CSE 413 Autumn 2008

Ruby Blocks, Procs, and Closures

Blocks

- Recall that any method call can have a trailing block, which can be executed by the method (almost like a coroutine)

```
all_words = “”
words.each { | w | all_words = all_words + w + “ ” }
```

Block Execution

- A block is executed in the context of the method call.
  - Implications: Access to variables at the call location; return from a block returns from surrounding method

```
def search(it, words)
   words.each { | w | if it == w return }
   puts “not found”
end
```

yield

- Any method call can include a trailing block. A method “calls” the block with a yield statement.

```
def repeat
  yield
  yield
end
repeat { puts “hello” }
```

Output:

```
hello
hello
```

yield with arguments

- If the block has parameters, you can provide expressions with yield to pass arguments

```
def xvii
  yield 17
end
xvii { | n | puts n+1 }
```

  - This is exactly what an iterator does

Blocks and Procs

- Blocks (and methods) are not objects in Ruby – i.e., not things that can be passed around as first-class values
- But we can create a Proc object from a block
  - Procs are closures consisting of the block and the surrounding environment
  - Variations: procs and lambdas; slightly different behavior
  - Several different ways to construct these; see the language documentation for details

Making Procs

- In a method, can have a parameter that explicitly represents the block

```
def return_a_block (& block)
  block.call(17)
  return block
end
```

  - The ‘&’ turns the block into a proc object
  - Proc objects support a call method

Proc.new; lambdas

- Can also create a proc object explicitly

```
p = Proc.new { | x, y | x+y }
…
p.call(x,y)
```

- The kernel's lambda method also creates proc objects

```
is_positive = lambda {|x| x > 0 }
```

Procs vs. Lambdas

- A Proc is a block wrapped in an object – and behaves just like a block
  - In particular, a return in a Proc will return from the *surrounding* method where the Proc's closure was created
    - Error if that method has already terminated
- A Lambda is more like a method
  - Return just exits from the lambda

Functional Programming in Ruby

- Ruby is definitely not a functional programming language, but with blocks, procs, and lambdas, you can do most anything you could in a functional language
- For a good discussion, see ch. 6 in *The Ruby Programming Language* by Flanagan and Matsumoto