First-class and Higher-order Functions

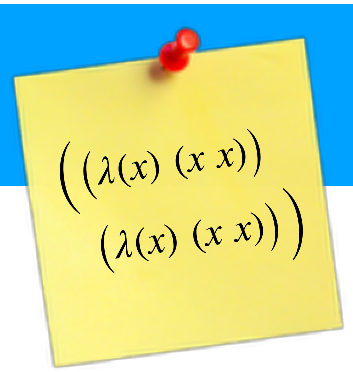
CIS 352 — Spring 2020

Example

Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      ‘()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Example

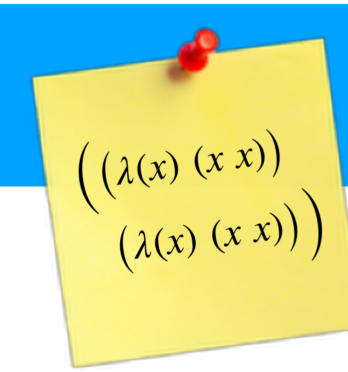


Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      '()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Defines **base case**

Example

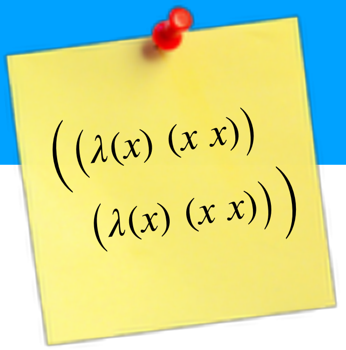


Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      ‘()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Recursive case first computes the square of (car lst)

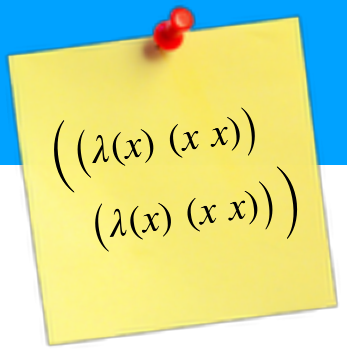
Example



Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      '()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Recursive case next recurs on the list's tail (cdr lst)



Squaring every element of a list

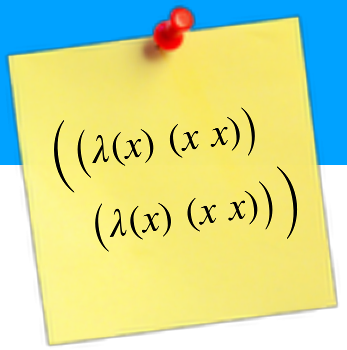
```
(define (square-list-values lst)
  (if (null? lst)
      ‘()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Recursive case finally extends the *new* tail list

Anonymous Functions

- Like many languages (e.g., JS, Python, Ruby), Racket allows anonymous functions to be defined and treated as **values**.
  - `(lambda (args …) body)` ; returns a function as a value
  - E.g., `(lambda (x) (* x x))` ; returns a square function
  - When a language permits functions to be treated as any other value may be treated (passed to other functions, bound to variables, stored in a list, etc), such functions are called **first-class** functions.
- Actually, all functions are anonymous—these are not special.
  - `(define (id x) x)` == `(define id (lambda (x) x))`

Example



Squaring every element of a list

```
(define (map f lst)
  (if (null? lst)
      ‘()
      (cons (f (car lst))
            (map f (cdr lst)))))


(define (square-list-values lst)
  (map (lambda (x) (* x x)) lst))
```

Example

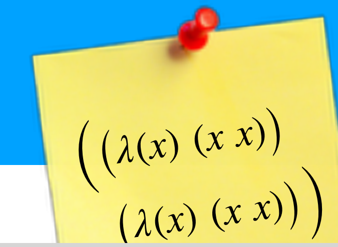
Squaring every element of a list

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

`map` takes a (unary) function and list

```
(define (square-list-values lst)
  (map (lambda (x) (* x x)) lst))
```

Example



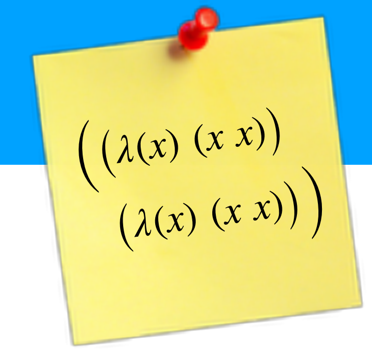
Works essentially as **`square-list-values`** except each new value is **`(f (car lst))`**

Squaring every elem

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

```
(define (square-list-values lst)
  (map (lambda (x) (* x x)) lst))
```

Example



Squaring every element of a list

Now we may define
`square-list-values`
(in one line) in terms of our
(highly-reusable) **`map`** component

```
(define (square-list-values lst)
  (map (lambda (x) (* x x)) lst))
```

Exercise

What is the return value of the following expression?

```
(let ([f (lambda (a) (* a a a))])
  (let ([g add1])
    (let ([h f])
      (g (h 5)))))
```

Exercise

What is the return value of the following expression?

```
(let ([f (lambda (a) (* a a a))])
  (let ([g add1])
    (let ([h f])
      (g (h 5)))))
```

Answer: 126

Exercise

What is the return value of the following expression?

```
(let ([tw (lambda (f x) (f (f x)))])
  (let ([th (lambda (f x) (f (f (f x))))])
    (let ([f add1])
      (tw (lambda (x) (th add1 x)) 0))))
```

Exercise

What is the return value of the following expression?

