necessary, e.g.,
  - Writing generic sort, search routines:

```
(member 1 '(3 4 1 0 5 7) :test #'>)
(member 1 '(3 4 1 0 5 7) :test #'<)
```

  - When creating a subprogram within another → pass it back to caller
  - Often just referred to as "function parameters"
- Some languages (JavaScript, Lisp, Scheme…) allow anonymous function parameters

```
sort(foo,
     function(a,b){if (a<b){return true}
                   else {return false}});
```

Subprograms as parameters

- Issues to address:
  - Are parameter types checked?
  - What is the correct referencing environment for a subprogram that was sent as a parameter?

Function parameters: Type checking

- C/C++ checks types:
  - Can't pass functions directly
  - However, can pass pointers to functions
  - Formal parameter includes the types of parameters, so type checking can work:

```
void foo(float a, int (*fcn)(int, float));
```

- FORTRAN 95: also checks types

Function parameters: Type checking

- Ada:
  - <u>no</u> subprogram parameters
  - alternative: Ada's generic facility (later)
- Java:
  - no <u>method names </u>as parameters
  - however, can have *interfaces* as formal parameters
  - pass as argument an instance implementing interface
  - called method still has to invoke a method of the instance

Referencing environment

- Recall *referencing environment* = collection of all visible names (e.g., variables)
- Referencing environment for nested subprograms?
- E.g., where to find nonlocal variables in call to C in:

```
void C(x){…d…}
void B(void (*fcn)(float)){…fcn(a)…}
void A(){…B(&C)…}
A();
```

- Possibilities: shallow, deep, or ad hoc binding

Shallow (late) binding

```
void C(x){…d…}
void B(void (*fcn)(float)){…fcn(a)…}
void A(){…B(&C)…}
A();
```

- Referencing environment in C:
  - At the place C is **called** — i.e., B's environment when called via "fcn"
- Natural for dynamically-scoped languages

Deep (early) binding

```
void C(x){…d…}
void B(void (*fcn)(float)){…fcn(a)…}
void A(){…B(&C)…}
A();
```

- Environment of variable in C:
  - Environment of the **subprogram definition**
  - I.e., of C's definition
- Natural for statically-scoped (lexically-scoped) languages

Ad hoc binding

```
void C(x){…d…}
void B(void (*fcn)(float)){…fcn(a)…}
void A(){…B(&C)…}
A();
```

- Environment in C is that of the **call statement** that **passed** the function
- I.e., environment of the call in A

Example

```
function sub1(){
  var x;
  function sub2(){
    alert(x);
  }
  function sub3(){
    var x;
    x = 3;
    sub4(sub2)
  }
  function sub4(subx){
    var x;
    x = 4;
    subx();
  }
  x = 1;
  sub3();
}

sub1();
```

- What is the output of `alert(x)`:
  - with shallow binding?
  - with deep binding?
  - with ad hoc binding?

Example

```
function sub1(){
   var x;
   function sub2(){
      alert(x);
   }
   function sub3(){
      var x;
      x = 3;
      sub4(sub2)
   }
   function sub4(subx){
      var x;
      x = 4;
      subx();
   }
   x = 1;
   sub3();
}

sub1();
```

- sub1 → sub3 → sub4 → sub2
- What does x refer to?
- Shallow binding:
  - Reference to x is bound to local x in sub4 so output is 4
- Deep binding:
  - Referencing environment of sub2 is x in sub1 so output is 1
- Ad hoc binding:
  - Referencing environment of sub2 is x in sub3 so output is 3
- E.g.: Javascript uses ad hoc binding

Topics

- Fundamentals of Subprograms
- Design Issues
- Parameter-Passing Methods
- Function Parameters
- Local Referencing Environments
- Overloaded Subprograms and Operators
- Generic Subprograms
- Coroutines

Overloaded subprograms

- Same name as another in referencing environment
  - Each has to have same protocol
  - I.e., same parameter profile + same return value type
- C++, Java, C#, and Ada:
  - predefined overloaded subprograms
  - user-defined overloaded subprograms
- **Disambiguation** can be significant problem

Disambiguation

- Consider these prototypes:
  - double fun (int a, double b);
  - double fun (double a, int b);
- Sometimes disambiguation is easy:
  - fun(1, 3.14);
  - fun(3.14, 1);
- But sometimes problematic:
  - int z = (int)fun(1,2);
  - No prototype matches the calling profile
  - Can match either through coercion — so which to choose?

Disambiguation

- One solution: rank the coercions
  - but in what order?
- Another problem: default parameters
      - `double fun(int a = 5);`
      - `double fun(float b = 7.0);`
  - Call: `x = fun();`
  - Which one should be called?

