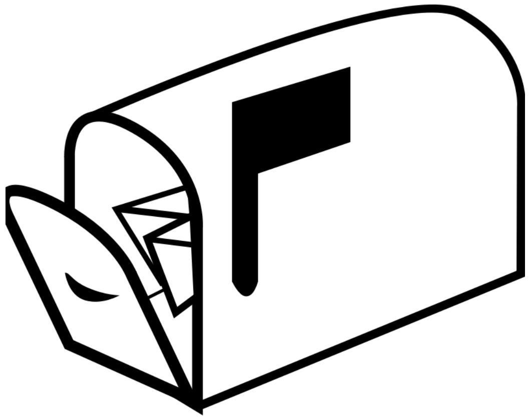
CS 430
Spring 2019

Mike Lam, Professor



Subprograms and Parameters

Subprograms

- Subprograms are fundamental building blocks for programs
  - A form of process abstraction
  - Facilitates modularity and code re-use
- General subprogram characteristics
  - Single entry point
  - Caller is suspended while subprogram is executing
  - Control returns to caller when subprogram completes
  - Most subprograms have names (but not all!)
- Procedure vs. function vs. method
  - Functions have return values
  - Methods are associated with classes & objects

Subprograms

- New-ish terms
  - Header: signaling syntax for defining a subprogram
  - Parameter profile: number, types, and order of parameters
  - Signature/protocol: parameter types and return type(s)
  - Prototype: declaration without a full definition
  - Referencing environment: variables visible inside a subprogram
  - Call site: location of a subprogram invocation

Parameters

- Formal vs. actual parameters
  - Formal: parameter inside subprogram definition
  - Actual: parameter at call site
- Semantic models: *in*, *out*, *in-out*
- Implementation models (when/what is copied):
  - **Pass-by-value (*in, value*)**
  - Pass-by-result (*out, value*)
  - Pass-by-copy or pass-by-value-result (*in-out, value*)
  - **Pass-by-reference (*in-out, reference*)**
  - **Pass-by-name (*in-out, text*)**

Example

- Trace x, y, a, b, c, and d after each numbered line:

```
   foo(a,b,c,d):
1:   a = a + 1            # a is passed by value
2:   b = b + 1            # b is passed by copy
3:   c = c + 1            # c is passed by reference
4:   d = d + 1            # d is passed by name

     x = [1,2,3,4]
     y = 2
5:   foo(x[0],x[1],y,x[y])
```

```
      x = [1,2,3,4]  y=2   a=1  b=2  c=&y  d=x[y]
   1:
   2:
   3:
   4:
   5:
```

Parameter Implementations

- Pass-by-value
  - Pro: simple
  - Con: costs of allocation and copying
  - Often the default
- Pass-by-reference
  - Pro: efficient (only copy 32/64 bits)
  - Con: hard to reason about, extra layer of indirection, aliasing issues
  - Often used in object-oriented languages
- Pass-by-name
  - Pro: powerful
  - Con: expensive to implement, very difficult to reason about
  - **Rarely used!**

Other Design Issues

- How are formal/actual parameters associated?
  - Positionally, by name (“keyword parameters”), or both?
- Are parameter default values allowed (i.e., can a parameter be optional)?
  - Any parameter or only the right-most one?
- In what order are parameters handled/copied?
  - Generally left-to-right or right-to-left
- Are parameters type-checked?
  - Statically or dynamically?

```
def bar(a:0, b:1)
  puts “a is #{a}, b is #{b}”
end

bar(a:3, b:4)
bar(b:4, a:3)
bar(a:3)
bar(b:4)
bar()
```

Name association in Ruby

```
def foo(a=0, b=1)
  puts “a is #{a}, b is #{b}”
end

foo(3, 4)
foo(3)
foo()
```

Default parameters in Ruby

Other Design Issues

- Are local variables statically or dynamically allocated?
- Can a subprogram have a variable number of parameters?
- Can subprograms be nested?
- Can subprograms be polymorphic?
  - Ad-hoc/manual polymorphism via overloading
  - Subtype polymorphism
  - Parametric polymorphism (e.g., templates or generics)
- Are side effects allowed?
- Can a subprogram return multiple values?
  - Unnecessary if robust support for tuples and pattern matching

Other Design Issues

- Can subprograms be passed as parameters?
  - How is this implemented?
    - Explicit via function pointers or implicit (e.g., lambdas)
  - Are subprograms first-class?
    - Can they also be returned or stored in variables?
  - If nested subprograms are also allowed, which referencing environment should be used?
    - Shallow/dynamic: call that invoked the subprogram
    - Deep/static: definition of subprogram
    - Ad-hoc: call that passed the subprogram (not used)

Misc. Topics

- Macros
  - Call-by-name, “executed” at compile time
- Closures
  - A nested subprogram and its referencing environment
- Coroutines
  - Co-operating subprograms

```
def foo(a)
  inner = 10
  return proc {puts "#{a} + #{inner} is #{a + inner}"}
end

p = foo(5)
puts p.class
p.call
```

Closures in Ruby

```
var q := new queue

coroutine produce
  loop
    while q is not full
      add new items to q
    yield to consume

coroutine consume
  loop
    while q is not empty
      remove/use some items from q
    yield to produce
```

https://en.wikipedia.org/wiki/Coroutine