```
(let ([tw (lambda (f x) (f (f x)))])
  (let ([th (lambda (f x) (f (f (f x))))])
    (let ([f add1])
      (tw (lambda (x) (th add1 x)) 0))))
```

Answer: 6

Higher-order functions

- Languages with first-class functions also have higher-order (HO) functions and are called **higher-order languages** (HOL).
  - A **higher-order function** is a function over functions: a function that takes a function as input, returns a function as output, or both.
- Common higher-order functions include `map`, `foldl`, `foldr`, `filter`, `andmap`, `ormap`, etc…
  - `foldl/foldr` walks a list and uses a function to reduce it
  - `map` walks a list to turn every x into (f x) for parameter f
  - `andmap/ormap` lift a predicate (param) to a list predicat

Exercise

Write an implementation of andmap, such that:

```
> (andmap list? ‘((1 2) () (3)))
#t
> (andmap list? ‘((1 . 2) ()))
#f
> (andmap list? ‘(1 2 3))
#f
```

Exercise

Double-check: does your implementation ***short-circuit***? What does your implementation give for:

```
> (andmap list? '())
```

Exercise

Answer:

```
(define andmap
  (lambda (p? lst)
    (if (null? lst)
        #t
        (and (p? (car lst))
             (andmap p? (cdr lst))))))
```

A predicate p? ***trivially*** holds for all elements of '()

Exercise

Answer:

```
(define andmap
  (lambda (p? lst)
    (if (null? lst)
        #t
        (and (p? (car lst))
             (andmap p? (cdr lst))))))
```

This short-circuits because (and …) does!

Exercise

Another definition, without using (and …):

```
(define (andmap p? lst)
  (if (null? lst)
      #t
      (if (p? (car lst))
          (andmap p? (cdr lst))
          #f)))
```

Use an if to check the next element. If the test fails, short-circuit and return #f, otherwise recur.

Exercise

Yet another definition, using a fold:

```
(define (andmap p? lst)
  (foldl (λ (elem b)
            (and b (p? elem)))
         #t
         lst))
```

fold over the list, accumulating a single boolean:
at each step, conjoin this bool with (p? elem)

Exercise

Write an implementation of map, using a fold:

```
> (map add1 ‘(1 2 3))
‘(2 3 4)
```

Exercise

Answer:

```
(define (map f lst)
    (foldr (lambda (x tail)
             (cons (f x) tail))
           '()
           lst))
```

Fold over the list from right-to-left, accumulating an updated tail of the list, replacing each x with (f x)

Free variables

- A variable x is called **free** in expression e, if there exists a reference to x within e whose definition is not also within e.
    - E.g., **`x`** is free in **`(let ([y 3]) (+ x y))`**, but y is not.
    - E.g., **`x`** is free in **`(list x y z)`**; so are **`y`**, **`z`**, and **list**!
    - Expressions with no free variables are valid programs!
- A function with no free variables is called a **combinator**.
    - E.g., **`(lambda (x) (* x x))`** or **`(λ (f x) (f (f x)))`**
    - Combinators are stand-alone, reusable components
- Functions with free variables, *save* their values! *(More soon)*

Exercise

What are the free variables of the high-lit expression?

```
(let ([f (lambda (x)
           (lambda (y)
             (+ x y)))])
  (let ([g (f 2)])
    (g 3)))
```

Exercise

What are the free variables of the high-lit expression?

```
(let ([f (lambda (x)
           (lambda (y)
             (+ x y)))])
  (let ([g (f 2)])
    (g 3)))
```

Answer: { f }

Exercise

What are the free variables of the high-lit expression?

```
(let ([h (λ (x) (+ 3 x))])
  (let ([g (λ (x y) (* x y y))])
    (h (g 3 4))))
```

Exercise

What are the free variables of the high-lit expression?

```
(let ([h (λ (x) (+ 3 x))])
  (let ([g (λ (x y) (* x y y))])
    (h (g 3 4))))
```

Answer: { h, * }

Exercise

What are the free variables of the high-lit expression?

```
(lambda (x)
  (lambda (y)
    ((lambda (x z) (- x z)) x y)))
```

Exercise

What are the free variables of the high-lit expression?

```
(lambda (x)
  (lambda (y)
    ((lambda (x z) (- x z)) x y)))
```

Answer: { x, y, - }

Currying

- Using higher-order functions, it is always possible to encode a k-ary function as a set of unary functions via **currying**:
  - Invented by Frege; popularized by Schönfinkel, Curry
  - A function **`(define twice (λ (f x) (f (f x))))`** is curried as two nested functions:
    `(define twice (λ (f) (λ (x) (f (f x)))))`
    and to apply the function we call it twice
    `((twice add1) 0)`
  - The first call binds **`f`** to **`add1`** and returns a function that ***saves / remembers*** this value for **`f`**.
  - The second call binds **`x`** and returns **`(f (f x))`**

Exercise

Define a curried version of the slope function:

```
(define (slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

```
> (slope 1 1 5 9)
2
```

Exercise

Answer:

```
(define (slope x0)
  (lambda (y0)
    (lambda (x1)
      (lambda (y1)
        (/ (- y1 y0) (- x1 x0))))))
```

```
> ((((slope 1) 1) 5) 9)
2
```