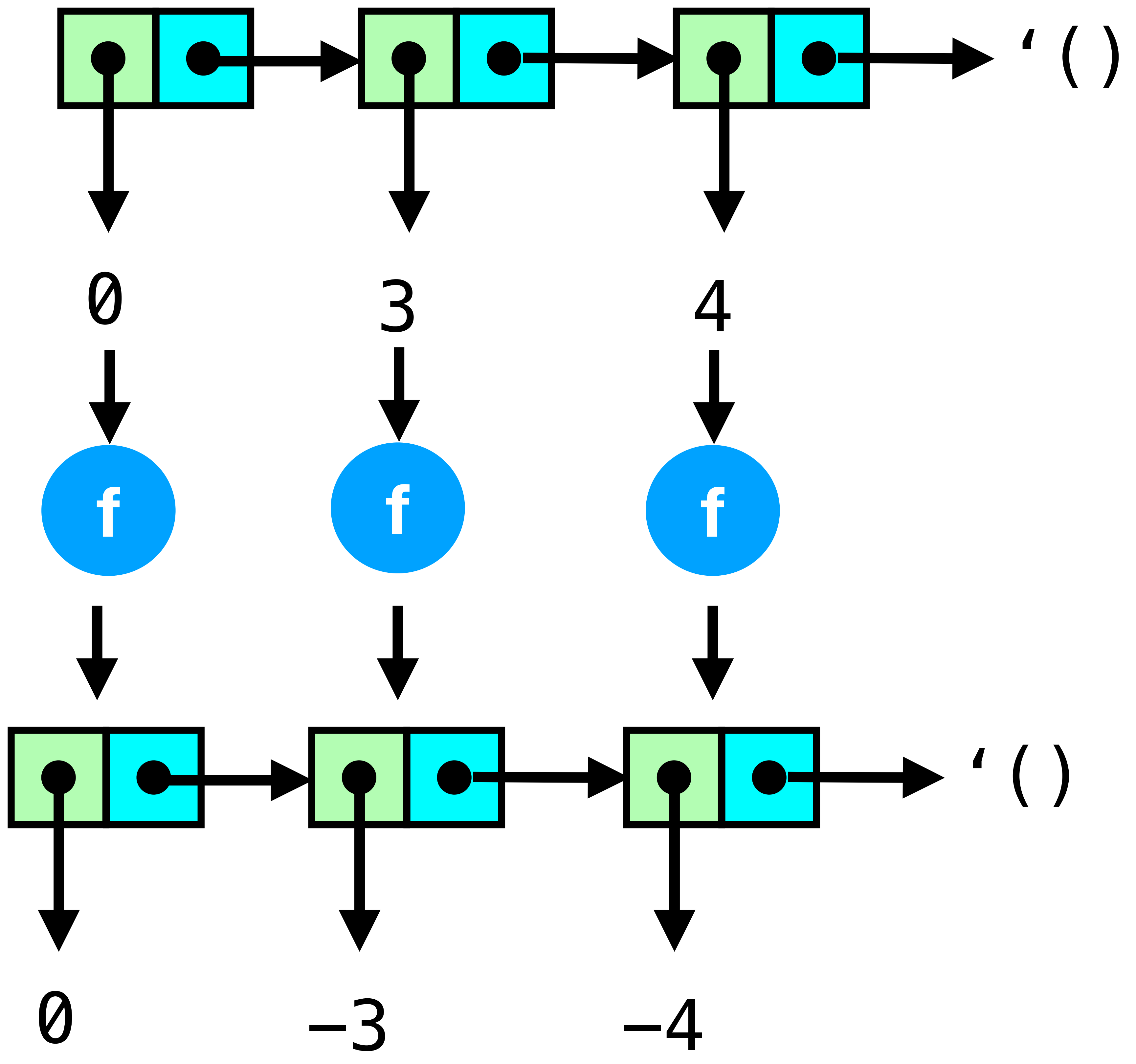
Mapping over Lists

CIS352 — Fall 2022

Kris Micinski

In today's class we will talk about a common pattern: **mapping over a list**

Mapping over a list *transforms* each element by applying a function to it



When does this happen?

- Input and output must both be lists
- Elements mapped “uniformly” (i.e., same function applies to each element)
- Structure of list (length) is maintained

Which one of the below functions has these properties?

```
def invert(l):
    res = []
    for item in l:
        res.append(-item)
    return res
```

```
def sum(l):
    res = 0
    for item in l:
        res += item
    return res
```

When does this happen?

- Input and output must both be lists
- Elements mapped “uniformly” (i.e., same function applies to each element)
- Structure of list (length) is maintained

Which one of the below functions has these properties?

```
def invert(l):
    res = []
    for item in l:
        res.append(-item)
    return res
```

(This one does)

```
def sum(l):
    res = 0
    for item in l:
        res += item
    return res
```

(This one doesn’t; return value is a number!)

