

# CS450

## Structure of Higher Level Languages

Lecture 07: foldr, looping first-to-last

Tiago Cogumbreiro

# Exercises on lists

# Searching

# Element in list?

Spec

```
(require rackunit)
(check-true (member? "d" (list "a" "b" "c" "d")))
(check-false (member? "f" (list "a" "b" "c" "d")))
```

# Element in list?

## Spec

```
(require rackunit)
(check-true (member? "d" (list "a" "b" "c" "d")))
(check-false (member? "f" (list "a" "b" "c" "d")))
```

## Solution

```
(define (member? x l)
  (match l
    [(list) #f]
    [(list h _ ...) #:when (equal? x h) #t]
    [(list _ l ...) (member? x l)]))
```

## Solution in Python

```
def member(x, l):
    for h in l:
        if h == x:
            return True
    return False
```

# Prefix in list?

## Spec

```
(require rackunit)
(check-true (string-prefix? "Racket" "R")) ; available in standard library
(check-true (match-prefix? "R" (list "foo" "Racket")))
(check-false (match-prefix? "R" (list "foo" "bar")))
```

# Prefix in list?

## Spec

```
(require rackunit)
(check-true (string-prefix? "Racket" "R")) ; available in standard library
(check-true (match-prefix? "R" (list "foo" "Racket")))
(check-false (match-prefix? "R" (list "foo" "bar")))
```

## Solution

```
(define (match-prefix? p l)
  (match l
    [(list) #f]
    [(list h _ ...) #:when (string-prefix? h p) #t]
    [(list _ _ 1 ...) (match-prefix? p l)]))
```

# Can we generalize the search algorithm?

```
; Example 1  
(define (member? x l)  
  (match l  
    [(list) #f]  
    [(list h _ ...) #:when (equal? h x) #t]  
    [(list _ 1 ...) (member? x l)]))
```

```
; Example 2  
(define (match-prefix? p l)  
  (match l  
    [(list) #f]  
    [(list h _ ...) #:when (string-prefix? h p) #t]  
    [(list _ 1 ...) (match-prefix? p l)]))
```



# Can we generalize the search algorithm?

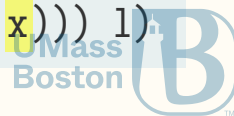
```
; Example 1
(define (member? x l)
  (match l
    [(list) #f]
    [(list h _ ...) #:when (equal? h x) #t]
    [(list _ 1 ...) (member? x l)]))
```

```
; Example 2
(define (match-prefix? p l)
  (match l
    [(list) #f]
    [(list h _ ...) #:when (string-prefix? h p) #t]
    [(list _ 1 ...) (match-prefix? p l)]))
```

## Solution

```
(define (exists? found? l)
  (match l
    [(list) #f]
    [(list h _ ...) #:when (found? h) #t]
    [(list _ 1 ...) (exists? found? l)]))
```

```
; Example 1
(define (member? x l)
  (exists?
    (lambda (y) (equal? x y)) l))
; Example 2
(define (match-prefix? x l)
  (exists?
    (lambda (y) (string-prefix? y x)) l))
```



# Removing elements from list

# Remove zeros from a list

Spec

```
(require rackunit)
(check-equal? (list 1 3 4) (remove-0s (list 0 1 3 0 4)))
(check-equal? (list 1 2 3) (remove-0s (list 1 2 3)))
```

# Remove zeros from a list

## Spec

```
(require rackunit)
(check-equal? (list 1 3 4) (remove-0s (list 0 1 3 0 4)))
(check-equal? (list 1 2 3) (remove-0s (list 1 2 3)))
```

## Solution

```
(define (remove-0s l)
  (match l
    [(list) (list)]
    [(list h l ...) #:when (not (equal? h 0))
     (cons h (remove-0s l))]
    [(list _ l ...) (remove-0s l)]))
```

## Solution in Python

```
def remove_0s(l):
    result = []
    for h in l:
        if h != 0:
            result.append(h)
    return result
```

# Can we generalize this functional pattern?

## Original

```
(define (remove-0s l)
  (match l
    [(list) (list)]
    [(list h l ...) #:when (not (equal? h 0))
     (cons h (remove-0s l))]
    [(list _ l ...) (remove-0s l)]))
```

## Generalized

```
(define (filter keep? l)
  (match l
    [(list) (list)]
    [(list h l ...) #:when (keep? h)
     (cons h (filter keep? l))]
    [(list _ l ...) (filter keep? l)]))

;; Usage example
(define (remove-0s l)
  (filter
   (lambda (x) (not (equal? x 0))) l))
```

Concatenate two lists

# Concatenate two lists

Implement function `(append l1 l2)` that appends two lists together.

Spec

```
(check-equal?  
  (append (list 1 2) (list 3 4))  
  (list 1 2 3 4))
```

# Concatenate two lists

Implement function `(append l1 l2)` that appends two lists together.

