Warm up 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.

Allatoms: version 1

(define a1
  (lambda (lst)
    (cond ((null? lst) '1)
           ((pair? lst) (append a1 (cdr lst)))
           (else (cons (car lst) (a1 (cdr lst))))
    )
  )
1⇒ (a1 '((b c) d)) ;Value 1: '((b c) d)

Allatoms: version 2

(define a2
  (lambda (lst)
    (cond ((null? lst) '1)
           (else (append (if (pair? (car lst))
                            (a2 (cdr lst))
                            (list (car lst)))
                       (a2 (cdr lst)))
    )
  )
1⇒ (a2 '((a (b c) d))
    ;Value 4: '(a (b c) d)

Allatoms: version 3

(define a3
  (lambda (lst)
    (cond ((null? lst) '1)
           (else (append (if (pair? (car lst))
                            (a3 (cdr lst))
                            (list (car lst)))
                       (a3 (cdr lst)))
    )
  )
1⇒ (a3 '((a (b c) d) (e (f g) h))
    ;Value 7: '((a (b c) d) (e (f g) h))

Allatoms: version 4

(define a4
  (lambda (lst)
    (cond ((null? lst) '1)
           ((pair? lst) (append a4 (cdr lst)))
           (else (list lst))
    )
  )
This is simpler, but changes the specification of the procedure:
1⇒ (a4 '((a (c) ((a)) (e (f (g) h))))
    ;Value 10: '(a (c) ((a)) (e (f (g) h)))
1⇒ (a4 '(a ,b))
    ;Value 11: '(a ,b)
1⇒ (a4 'a)
    ;Value 12: 'a

Review from Last Day

- Lists (cons, cdr, car, last), creating lists (cons, map, apply)
- Trees (equal?, +, *, eq?)
- Example of car or cdr recursion (counting atoms)

- Efficiency
  - Recursions
  - local variables limited (let, let*)
- Higher Order Procedures
  - Procedures as inputs
  - 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: 6
1⇒ (apply append '((a) (b)))
    ;Value 5: '(a b)
1⇒ (a3 '((a (c) ((a)) (e (f g) h))))
    ;Value: atomcount
1⇒ (atomcount '(a (b) c))
    ;Value: 3

Higher-order Procedures: reduce

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

A binary → 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: reduce

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

```scheme
1 0 2 3 4 => (apply union ((1 (2 3 3 4)))
;Value 1; (1 2 3 4)
1 0 2 3 4 => (apply union ((1 (2 3 3 4)))
;Value 2; (1 2 3 4 5)

Question: How would you have to change reduce to be able to take intersection as its function argument?
```

Note: (2 4 6 2) => 2

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 the elements of my-list that are not atoms.

Now let's write the procedure

```scheme
(define prune
  (lambda (test lst)
    (cond ((null lst) ()
      (test (car lst))
      (prune test (cdr lst)))
      (else (prune test (cdr lst))))
        )
)
```

Write calls to prune that will prune my-list in these ways:

- Prune out elements that are null.
- (Assume my-list contains lists of integers.) Prune out elements whose minimum is not at least 50. Hint: there is a built in min procedure.
- (Assume my-list 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

Example Practice Procedures

- `cat-lists`: given a list of lists, form new list giving all elements of the car's of the sub-lists,
  ```scheme
  ((1 2) (3 4 5) (6)) => (2 4 5)
  ```
- `swapFirstTwo`: given a list, swap the first two elements of the list,
  ```scheme
  (1 2 3 4) => (2 1 3 4)
  ```
- `swapTwoWitllists`: given a list of lists, form a new list of all elements in all lists, with first two of each swapped,
  ```scheme
  ((1 2 3 4) (5 6)) => (2 3 4 6)
  ```
- `addSum`: given a list of numbers, sum the total of all sums from 0 to each number
  ```scheme
  (1 3 5) => 22
  ```

More Practice Procedures

- `addToEnd`: add an element to the end of a list,
  ```scheme
  (addToEnd 'a (b c)) => (a b c)
  ```
- `rev-lists`: given a list of lists, form new list consisting of all elements of the sub-lists in reverse order,
  ```scheme
  ((1 2) (3 4 5) (6)) => (6 5 4 3 2 1)
  ```
- `rev-lists-all`: given a list of lists, form new list from reversal of elements of each list,
  ```scheme
  ((1 2) (3 4 5) (6)) => (2 1 5 4 3 6)
  ```

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`

```
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 class size < 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.
```

```
Calling Procedure

; Precondition: smaller? is a procedure that can be applied to any two elements of list, it should return #t in the first argument is 'smaller' than the second
(define bubblesort
  (lambda (let smaller? (lambda (let smaller? (lambda (length list) ))
    (lambda (x y) (if (null? x) y (y smaller? x))
      (lambda (list)
        (if (null? list) list
          (let ((smaller? smaller?))
            (let loop ((list list) (result nil))
              (if (null? list) result
                (let ((first list)
                  (rest (tail list)))
                  (loop rest (append (if (smaller? first rest) (cons first result) (cons (car rest) result))))))))))))))
```

```
What we want in the end

Sample run of procedure bubblesort

```
```
The Inner Loop

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

(define bubbletrip
  (if (zero? (length lst))
      (lst)
      (let ((smaller? (cdr lst) (cdr lst)))
        (do (car lst)
             (bubbletrip (cadr lst))
             smallers?
             (+ n 1))
      
      (let ((smaller? (cadr lst) (cadr lst))
             (smallers? (cadr lst)))
        (bubbletrip (cadr lst))
        smallers?
        (+ n 1))
      )
    )
)

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