User-defined overloaded

- Operators can be overloaded in some languages
- E.g., Ada:

```
function "*" (A,B: in Vec_Type): return Integer is
   Sum: Integer := 0;
   begin
   for Index in A'range loop
         Sum := Sum + A(Index) * B(Index)
   end loop
   return sum;
end "*";
```

`c = a * b;` → this function if a, b are Vec_Type

`c = x * y;` → multiplication if x, y are ints or floats

Topics

- Fundamentals of Subprograms
- Design Issues
- Parameter-Passing Methods
- Function Parameters
- Local Referencing Environments
- Overloaded Subprograms and Operators
- Generic Subprograms
- Coroutines

Polymorphism and generics

- Operator & subprogram overloading are examples of polymorphism
- One type — **generic functions**
  - Function/operator that can be applied to different, related types for same general result
  - E.g., generic sort routines
- Another kind of **generic function** has multiple methods for different kinds of parameters
  - E.g., CLOS
  - Methods usually have to have congruent parameter profiles — e.g., same # positional parms, etc.
  - Somewhat like C++'s **template functions**
- Advantages: readability, lack of code duplication

Generic subprograms

- **Subtype polymorphism:** in OO languages (later)
- **Duck typing:**
  - "If it walks like a duck, quacks like a duck…"
  - Ignoring type of parameters entirely
  - Relies on operators/functions being defined for the parameter's type
  - Often in dynamically-typed languages (e.g., Python, Ruby, JavaScript, Lisp)
  - E.g.,

```
(defun move-to (object location &optional (delta .5))
   (orient object location);object needs orient method
   (loop until (near object location) ;needs near method
      do (move object delta))   ;needs move method
```

  - Convenient — not very safe
  - Compare to Java's interface mechanism?

Parametric polymorphism

- **Parametric polymorphism:** compile-time polymorphism
  - Relies on defining a subprogram with generic parameters
  - Make different **instances** of subprogram with actual parameter type
- All instances behave the same

Generic Ada sort

```
generic
   type element is private;
   type list is array(natural range <>) of element;
   with function ">"(a, b : element) return
    boolean;
procedure gen_sort (in out a : list);
```

```
procedure gen_sort (in out a : list) is
begin
   for i in a'first .. a'last - 1 loop
      for j in i+1 .. a'last loop
         if a(i) > a(j) then
            declare t : element;
            begin
               t := a(i); a(i) := a(j); a(j) := t;
            end;
         end if;
      end loop;
   end loop;
end gen_sort;
```

```
procedure sort is new sort(Integer, ">" );
procedure sort2 is new sort(Float,  ">" );
procedure sort3 is new sort(MyElementType,  "MyComparisonOp" );
```

C++ Templates

- Basic implementation mechanism similar to macro expansion

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a > b)? a : b;
  return (result);
}
```

```
int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Generics through subclassing

- OO languages (Java, Smalltalk, Objective-C, Lisp/CLOS,…)
  - Everything is an object (most languages)
  - Can define subclasses
  - Can define **methods** that of same name for different subclasses
- Behavior depends on the classes of the parameters

Topics

- Fundamentals of Subprograms
- Design Issues
- Parameter-Passing Methods
- Function Parameters
- Local Referencing Environments
- Overloaded Subprograms and Operators
- Generic Subprograms
- Coroutines

Coroutines

- **Coroutine:** Subprogram with multiple entry points
  - Controls them itself
  - Maintains state between activations
  - Coordinates with other coroutines to carry out work
- Sometimes called **symmetric control** — caller/called are on equal basis
- Languages with direct (sometimes limited) support for coroutines:

