Warmup your Diagnostic Skills.

In the following slides we present 4 versions of the "Allatoms" procedure, designed to take an arbitrary list as input and return a flat list containing the atoms in the initial list.

Each version has a problem, that is corrected in the next. The final version is correct.

```
(define a1
  (lambda (lst)
    (cond ((null? lst) '())
          (= (length lst) 1) lst)
         (else (cons (a1 (car lst))
                     (a1 (cdr lst)))))
  )
)

1 ]=> (a1 '(b c))
;Value 1: ((b c))

1 ]=> (a1 '(a (b c) d))
;The object b, passed as the first argument to length, is not the correct type.
```
Allatoms: version 2

(define a2
  (lambda (lst)
    (cond ((null? lst) '())
          (= (length lst) 1) lst
          (else (append (if (pair? (car lst))
                          (a2 (car lst))
                          (list (car lst)))
                      (a2 (cdr lst)))))
  )
)

1 ]=> (a2 '(a (b c) d))
;Value 4: (a b c d)

1 ]=> (a2 '((a () c) ((d)) (e (f (g))) h))
;Value 5: (a () c (d) (e (f (g))) h)

1 ]=> (a2 '((b c)) )
;Value 6: ((b c))

Allatoms: version 3

(define a3
  (lambda (lst)
    (cond ((null? lst) '())
          (else (append (if (pair? (car lst))
                          (a3 (car lst))
                          (list (car lst)))
                      (a3 (cdr lst))))
  )
)

1 ]=> (a3 '((b c)) )
;Value 7: (b c)

1 ]=> (a3 '((a (b) c) d) )
;Value 8: (a b c d)

1 ]=> (a3 '((a () c) ((d)) (e (f (g))) h))
;Value 9: (a () c d e f g h)
(define a4
  (lambda (lst)
    (cond ( (null? lst) '( )
             ( (pair? lst) (append (a4 (car lst)) (a4 (cdr lst)))
             ( else (list lst) )
             )
    )
  )
)

This is simpler, but changes the specification of the procedure:

1 ]=> (a4 '((a () c) ((d)) (e (f (g)) h)))
;Value 10: (a c d e f g h)

1 ]=> (a4 '(a . b))
;Value 11: (a b)

1 ]=> (a4 'a)
;Value 12: (a)

---

**Review from Last Day**

- Lists (cons cells, proper list, creating lists (append, list, cons))
- Testing for Equality (eq?, =, eqv?, equal?)
- Example of car-cdr Recursion (counting atoms ex.)
- Efficiency
  - helper functions
  - local variable binding (let, let*)
- Higher Order Procedures
  - Procedures as input
  - Procedures as returned values
  - Built-in procedure map
  - Built-in procedure eval
  - ..and we pick up from here...
Applying Procedures with apply

1 ]=> (apply + '(1 2 3))
;Value: 6
1 ]=> (apply append '((a) (b)))
;Value 5: (a b)

1 ]=>
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else
          (apply + (map atomcount s)))))

;Value: atomcount
1 ]=> (atomcount '(a (b) c))
;Value: 3

Higher-order Procedures: reduce

(define (reduce op 1 id)
  (if (null? 1)
      id
      (op (car 1)
          (reduce op (cdr 1) id))))

A binary \(\mapsto\) n-ary procedure.

The reduce procedure takes a binary operation and applies it right-associatively to a list of an arbitrary number of arguments.

NOTE: reduce is not equivalent to apply.
Higher-order Procedures: \textit{reduce}

\[(\text{reduce } + \ '(1 \ 2 \ 3) \ 0) \Rightarrow 6:\]
\[(\text{reduce } + \ '(1 \ 2 \ 3) \ 0)\]
\[ (+ \ 1 \ (\text{reduce } + \ '(2 \ 3) \ 0))\]
\[ (+ \ 1 \ (+ \ 2 \ (\text{reduce } + \ '(3) \ 0)))\]
\[ (+ \ 1 \ (+ \ 2 \ (+ \ 3 \ (\text{reduce } + \ '(0))))\]
\[ (+ \ 1 \ (+ \ 2 \ (+ \ 3 \ 0)))\]
\[ 6\]
\textbf{Note:} \ (+ \ 1 \ 2 \ 3) \Rightarrow 6

\[(\text{reduce } / \ '(24 \ 6 \ 2) \ 1) \Rightarrow 8:\]
\[(\text{reduce } / \ '(24 \ 6 \ 2) \ 1)\]
\[ (/ \ 24 \ (\text{reduce } / \ '(6 \ 2) \ 1))\]
\[ (/ \ 24 \ (/ \ 6 \ (\text{reduce } / \ '(2) \ 1)))\]
\[ (/ \ 24 \ (/ \ 6 \ (/ \ 2 \ (\text{reduce } / \ '(0))))\]
\[ (/ \ 24 \ (/ \ 6 \ (/ \ 2 \ 1)))\]
\[ 8\]
\textbf{Note:} \ (/ \ 24 \ 6 \ 2) \Rightarrow 2