Writing map

```
;; map the function f over each element of lst
(define (map f lst)
  (if (empty? lst)
      '()
      (cons (f (first lst)) (map f (rest lst)))))
```

```
def map(f,l):
    res = []
    for item in l:
        res.append(f(item))
    return res
```

Calling maps

```
(map - ‘(1 2 3)) ;; ‘(-1 -2 -3)
;; equivalent to (via “η-extenstionality”)
(map (lambda (x) (- x)) ‘(1 2 3))

(define (foo x y l)
  (map (lambda (z) (* x y z)) l))
```

Why learn map?

- Basic functional idiom: lists are common
  - Good motivator for lambda notation
- When can we use it?
  - Any time we change each element of a list independently
  - We will soon learn a more general pattern—folds—which allows defining *accumulators* over lists

Quasiquoting and Pattern Matching

CIS352 — Fall 2022

Kris Micinski

- Racket **quasi-quotes** build S-expressions nicely
- `` `(,x y 3) `` is equivalent to `(list x ‘y 3)`
  - I.e., Racket splices in values that are unquoted via `,`
  - (quasiquote …), or `…, substitutes any sub-expr `,e` with the return value of `e` within the quoted s-expression

- Works multiple list “levels” deep:
  - `` `(square (point ,x0 ,y0) (point ,x1 ,y1)) ``
- Can unquote arbitrary expressions, not just references:
  - `` `(point ,(+ 1 x0) ,(- 1 y0)) ``

Exercise

Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list ‘point x y))

(define (mk-square pt0 pt1)
  (list ‘square pt0 pt1))
```

Exercise

Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list ‘point x y))
```

```
(define (mk-square pt0 pt1)
  (list ‘square pt0 pt1))
```

```
(define (mk-point x y)
  `(point ,x ,y))
```

```
(define (mk-square pt0 pt1)
  `(square ,pt0 ,pt1))
```

- Racket also has **pattern matching**
    - `(match e [pat₀ body₀] [pat₁ body₁]…)`
- Evaluates e and then checks each **pattern**, in order
- Pattern can bind variables, body can use pattern variables

- Many patterns (check docs to learn various useful forms)
- Patterns checked in order, first matching body is executed
  - Later bodies won't be executed, **even if they also match**!
    - **Students make frequent mistakes on this!**
- E.g.,

```
(match '(1 2 3)
       [`(,a ,b) b]
       [`(,a . ,b) b])  ; returns '(2 3)
```

Matching a literal

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

(binds n)

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches when e evaluates to some number?

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
   (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Never matches!

Subsumed by previous case!

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches a cons cell, binds x and y

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches a list of length three

Binds first element as `a0`, second as `a1`, etc…

Called a “quasi-pattern”

Can also test predicates on bound vars:

```
`(,(? nonnegative-integer? x) ,(? positive? y))
```

```
(match e
  ['hello 'goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)]
  [_ 23])
```

Can also have a **default case** written via **wildcard** _

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

```
(define (foo x)
  (match x
    [(? …) …]
    …))
```

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

Answer (one of many)

Observe how quasipatterns and quasiquotes interact

```
(define (foo x)
  (match x
    [(? number? n) (* n 2)]
    [`(,a ,b ,_) `(,a ,b)]
    [_ "error"]))
```

- Using pattern matching, we can build **type predicates**
  - Predicates that specify data formats
- We will **frequently** use these in-lieu of static typing

```
(define (tree? t)
  (match t
    ['empty #t]
    [`(leaf ,v) #t]
    [`(binary ,(? tree?) ,(? tree?)) #t]
    ;; don't forget this!
    [_ #f]))
```

- We can use **define/contract** to specify dynamically-checked **contracts** on functions

```
(define/contract (tree-min t0)
  (-> tree? any/c)
  (match t
    ['empty (error "no min of empty tree")]
    [`(leaf ,v) v]
    [`(binary ,t0 ,t1) (tree-min t0)]))

> (tree-min '(binary (leaf 2) empty))
2
```

```
> (tree-min '(binary 2 empty))
. . tree-min: contract violation
  expected: tree?
  given: '(binary 2 empty)
  in: the 1st argument of
      (-> tree? any/c)
  contract from: (function tree-min)
  blaming: anonymous-module
   (assuming the contract is correct)
```

Squaring every element of a list

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

Defines **base case**

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)

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

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))
```

`map` takes a (unary) function and list

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))
```

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

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

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

We can write the def of map in just one line!

```
(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))
```

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

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

```
> (andmap list? ‘())
```

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