  C#   F#   Go   Haskell   Javascript   Lua

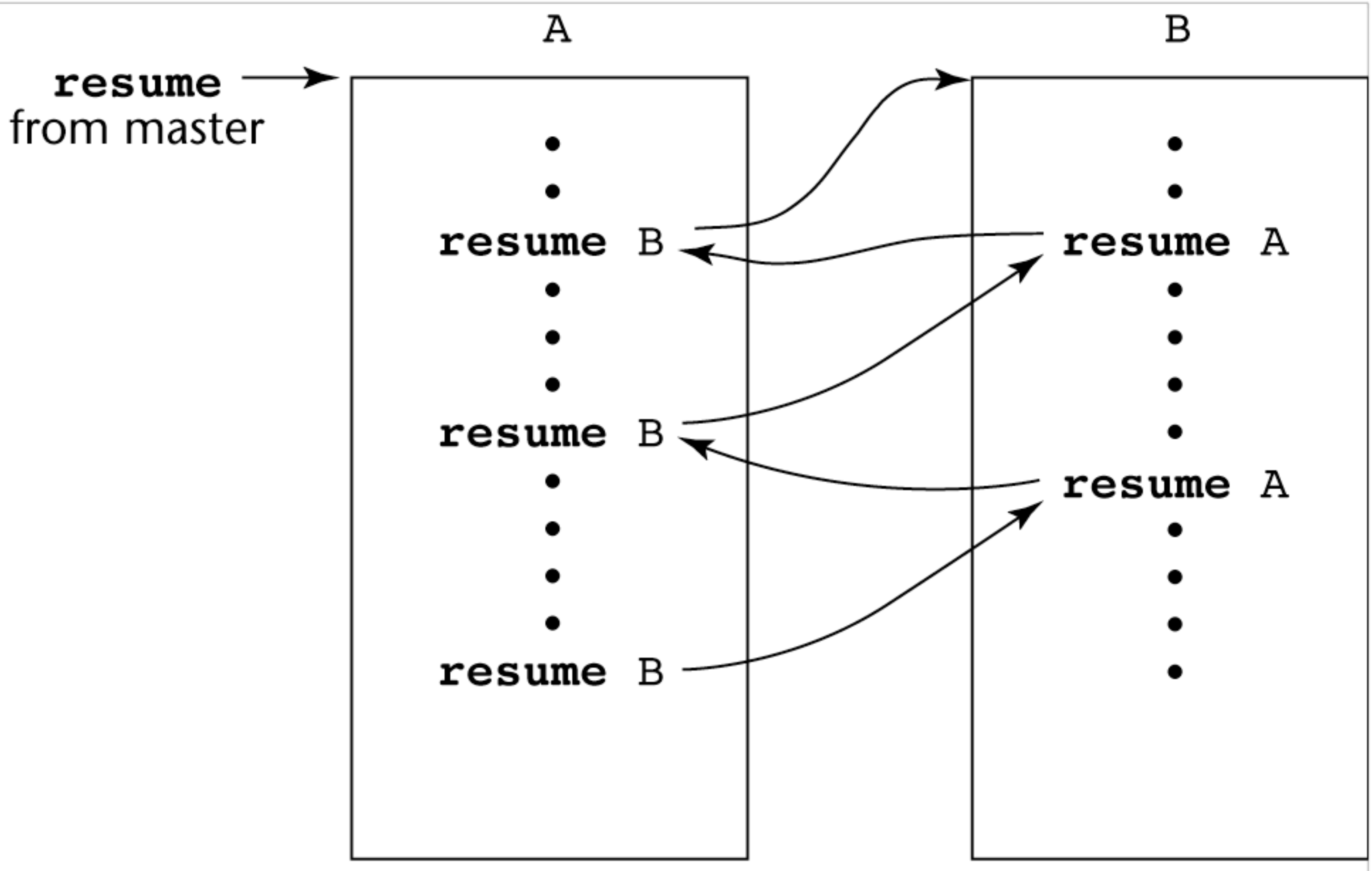
  Perl   Prolog   Python   Ruby   Scheme

  Lisp (some Lisps; or implement with macros [Graham])

