Membership revisited

From `duplicates?` function, `member?`

Laws:

```
(member? m '()) == #f
(member? m (cons m ks)) == #t
(member? m (cons k ks)) == (member? m ks), m != k
```

What kind of algorithm is this?

Your turn: Common list algorithms

Algorithms on linked lists (or arrays in sequence):

- **Search for an element**
- **What else?**

Predefined list algorithms

Some classics:

- **exists? (Example: Is there a number?)**
- **all? (Example: Is everything a number?)**
- **filter (Example: Select only the numbers)**
- **map (Example: Add 1 to every element)**
- **`foldr` (*Visit* every element)**

Fold also called `reduce`, `accum`, a "catamorphism"

Coding: Generalize linear search

Laws:

```
(member? m '())           = #f
(member? m (cons k ks)) = #t,                  if m == k
(member? m (cons k ks)) = (member? m ks),  if m != k
```

Generalize selection; make predicate a parameter:

```
(exists? p? '())           = #f
(exists? p? (cons y ys)) = #t,                  if (p? y)
(exists? p? (cons y ys)) = (exists? p? ys),  otherwise
```

Predicate `p?` could come from `curry` (forthcoming)

Defining exists?

```
; (exists? p? '())          = #f
; (exists? p? (cons y ys)) = #t,          if (p? y)
; (exists? p? (cons y ys)) = (exists? p? ys),
                                          otherwise

-> (define exists? (p? xs)
     (if (null? xs)
         #f
         (if (p? (car xs))
             #t
             (exists? p? (cdr xs)))))
-> (exists? symbol? '(1 2 zoo))
#t
-> (exists? symbol? '(1 2 (zoo)))
#f
```

Defining filter

```
; (filter p? '()) == '()
; (filter p? (cons y ys)) ==
;          (cons y (filter p? ys)), when (p? y)
; (filter p? (cons y ys)) ==
;          (filter p? ys), when (not (p? y))

-> (define filter (p? xs)
     (if (null? xs)
       '()
       (if (p? (car xs))
         (cons (car xs) (filter p? (cdr xs)))
         (filter p? (cdr xs)))))
```

Running filter

```
-> (filter (lambda (n) (>  n 0)) '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter (lambda (n) (<= n 0)) '(1 2 -3 -4 5 6))
(-3 -4)
-> (filter ((curry <)  0) '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter ((curry >=) 0) '(1 2 -3 -4 5 6))
(-3 -4)
```

Your turn: `map`

```
-> (map      add3 '(1 2 3 4 5))
(4 5 6 7 8)

;; (map f '())          =
;; (map f (cons y ys)) =
```

Answers: map

```
-> (map      add3 '(1 2 3 4 5))
(4 5 6 7 8)

; (map f '())          ==  '()
; (map f (cons y ys)) ==  (cons (f y) (map f ys))
```

Defining and running map

```
; (map f '())          == '()
; (map f (cons y ys)) == (cons (f y) (map f ys))
-> (define map (f xs)
     (if (null? xs)
       '()
       (cons (f (car xs)) (map f (cdr xs)))))
-> (map number? '(3 a b (5 6)))
(#t #f #f #f)
-> (map *100 '(5 6 7))
(500 600 700)
```

Foldr

Algebraic laws for foldr

Idea: $\lambda + . \lambda 0 . x_1 + \cdots + x_n + 0$

```
(foldr (plus zero '()))          = zero
(foldr (plus zero (cons y ys))) =
                    (plus y (foldr plus zero ys))
```

Note: Binary operator + associates to the right.

Note: `zero` might be identity of `plus`.

Code for foldr

Idea: $\lambda + . \lambda 0 . x_1 + \cdots + x_n + 0$

```
-> (define foldr (plus zero xs)
     (if (null? xs)
       zero
       (plus (car xs) (foldr plus zero (cdr xs)))))
-> (val sum  (lambda (xs) (foldr + 0 xs)))
-> (sum '(1 2 3 4))
10

-> (val prod (lambda (xs) (foldr * 1 xs)))
-> (prod '(1 2 3 4))
24
```

Another view of operator folding

```
'(1 2 3 4)   =   (cons 1 (cons 2 (cons 3 (cons 4 '()))))
(foldr + 0 '(1 2 3 4))
             =   (+    1 (+    2 (+    3 (+    4 0  ))))
(foldr f z '(1 2 3 4))
             =   (f    1 (f    2 (f    3 (f    4 z  ))))
```

Your turn

Idea: $\lambda + . \lambda 0 . x_1 + \cdots + x_n + 0$

```
-> (define combine (x a) (+ 1 a))
-> (foldr combine 0 '(2 3 4 1))
???
```

Wait for it

Answer

Idea: $\lambda + . \lambda 0 . x_1 + \cdots + x_n + 0$

```
-> (define combine (x a) (+ 1 a))
-> (foldr combine 0 '(2 3 4 1))
4
```

What function have we written?

Your turn: Explain the design

1. Functions like `exists?`, `map`, `filter` are subsumed by
2. Function `foldr`, which is subsumed by
3. Recursive functions

Seems redundant: Why?

Cornucopia of one-argument functions

The “function factory”

The idea of currying

```
-> (map      ((curry +) 3) '(1 2 3 4 5))
; add 3 to each element

-> (exists? ((curry =) 3) '(1 2 3 4 5))
; is there an element equal to 3?

-> (filter  ((curry >) 3) '(1 2 3 4 5))
; keep elements that 3 is greater then
```

To get one-argument functions: Curry

```
-> (val positive? (lambda (y) (< 0 y)))
-> (positive? 3)
#t
-> (val <-c (lambda (x) (lambda (y) (< x y))))
-> (val positive? (<-c 0)) ; "partial application"
-> (positive? 0)
#f
```

What's the algebraic law for `curry`?

```
...  (curry f) ...   =  ... f ...
```

Keep in mind:
All you can do with a function is apply it!

```
(((curry f) x) y) = (f x y)
```

Three applications: so implementation will have three `lambda`s

No need to Curry by hand!

```
;; curry : binary function -> value -> function
-> (val curry
     (lambda (f)
       (lambda (x)
         (lambda (y) (f x y)))))
-> (val positive? ((curry <) 0))
-> (positive? -3)
#f
-> (positive? 11)
#t
```

Your turn!

```
-> (map      ((curry +) 3) '(1 2 3 4 5))
???
-> (exists? ((curry =) 3) '(1 2 3 4 5))
???
-> (filter  ((curry >) 3) '(1 2 3 4 5))
???                              ; tricky
```

Answers

```
-> (map      ((curry +) 3) '(1 2 3 4 5))
(4 5 6 7 8)
-> (exists? ((curry =) 3) '(1 2 3 4 5))
#t
-> (filter   ((curry >) 3) '(1 2 3 4 5))
(1 2)
```

One-argument functions compose

```
-> (define o (f g) (lambda (x) (f (g x))))
-> (define even? (n) (= 0 (mod n 2)))
-> (val odd? (o not even?))
-> (odd? 3)
#t
-> (odd? 4)
#f
```

Next up: proving facts about functions