Programming Languages

First Class Functions, continued

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 def'n; last two sometimes.

Review

- Lambda expression: on-the-fly function creation!

  ```
  (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: Takes a list **`(v1 v2 …)`** and a function **`f`**; returns a list of **`((f v1) (f v2) …)`**
  - Filter: Takes a list L and a predicate P; returns a list of all the values in L that satisfy P.

- 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

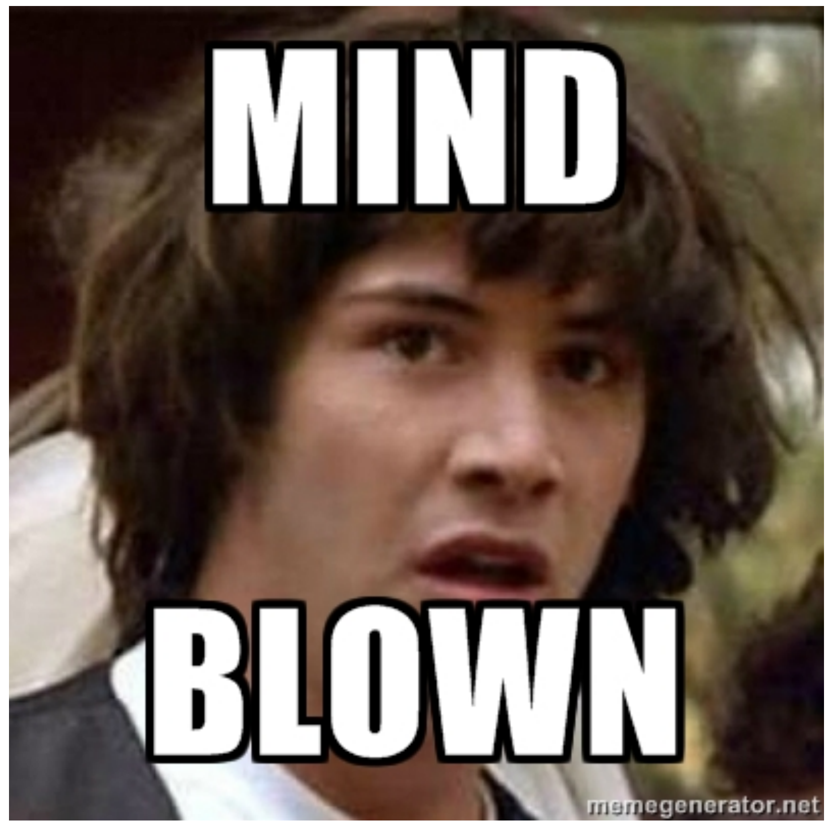
```
(define (divisible n)
  (lambda (x) (= 0 (remainder x n))))

(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…)`**
  - **`f`** is a function name and **`e1`**, **`e2`**… 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)
    (+ (cdr lst) (sum-list (cdr lst)))))

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

One more abstraction. Compare:

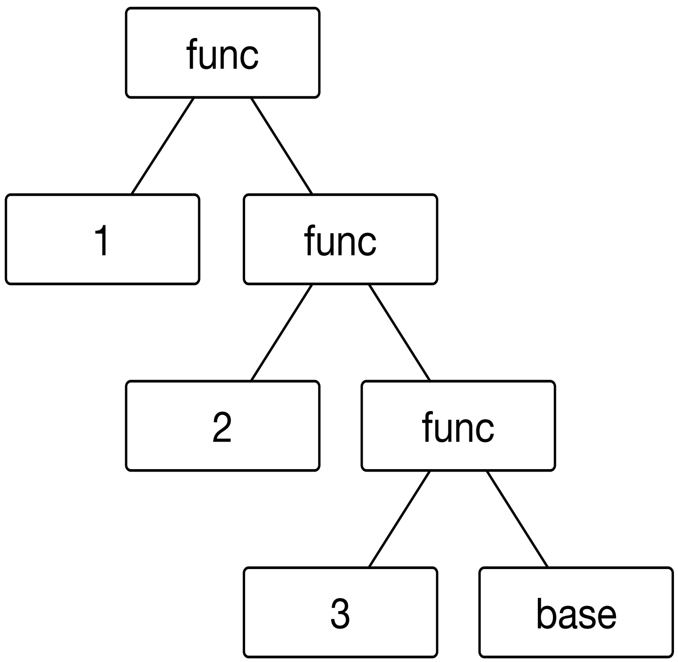
```
(define (length lst)
  (if (null? lst) 0
    (+ 1 (length (cdr lst)))))

(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 … (f xn base)))…)`**

```scheme
(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))
```