Spec

```
(check-equal?
  (append (list 1 2) (list 3 4))
  (list 1 2 3 4))
```

Solution

```
(define (append l1 l2) ; l2 = (list 4 5 6)
  (match l1             ; l1 = (list 1 2 3)
    [(list) l2]
    [(list h l1 ...) ; h = 1  l1 = (list 2 3)
     (define result (append l1 l2)) ; result = (append '(2 3) '(4 5 6))
     ; result = (list 2 3 4 5 6)
     (cons h result)]))
; output = (list 1 2 3 4 5 6)
```





# Generalizing order-preserving loops

# An order-preserving recursion pattern

1. **Case** `(list)` (handle-base)
2. **Case** `(list h l ...)` (handle-step)
3. Recursive call handles "smaller"

## Example 1

```
(define (map f l)
  (match l
    [(list) (list)]
    [(list h l ...)
     (define result (map f l))
     (cons (f h) result)]))
; = (handle-step h result)
```

```
(define (rec l)
  (match l
    [(list) handle-base]
    [(list h l ...)
     (define result (rec l))
     (handle-step h result)]))
```

## Example 2

```
(define (append l1 l2)
  (match l1
    [(list) l2]
    [(list h l1 ...)
     (define result (append l1 l2))
     (cons h result)]))
```

# A note about side-effects

- We need to be mindful when implementing `map` as the order of side-effects may matter.
- The **standard** implementation of `map` invokes `f` from left-to-right.
- Our implementation implementation of `map` invokes `f` from right-to-left.
- The reason we implement r-t-l is to allow for generalization with `foldr`
- In terms of code, to obtain an l-t-r ordering, call `f` before recursing
- **Run** `(map displayln (list 1 2 3))` on either version to observe the difference

Side-effects from right-to-left

```
(define (map f l)
  (match l
    [(list) (list)]
    [(list h l ...)
     (define result (map f l))
     (cons (f h) result))]))
```

Side effects from left-to-right

```
(define (map f l)
  (match l
    [(list) (list)]
    [(list h l ...)
     (define x (f h))
     (define result (map f l))
     (cons x result))]))
```



# An order-preserving recursion pattern

## Searching

```
(define (exists? found? l)
  (match l
    [(list) #f]
    [(list h _ ...) #:when (found? h) #t]
    [(list _ l ...) (exists? found? l)]))
```

## Following the recursion pattern

```
(define (exists? found? l)
  (match l
    [(list) #f]
    [(list h l ...)
     (define result (exists? found? l))
     (or (found? h) result)]))
```

# An order-preserving recursion pattern

## Removing

```
(define (filter keep? l)
  (match l
    [(list) (list)]
    [(list h 1 ...) #:when (keep? h)
     (cons h (filter keep? 1))]
    [(list _ 1 ...) (filter keep? 1)]))
```

## Following the recursion pattern

```
(define (filter keep? l)
  (match l
    [(list) (list)]
    [(list h 1 ...)
     (define result (filter keep? 1))
     (if (keep? h)
         (cons h result) result)]))
```

Implementing this recursion pattern

# Implementing this recursion pattern

## Recursive pattern for lists

```
(define (rec l)
  (match l
    [(list) base-case]
    [(list h l ...)
     (define result (rec l))
     (handle-step h result)]))
```

## Fold right reduction

```
(define (foldr handle-step base-case l)
  (match l
    [(list) base-case]
    [(list h l ...)
     (define result (foldr step base-case l))
     (handle-step h result)]))
```

```
# In Python
def foldr(step, base_case, l):
    result = base_case
    for h in reversed(l):
        result = step(h, result)
    return result
```



# Implementing map with foldr

```
(define (map f l)
  (match l
    [(list) (list)]
    [(list h l ...)
     (define result (map f l))
     (cons (f h) result)]))
```



# Implementing map with foldr

```
(define (map f l)
  (match l
    [(list) (list)]
    [(list h l ...)
     (define result (map f l))
     (cons (f h) result)]))
```

## Solution

```
(define (map f l)
  (foldr
    ; step: how do you build the next result
    (lambda (h result) (cons (f h) result))
    ; what to return when the list is empty
    (list)
    ; iterate/match over l
    l))
```

```
# Python pseudo-code
result = []
for h in reversed(l):
    # result = cons(f(h), result)
    result.append(f(h))
```



# Implementing append with foldr

```
(define (append l1 l2)
  (match l1
    [(list) l2]
    [(list h l1 ...)
     (define result (append l1 l2))
     (cons h result)]))
```

# Implementing append with foldr

```
(define (append l1 l2)
  (match l1
    [(list) l2]
    [(list h l1 ...)
     (define result (append l1 l2))
     (cons h result)]))
```

## Solution

```
(define (append l1 l2)
  (foldr
    ; step: add the element to the list being built
    (lambda (h result) (cons h result))
    ; base-case: start with list l2
    l2
    ; iterate/match over l1
    l1))
```

# Implementing filter with foldr

```
(define (filter keep? l)
  (match l
    [(list) (list)]
    [(list h l ...)
     (define result (filter keep? l))
     (if (keep? h) (cons h result) result)]))
```

# Implementing filter with foldr

```
(define (filter keep? l)
  (match l
    [(list) (list)]
    [(list h l ...)
     (define result (filter keep? l))
     (if (keep? h) (cons h result) result)]))
```

## Solution

```
(define (filter keep? l)
  (foldr
    ; handle-step
    (lambda (h result) (if (keep? h) (cons h result) result))
    ; base-case
    (list)
    ; iterate/match over l
    l))
```

# Implementing exists? with foldr

```
(define (exists? found? l)
  (match l
    [(list) #f]
    [(list h l ...)
     (define result (exists? found? l))
     (or (found? h) result)]))
```

# Implementing exists? with foldr

```
(define (exists? found? l)
  (match l
    [(list) #f]
    [(list h l ...)
     (define result (exists? found? l))
     (or (found? h) result)]))
```

## Solution

```
(define (exists? found? l)
  (foldr
    ; handle-step
    (lambda (h result) (or (found? h) result))
    ; base-case
    #f
    ; iterate over l
    l))
```

