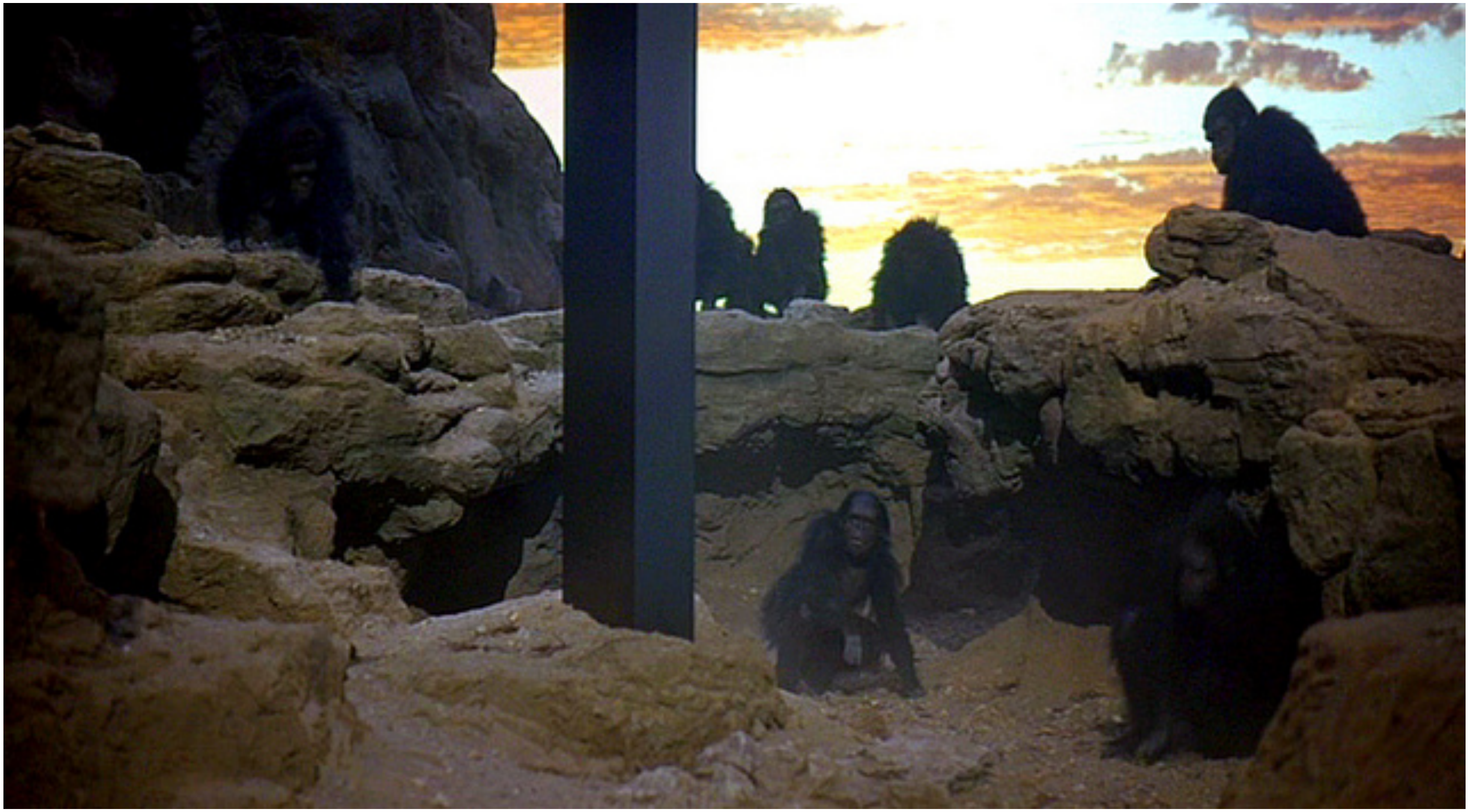
Programming Languages

First Class Functions

Material adapted from Dan Grossman's PL class, U. Washington


THE DAWN OF ~~MAN~~
Functions

Today's lecture will take your programming skills from this…



…to this!

An Example

- What if we wanted to add up all the numbers from a to b?

```
(define (sum a b)
  (if (> a b)
    0
    (+ a
       (sum (+ a 1) b))))
```

$$\sum_{i=a}^{b} i$$

An Example

- What if we wanted to add up the sum of the **squares** of the numbers from a to b:

```
(define (sum-squares a b)
  (if (> a b)
    0
    (+ (expt a 2)
       (sum-squares (+ a 1) b))))
```

$$\sum_{i=a}^{b} i^2$$

An Example

- What if we wanted to add up the sum of the **square roots** of the numbers from a to b:

```
(define (sum-square-roots a b)
  (if (= a b)
    0
    (+ (sqrt a)
       (sum-square-roots (+ a 1) b))))
```

$$\sum_{i=a}^{b} \sqrt{i}$$

These functions are all very similar

- All three of these functions differ only in how the sequence of integers from a to b are transformed before they are all added together.
- The adding process itself is identical in all of the functions:

```
(define (sum-something a b)
  (if (> a b)
    0
    (+ (do something to a)
       (sum-something (+ a 1) b))))
```

- What if there were a general sum function that could sum up any sequence of this form?