---

Higher-order Procedures: \textit{reduce}

Given union, which takes two lists representing sets and returns their union:

\[1 \ ]\Rightarrow \ (\text{apply union } '(1 \ 3)(2 \ 3 \ 4))\]
\[;\text{Value 21: (1 2 3 4)}\]

\[1 \ ]\Rightarrow \ (\text{apply union } '(1 \ 3)(2 \ 3 \ 4 \ 5))\]
\[;\text{The procedure #\text{[compound-procedure union]}\]
\[;\text{has been called with 3 arguments;}\]
\[;\text{it requires exactly 2 arguments.}\]

\[1 \ ]\Rightarrow \ (\text{reduce union } '(1 \ 3)(2 \ 3 \ 4 \ 5) \ '())\]
\[;\text{Value 22: (1 2 3 4 5)}\]

\textbf{Question:} How would you have to change reduce to be able to take \textit{intersection} as its function argument?
Example Practice Procedures

- cdrLists: given a list of lists, form new list giving all elements of the cdr's of the sub-lists.
  \((1\ 2)\ (3\ 4\ 5)\ (6)) \Rightarrow (2\ 4\ 5)\)

- swapFirstTwo: given a list, swap the first two elements of the list.
  \((1\ 2\ 3\ 4) \Rightarrow (2\ 1\ 3\ 4)\)

- swapTwoInLists: given a list of lists, form new list of all elements in all lists, with first two of each swapped.
  \(((1\ 2\ 3)\ (4)\ (5\ 6)) \Rightarrow (2\ 1\ 3\ 4\ 6\ 5)\)

- addSums: given a list of numbers, sum the total of all sums from 0 to each number.
  \((1\ 3\ 5) \Rightarrow 22\)

More Practice Procedures

- addToEnd: add an element to the end of a list.
  \((\text{addToEnd}\ 'a'\ (a\ b\ c)) \Rightarrow (a\ b\ c\ a)\)

- revLists: given a list of lists, form new list consisting of all elements of the sublists in reverse order.
  \(((1\ 2)\ (3\ 4\ 5)\ (6)) \Rightarrow (6\ 5\ 4\ 3\ 2\ 1)\)

- revListsAll: given a list of lists, form new list from reversal of elements of each list.
  \(((1\ 2)\ (3\ 4\ 5)\ (6)) \Rightarrow (2\ 1\ 5\ 4\ 3\ 6)\)
Passing procedures: prune

Suppose we want a procedure that will test every element of a list and return a list containing only those that pass the test.

We want it to be very general: it should be able to use any test we might give it. How will we tell it what test to apply?

What should a procedure call look like?
Example: Prune out the elements of myList that are not atoms.

Now let's write the procedure.

; Return a new list containing only the elements of list that pass the test.
; Precondition:

(define prune
  (lambda (test lst)
    (cond ((null? lst) '())
          ((test (car lst))
           (cons (car lst)
                  (prune test (cdr lst))
               )
           )
          (else (prune test (cdr lst))
               )
        )
  )
)

Sample run

1 ]=> (define (atom? x) (not (pair? x)))
;Value: atom?

1 ]=> (prune atom? '((3 1) 4 (x y z) (x) y ()))
;Value 12: (4 y ())

1 ]=> (prune null? '((a b c) (1 2) () () (x (y w) z)))
;Value 13: (() ())

84 85
Write calls to prune that will prune myList in these ways:

• Prune out elements that are null.

• (Assume myList contains lists of integers.) Prune out elements whose minimum is not at least 50.
  Hint: there is a built-in min procedure.

• (Assume myList contains lists.) Prune out elements that themselves have more than 2 elements.

This is becoming tedious. We need to declare a procedure for each possible test we might dream up.

Back to Unnamed Procedures

Exercise: What is the value of each of these Scheme expressions?

