Overview

- Review of problem set 1
- S-expressions
- Recursive list processing
- Applicative programming
- A look forward: local variables

Thinking recursively

```
(define sumprimes (n)
  (if (= n 0)
      0
      (+ (nthprime n) (sumprimes (- n 1))))
)
```

versus

```
(define sumprimeHelper (n m x y)
  (if (= n m)
      y
      (if (= (prime x) 1)
          (sumprimeHelper n (+ m 1) (+ x 1) (+ y x))
          (sumprimeHelper n m (+ x 1) y)))
)
```

```
(define sumprimes (n)
  (sumprimeHelper n 0 2 0)) ;1st prime is 2
```

Don't use `begin` unless you need a side effect:

```
(if (< i (+ (/ n 2) 1))
  (begin
    (if (= (mod n i) 0)
      0
      (primer n (+ i 1))))
  1)
```

is the same as

```
(if (< i (+ (/ n 2) 1))
  (if (= (mod n i) 0)
      0
      (primer n (+ i 1)))
  1)
```
Extending Impcore to uScheme

- A more general kind of value: the S-expression
- (Simplified) concrete syntax:
  
  literal ::= 
  integer | #t | #f | 'S-exp
  S-exp ::= 
  literal | symbol 
  | (S-exp1 ... S-expn)

List processing

- Creators
  ', list1, list2, ...
- Producers
  cons, append, reverse, ...
- Observers
  car, cdr, null?, caar, cadr, cdar, ...

Linear recursion on lists

- A simple pattern:
  
  (define some-function (l ...)
   (if (null? l) (base-case)
    (... 
     (some-function (cdr l)...)))
  )
- E.g. length, equal?, append, reverse

Accumulating parameters

- Most of you know this method and applied it in problem set 1
- Improving naive reverse:
  
  (define revapp (l1 l2)
   (if (null? l1) l2
    (revapp (cdr l1) 
            (cons (car l1) l2))))
  (define reverse (l) ?)
Using lists - primes

(define remove-multiples (n l)
  (if (null? l) '()
    (if (divides n (car l))
      (remove-multiples n (cdr l))
      (cons (car l)
        (remove-multiples n (cdr l))))))

Insertion sort

(define insert (x l)
  (if (null? l) (list1 x)
    (if (< x (car l)) (cons x l)
      (cons (car l)
        (insert x (cdr l))))))

(define sort (l)
  (if (null? l) '()
    (insert (car l) (sort (cdr l)))))

Primes continued

✧ Removing non-primes:

(define sieve (l)
  (if (null? l) '()
    (cons (car l)
      (sieve (remove-multiples (car l) (cdr l)))))

(define primes<= (n)
  (sieve (seq 2 n)))

Lists as sets

✧ Invariant: no duplicate elements

(define add-element (x s)
  (if (member? x s) s
    (cons x s)))

✧ ...simple definition of size:

(define size (s) (length s))
Sets continued

- ...complicated definition of union

```
(define union (s1 s2)
  (if (null? s1) s2
    (add-element (car s1)
      (union (cdr s1) s2))))
```

- Why not use `append`?

Sets are egalitarian!

- Anything can be a member of a "set": numbers, symbols, lists of all kinds

- E.g.

```
->(val s
  (add-element '(a b)
    (add-element 3 emptyset)))
((a b) 3)
```

Lists as maps: a-lists

- General form
  ```
  ((key value) (key value) ...)
  ```
- Keys and values can be anything
- Lookup by linear search
- `bind` and `find` create and use maps

A mental exercise

- Here's the definition of `bind`
  ```
  (define bind (x y alist)
    (if (null? alist)
      (list1 (list2 x y))
      (if (equal? x (caar alist))
        (cons (list2 x y) (cdr alist))
        (cons (car alist)
          (bind x y (cdr alist))))))
  ```
- What's the result of `(bind 'A 'B alist)` if `alist` is `((C D) (B C) (A E))`?
Property lists

- We can bind a key in an a-list to another a-list!
- This was very exciting in the early days of knowledge representation research.

```scheme
(define property (x p plist)
  (find p (find x plist)))
```

Applicative programming

- ...works by applying functions, not by mutating state
- The meaning of a name in a given scope doesn't change over time - no set
- Therefore we can have confidence in some simple laws.

Some laws

- If all subexpressions are applicative
  - `(car (cons x y)) = x`
  - `(cdr (cons x y)) = y`
- Why do we need the caveat?
  - `-> (val x 0)`
    - `0`
    - `-> (cdr (cons (set x (+ x 1)) (set x (+ x 2))))`  
      - `3`
    - `-> (val x 0)`
      - `0`
    - `-> (set x (+ x 2))`  
      - `2`

cons makes pairs!