Coroutines

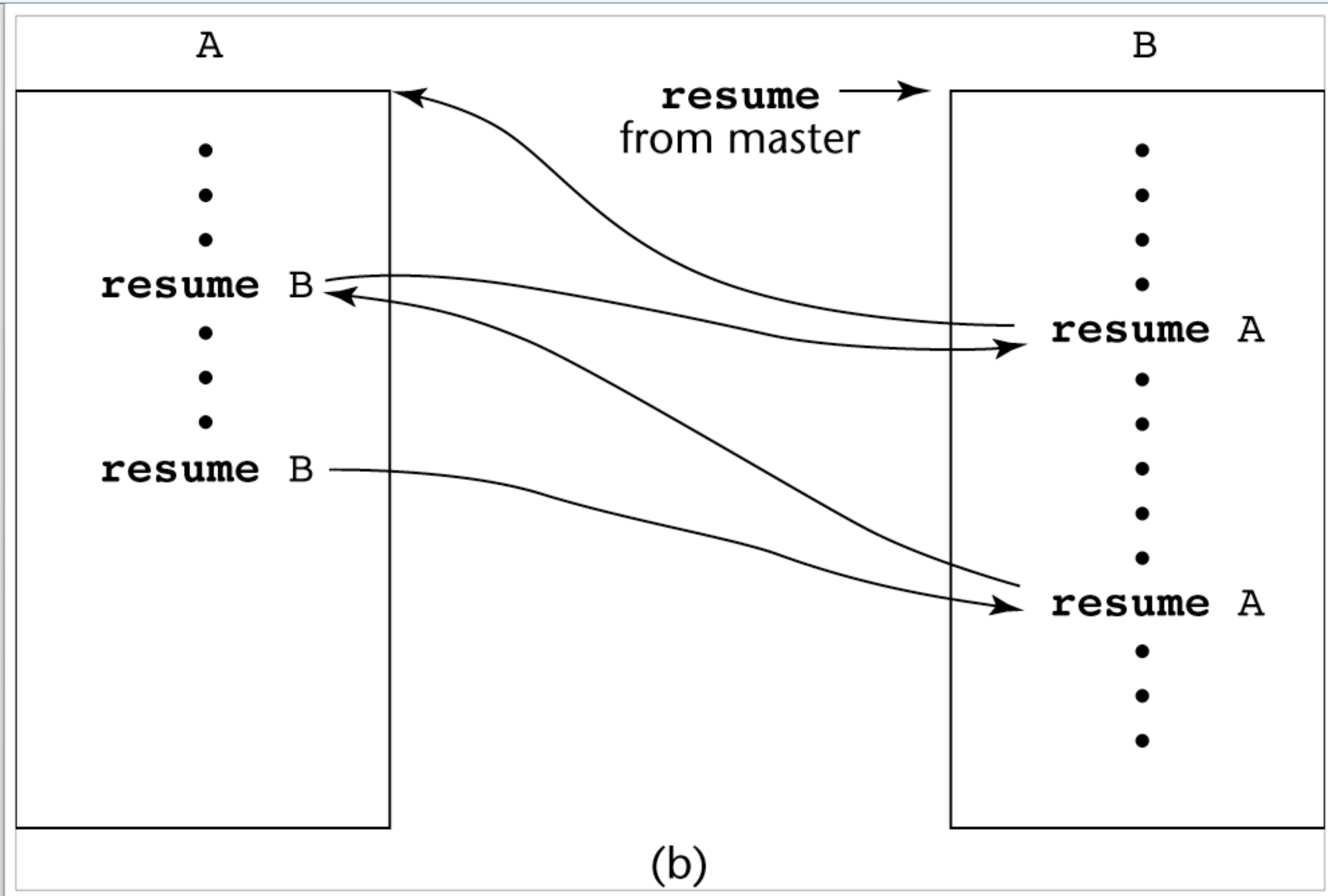
- Coroutine call is named a **resume**
- First resume is coroutine's beginning entry point
- Subsequent resumes: enter at point after statement previously executed
- Coroutines repeatedly resume each other — possibly forever
- Provides pseudo-concurrent execution — execution is interleaved, not overlapped

Possible Execution



(a)

Possible Execution



Coroutine applications

- Card (or other turn-taking) games: each coroutine → one player
- Producer-consumer: one routine produces items & queues them, other removes and consumes them
- Efficient traversal of complex data structures
- Coroutines very similar to multiple threads
  - Can be used for many of same applications
  - Some languages (e.g., Lisp Machine Lisp) → pseudo-concurrency within interpreter
  - Coroutines never execute in parallel — unlike OS threads (on multiple cores — otherwise interleaved by OS)

Simple Coroutine Example:

```
--[[
    This program shows how a coroutine routine works - by
    starting a function running, then suspending it at
    a yield to continue later
    ]]
    function gimmeval()
            me = 1243
            while (me > 1234) do
                    coroutine.yield()
                    me = me - 1
                    print ("duh") --

           end
    end
    -- main code
    print ("Simple co-routine")
    instream = coroutine.create(gimmeval)
    while coroutine.status(instream) ~= "dead" do
            -- kick coroutine and let it run until
            -- it suspends or dies
            coroutine.resume(instream)
            print (me, "Here - at ")
     end
```


from http://www.wellho.net/resources/ex.php4?item=u114/coro

Output

```
--[[ --------------- Sample output -----------------
    [trainee@easterton gwh]$ lua coro
    Simple co-routine
    1243    Here - at
    duh
    1242    Here - at
    duh
    1241    Here - at
    duh
    1240    Here - at
    duh
    1239    Here - at
    duh
    1238    Here - at
    duh
    1237    Here - at
    duh
    1236    Here - at
    duh
    1235    Here - at
    duh
    1234    Here - at
    [trainee@easterton gwh]$

    ]]
```