A function that takes a function

- Here's a general purpose sum function that takes an argument, called func, that will be applied to each element in the sequence from a to b before the elements are summed:

```
(define (sum-any func a b)
  (if (> a b)
    0
    (+ (func a)
       (sum-any func (+ a 1) b))))
```

Sum-any in action!

```
(sum-any sqrt 1 10)
  => sqrt(1) + sqrt(2) + sqrt(3) + …
  => about 22.5

(define (square x) (* x x))
(sum-any square 1 4)
  => 1^2 + 2^2 + 3^2 + 4^2 => 1 + 4 + 9 + 16 => 30

(define (identity x) x)
(sum-any identity 1 4)
  => 10
```

How to use sum-any

- You can put the name of any function in place of **`sqrt`**, **`square`**, or **`identity`**, and **`sum-any`** will compute

  f(a) + f(a + 1) + f(a + 2) + … + f(b)

  - Provided **`f`** is a function of a single numeric argument.

- What if you want to compute f(a^2/2) + f((a+1)^2/2) + …
  - Fine to do:

```
(define (silly-function x) (/ (* x x) 2))
(sum-any silly-function 1 10)
```

- But this is better:

```
(sum-any (lambda (x) (/ (* x x) 2)) 1 10)
```

- Recall that lambda creates an anonymous function:
  - `(lambda (arg1 arg2…) expression)`

```
(define (sum-any func a b) . . . )

(sum-any square 1 10)
(sum-any sqrt 3 5)
(sum-any identity -8 80)
(sum-any (lambda (x) (/ (* x x) 2)) 1 10)
```

Using anonymous functions

- Most common use: Argument to a higher-order function
  - Don’t need a name just to pass a function
- But: Cannot use an anonymous function for a recursive function
  - Because there is no name for making recursive calls

```
(define (triple x) (* 3 x); named version

(lambda (x) (* 3 x))       ; anonymous version
```

Named functions vs anonymous functions

- Named functions are mostly indistinguishable from anonymous functions.
- In fact, naming a function with **`define`** uses the anonymous form behind the scenes:

```
(define (func arg1 arg2 …) expression)
```

is converted to:

```
(define func (lambda (arg1 arg2 …) expression))
```

- It is poor style to define unnecessary functions in the global (top-level) environment
  - Use either nested defines, or anonymous functions.

Higher-order functions

- A higher-order function is a function that either takes a function (or more than one function) as an argument, or returns a function as a return value.
- Possible because functions are first-class values (or first-class citizens), meaning we can use a function wherever we use a value.
    - Arguments, results of functions, elements of lists, bound to variables, etc
- Most common use is as an argument / result of another function

Higher-order functions

- Let's see another:

```
(define (do-n-times func n x)
  (if (= n 0) x
    (do-n-times func (- n 1) (func x))))
```

- This function computes f(f(f…(x))), where the number of applications of f is n.

Some uses for do-n-times

- Get-nth:
  - `(define (get-nth lst n)`
    `(car (do-n-times cdr n lst)))`
- Exponentiation:
  - `(define (power x y) ; raise x to the y power`
    `(do-n-times (lambda (a) (* x a)) y 1))`
- Note how in the exponentiation example, the anonymous function uses variable x from the outer environment.
  - Couldn't do that without being able to nest functions.
- Note how do-n-times can work with any data type (e.g., lists, numbers…)

A style point

Compare:

```
(if x #t #f)
```

With:

```
(lambda (x) (f x)
```

So don’t do this:

```
(do-n-times (lambda (x) (cdr x)) 3 '(2 4 6 8))
```

When you can do this:

```
(do-n-times cdr 3 '(2 4 6 8))
```

What does this function do?

```
(define (mystery lst)
  (if (null? lst) '()
    (cons (car lst) (mystery (cdr lst)))))
```

Map

```
(define (map func lst)
  (if (null? lst) '()
    (cons (func (car lst)) (map func (cdr lst)))))
```

Map is, without doubt, in the higher-order function hall-of-fame

- The name is standard (same in most prog languages)
- You use it *all the time* once you know it: saves a little space, but more importantly, *communicates what you are doing*
- Built into Racket, so you don't have to include this definition in programs that use map.

Filter

```
(define (filter func lst)
  (cond ((null? lst) '())
        ((func (car lst))
            (cons (car lst) (filter func (cdr lst))))
        (#t
            (filter func (cdr lst)))))
```

Filter is also in the hall-of-fame

- So use it whenever your computation is a filter