- We can cons anything to anything
  - `-> (cons 1 2)`
    - `(1 . 2)`
    - `-> (cons 1 '(2))`  
      - `(1 2)`
- ...so beware! Normally you want the second thing to be a list.
More laws

- `(pair? (cons x y)) = #t`
- `(null? (cons x y)) = #f`
- `(pair? '()) = #f`
- `(null? '()) = #t`
- `(car '()) ==> ERROR`
- `(cdr '()) ==> ERROR`

Laws for Booleans

- Again, if all subexpressions are applicative
  
  `(if #t x y) = x`
  `(if #f x y) = y`
  `(if (not p) x y) = (if p y x)`

- Laws for the novice programmer:
  
  `(if p #t #f) = p`
  `(if p #f #t) = (not p)`

A look ahead

- Local variables using let-binding
- E.g., instead of
  
  ```scheme
  (begin (set r (random))
    (if (< r 20) ...
      (if (< r 40) ... )))
  ```

  introduce a fresh variable:

  ```scheme
  (let ((r (random))
    (if (< r 20) ...
      (if (< r 40) ... )))
  ```

Assignment

- Read R&K 3.5, 3.6, and 3.7 to the beginning of section 3.7.2 on page 75.
- Problem set 2 is due Friday.
Local variables

- New syntax; the bindings look like an a-list
- Three types
  - Simple let binds simultaneously
  - let* binds sequentially
  - letrec binds recursively

Simultaneous binding

```
(define foo (x)
  (let ((x (car x))
        (y (cdr x)))
    (cons y x)))
-> (foo '(a b))
((b) . a)
-> (foo '((a b) c))
((c) a b)
```
Sequential binding

(define foo (x)
  (let* ((x (car x))
         (y (cdr x)))
    (cons y x)))

-> (foo '(a b))
error: cdr applied to non-pair a in (cdr x)
-> (foo '((a b) c))
  ((b) a b)

Recursive binding

(define foo (x)
  (letrec ((x (car x))
            (y (cdr x)))
    (cons y x)))

-> (foo '(a b))
error: car applied to non-pair () in (car x)

Local functions

(define naive-fib (n)
  (begin
    (set naive-fib-count (+ naive-fib-count 1))
    (if (< n 2) n
      (+ (naive-fib (- n 1))
         (naive-fib (- n 2))))))

-> (naive-fib 10)
55
-> naive-fib-count
177

(define fib (n)
  ;; bottom-up helper function: return a list of fib_i, fib_i-1
  (letrec ((fib* (lambda (i)
                    (begin
                      (set fib-count (+ fib-count 1))
                      (if (= 1 i) (list2 1 0)
                        (let* ((fibs (fib* (- i 1)))
                                (fib_i (car fibs))
                                (fib_i-1 (cadr fibs)))
                          (list2 (+ fib_i fib_i-1) fib_i))))))
    (car (fib* n))))

-> (fib 10)
55
-> fib-count
10
Let's do binary search trees. We need a slightly different rep from the textbook's:

```
(1 () (5 (2 () ()) ()))
```

```scheme
(define empty? (tree) (null? tree))
(define left (node) (cadr node))
(define right (node) (caddr node))
(define label (node)
  (if (empty? node) node
      (car node)))
(val empty_tree '())
```

Searching is easy - follow the data structure

```scheme
(define lookup (v t)
  (if (empty? t) #f
    (if (= v (label t)) ?
      (if (< v (label t))
        ?
        ?
    ))))
```

Insertion is also easy

```scheme
(define insert (v t)
  (if (empty? t) (list3 v '() '())
    (if (< v (label t))
        ?
        ?
    )))
```

Examples

```
-> (val t1
  (insert 2 (insert 1 (insert 5 empty_tree))))
(5 (1 () (2 () ())) ())
-> (val t2
  (insert 2 (insert 5 (insert 1 empty_tree))))
(1 () (5 (2 () ())) ())
```

```
-> (lookup 0 t2)
#f
-> (lookup 5 t2)
#t
```
Traversals

- Preorder, "level-order", and so on are left as exercises for the reader ;->
- ...but I will take questions

The power of lambda

- At the top level, not interesting
- Used for local function definition, more interesting
- We can pass functions as parameters
- ...and - more interesting - return them as results

Creating functions

- Remember the binary search trees
- We can create functions in our programs, e.g.

\[
\begin{align*}
\text{-> (val look_t2 (lambda (v) \text{(lookup v t2))})} \\
\text{look_t2} \\
\text{-> (look_t2 5)} \\
\#t \\
\text{-> (look_t2 7)} \\
\#f
\end{align*}
\]
Functions on the fly

-> (define look_in (t)
   (lambda (v) (lookup v t)))
look_in
-> (val look_t2 (look_in t2))
<procedure>
-> (look_t2 7)
#f
-> (look_t2 2)
#t

Closures

- We know what v means - a formal parameter that will be bound when the function is applied - but what does t mean in
  (lambda (v) (lookup v t))

Values for free variables

- Answer: it depends on the environment
- Evaluating a lambda-expression requires capturing the environment in a \textit{closure}
- We don't write closures explicitly; the interpreter pairs the lambda-expression with the current environment:
  \[
  \langle (\lambda (v) (\text{lookup} v t)), \{x \mapsto '()\} \rangle
  \]

Mutation and closures

- I just lied to you...
- ...we can't bind variables to values in environments
- Because of assignment, we bind variables to locations
- Bindings in a closure don't change, contents of locations can
Example

- A pair of functions that manipulate a counter

```scheme
(val resettable-counter-from
 (lambda (n)
  (list2
   (lambda () (set n (+ n 1)))
   (lambda () (set n 0))))
 (define step (c) ((car c)))
 (define reset (c) ((cadr c)))
 (val ten (resettable-counter-from 10))
 (val twenty (resettable-counter-from 20))
-> (step ten)
  11
 -> (step twenty)
  21
```

Preview of next week

- Closures for first-class functions
- Higher-order programming, e.g.

```scheme
(foldl (lambda (n t)
  (insert n t)) empty_tree
 '(1 5 2 4 3)))
... instead of
(insert 3 (insert 4 ... (insert 1 '())))
```