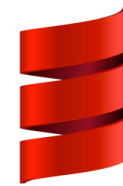
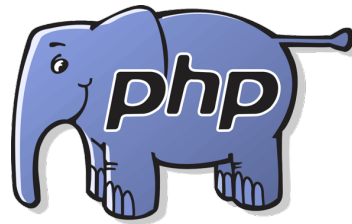


CS 360

Programming Languages

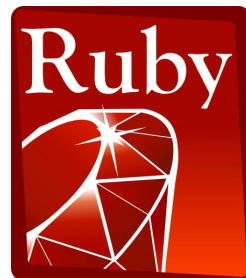
Day 8



Scala



Swift



Review

- A first-class citizen is a data type that can be
 - Passed as an argument to a function.
 - Returned as a value from a function.
 - Assigned to a variable.
 - (Stored in a data structure.)
 - (Created at run-time [dynamically, on-the-fly])
- First three are always part of the definition; last two sometimes.

Review

- Lambda expression: creates and returns an ***anonymous function***.

`(lambda (arg1 arg2 ...) expression)`

Term comes from the ***lambda calculus***.

- Developed by Alonzo Church.
- A formal way of studying the properties of computation, like Turing machines.



Review

- Higher order functions:
 - Take functions as arguments, or
 - Return functions.
- Map and filter both take functions as arguments.
 - Map: Applies a function to every (top-level) item within a list.
`(map func lst)`
 - Filter: Takes a list L and a predicate P; returns a list of all the values in L that satisfy P.
`(filter pred lst)`

Map examples

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

```
(define (double x) (* x 2))
(map double '(1 2 3)) => '(2 4 6)
```

```
(map (lambda (x) (* x 2)) '(1 2 3)) => '(2 4 6)
```

```
(map car '((1 2 3) (4 5) (6) (7 8 9))) => '(1 4 6 7)
```

```
(define (scale factor lst)
  (map (lambda (x) (* x factor)) lst))
```

```
(scale 2 '(1 2 3)) => '(2 4 6)
```

Map examples

```
(map (lambda (x) (+ x 1)) '(1 2 3)) => ?
```

```
(map (lambda (x) (cons x ' ())) '(1 2 3)) => ?
```

```
(map (lambda (x) (list x)) '(1 2 3)) => ?
```

```
(map (lambda (x)  
      (if (> x 0) (* x 2) (* x 3))) '(1 -2 -3 4))
```

Key to using map with lambda expression: the argument to the lambda expression (x) represents each element of the list in turn.

Filter examples

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

```
(filter odd? '(1 2 3)) => '(1 3)
```

```
(define (keep-odds lst)
  (filter odd? lst))
```

```
(filter (lambda (x) (> x 0)) '(-1 2 -3 4)) => '(2 4)
```

Filter examples

```
(filter (lambda (x) (= 1 (remainder x 2))) '(1 2 3)) => ?
```

```
(define (keep-divisible factor lst)
  (filter _____ lst))
```

```
(filter (lambda (lst) (even? (car lst)))
        '((1 2 3) (4 5) (6 7))) => ?
```

```
(filter (lambda (lst) (even? (car lst)))
        '((1 2 3) (4 5) (6 7) ())) => ?
```

```
(define (keep-longer-than n lst)
  (filter _____ lst))
```






- Recall that Racket has a `expt` function:
 - `(expt x y)` => x raised to the y power
- We can define a square function like this:
`(define (square x) (expt x 2))`
- Or a cube function like this:
`(define (cube x) (expt x 3))`
- But this gets rather repetitive.
- What if we wanted to create a lot of these "raise to a power" functions?

Functions that return functions!

```
(define (to-the-power exponent)  
  (lambda (x) (expt x exponent)))
```



Functions that return functions!

```
(define (to-the-power exponent)  
  (lambda (x) (expt x exponent)))
```

Define a function called to-the-power that takes a variable called exponent...

...that returns an anonymous function of a single variable x...

...that raises x to the power of the exponent variable.

How to use this

- Old way:
 - `(define (square x) (expt x 2))`
 - `(define (cube x) (expt x 3))`
- New way:
 - `(define square (to-the-power 2))`
 - `(define cube (to-the-power 3))`
- Notice that the new way doesn't use extra parentheses around the name of the function
 - Don't need 'em: what would we do with the argument?