( (lambda (x) (cons x ())) 'y )
;

( (lambda (x y) (> (length x) (length y)))
  '(a b c) '(d) )
;

( (lambda (x) (list? x)) '(lambda (x) (list? x)) )
;

( (lambda (x y) (append x y)) '(1 2) '(3 4 5) )
Using unnamed procedures to call `prune`

1 ]=> (define myList
   '(() (a b c) (1 2) () ((x y w z)))
;Value: myList

1 ]=> (prune (lambda (x) (not (null? x))) myList)
;Value 4: ((a b c) (1 2) () (x (y w) z))

1 ]=> (define myList '((59 72 40) (85 70 88 56)))
;Value: myList

1 ]=> (prune (lambda (x) (> (apply min x) 50)) myList)
;Value 5: ((85 70 88 56))

1 ]=> (define myList '((23 34) (10 1 3 4) () (2 3 4)))
;Value: myList

1 ]=> (prune (lambda (x) (<= (length x) 2)) myList)
;Value 6: ((23 34) ()))

Uses of unnamed lambda-expressions

Example: Suppose we have tables of data (represented using Scheme lists), and procedures that can do things like select out the rows of a given table that pass some test.

Suppose we want the user to be able to specify any criterion they might want. Examples:

- Retrieve students where gpa > 3.0
- Retrieve courses where classSize < 100
- Retrieve profs where building = SF

It would be tedious to write a named procedure for every single criterion that the user might specify.

Instead, we can have the `program` construct an appropriate lambda-expression, based on the user’s query.
Passing Procedures: Bubblesort

What we want in the end

Sample run of procedure bubblesort

```
eddie 1% scheme
Scheme Microcode Version ..
1 ]=> (load "sort.scm")
;Loading "sort.scm" -- done
;Value: bubblesort

1 ]=> (bubblesort '(3 4 1 5 0 2 3) <)
;Value 1: (0 1 2 3 3 4 5)

1 ]=> (bubblesort
  '((a b c) (a) (1 2 3 4) () (z z z) (y y))
  (lambda (x y) (< (length x) (length y))) )
;Value 2: (() (a) (y y) (z z z) (a b c) (1 2 3 4))

1 ]=> (trace helper)
;No value

1 ]=> (trace bubbleFirstN)
;No value```
1] => (bubblesort '(3 4 1 5 0 2 3) <)

[Entering #[helper]  Args: (3 4 1 5 0 2 3) #< 6]
[Entering #[bubblefirstn]  Args: (3 4 1 5 0 2 3) #< 6]
...
((3 1 4 0 2 3 5)
  <= #[bubblefirstn]  Args: (3 4 1 5 0 2 3) #< 6]
[Entering #[helper]  Args: (3 1 4 0 2 3 5) #< 5]
[Entering #[bubblefirstn]  Args: (3 1 4 0 2 3 5) #< 5]
...
((1 3 0 2 3 4 5)
  <= #[bubblefirstn]  Args: (3 1 4 0 2 3 5) #< 5]
[Entering #[helper]  Args: (1 3 0 2 3 4 5) #< 4]
[Entering #[bubblefirstn]  Args: (1 3 0 2 3 4 5) #< 4]
...
((0 1 2 3 4 5)
  <= #[bubblefirstn]  Args: (1 3 0 2 3 4 5) #< 4]
[Entering #[helper]  Args: (1 0 2 3 3 4 5) #< 3]
[Entering #[bubblefirstn]  Args: (1 0 2 3 3 4 5) #< 3]
...
((0 1 2 3 3 4 5)
  <= #[bubblefirstn]  Args: (1 0 2 3 3 4 5) #< 3]
[Entering #[helper]  Args: (0 1 2 3 3 4 5) #< 2]
[Entering #[bubblefirstn]  Args: (0 1 2 3 3 4 5) #< 2]
...
### Calling Procedure

; Precondition: smaller? is a procedure that can be applied to any two elements of lst. It should return #t iff the first argument is "smaller" than the second.

(define bubblesort
  (lambda (lst smaller?)
    (helper lst smaller? (- (length lst) 1)))

)

### The Outer Loop

#### Helper procedure - actual outer loop

; Bubblesorts the first n elements of lst. Returns a new list with the first n elements of lst sorted, followed by the rest of lst unchanged.
; Precondition: n < (length lst).

(define helper
  (lambda (lst smaller? n)
    (if (<= n 0)
      lst
      (helper (bubbleFirstN lst smaller? n) smaller? (- n 1))
    )
  )
)
The Inner Loop

; Does a single "bubble run".
; Precondition: n < (length lst)

(define bubbleFirstN
  (lambda (lst smaller? n)
    (cond ((= n 0) lst)
          ((smaller? (car lst) (cadr lst))
           (cons (car lst)
                 (bubbleFirstN (cadr lst)
                               smaller?
                               (- n 1)))
          )
          (else (cons (cadr lst)
                      (bubbleFirstN (cons (car lst)
                                     (cddr lst))
                                   smaller?
                                   (- n 1)))
          )
    )
  )
)

Is our bubblesort procedure \(O(n^2)\), where \(n\) is
the length of the original list, as it should be?