Another example

- `(define (add3 num) (+ 3 num))`
- `(define (add17 num) (+ 17 num))`
- New way:
 - `(define (create-add-function inc)`
 `(lambda (num) (+ inc num)))`
 - `(define add3 (create-add-function 3))`
 - `(define add17 (create-add-function 17))`

Getting more complicated

- How about a function that takes functions as arguments and returns a new function?
- ```
(define (compose f g)
 (lambda (x) (f (g x))))
```
- ```
(define second (compose car cdr))
```
- ```
(define third (compose car
 (compose cdr cdr)))
```
- ```
(map third ' ((2013 5 6) (2012 1 8)
              (2000 7 7)))
```


Transformations on functions

- Imagine you have a function that must take a non-empty list argument:
- ```
(define (make-safe func)
 (lambda (lst)
 (if (or (not (list? lst))
 (null? lst))
 "No can do!"
 (func lst))))
```

## *More families of functions*

```
(define (divisible n)
 (lambda (x) (= 0 (remainder x n))))

(define (make-quad-polynomial a b c)
 (lambda (x)
 (+ (* a x x) (* b x) c)))
```

# *A little syntax*

- How to call a function:
  - $(f\ e1\ e2\ e3\dots)$
  - $f$  is a function name and  $e1, e2\dots$  are expressions that will be evaluated and passed as the values of the arguments to  $f$ .
- Turns out  $f$  doesn't have to be a function name.
- $f$  can be any expression that evaluates to a function!

# *A little syntax*

- All of these evaluate to a function:
  - the name of a function (e.g., cons, car, +, ...)
  - a lambda expression
  - a function call that returns a function



One more abstraction. Compare:

```
(define (length lst)
 (if (null? lst) 0
 (+ 1 (length (cdr lst)))))
```

```
(define (sum-list lst)
 (if (null? lst) 0
 (+ (car lst) (sum-list (cdr lst)))))
```

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

One more abstraction. Compare:

```
(define (length lst)
 (if (null? lst) 0
 (+ 1 (length (cdr lst)))))
```

```
(define (sum-list lst)
 (if (null? lst) 0
 (+ (car lst) (sum-list (cdr lst)))))
```

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

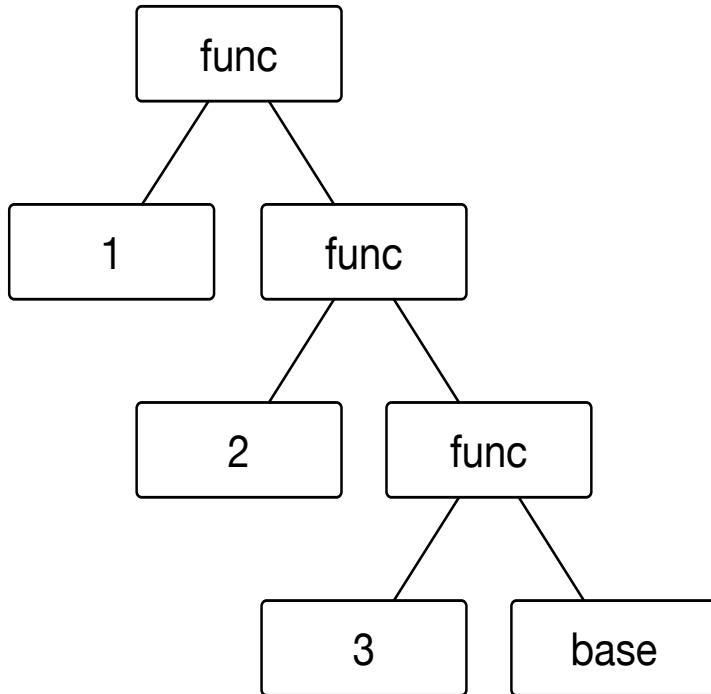
## *One function to rule them all*

```
(define (foldr func base lst)
 (if (null? lst) base
 (func (car lst)
 (foldr func base (cdr lst)))))
```



# (foldr func base lst)

Say `lst = '(1 2 3)`



- **Foldr applies `func` repeatedly to pairs of items, starting from the right end of the list.**
- **The first two items are the last item in the list and the base element.**
- **The function must be a function of two items.**  
$$(f\ 1\ (f\ 2\ (f\ 3\ base)))$$
- **In general, for `lst = (x1 x2 ... xn)`**
- **$(f\ x1\ (f\ x2\ (f\ x3\ (f\ \dots\ (f\ xn\ base))))\dots)$**



```
(define (sum-list-new lst)
 (foldr + 0 lst))
```

```
(define (length-new lst)
 (foldr
 (lambda (elt cdr-len) (+ 1 cdr-len))
 0 lst))
```

```
(define (my-map func lst)
 (foldr
 (lambda (car cdr) (cons (func car) cdr))
 '() lst))
```