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;**
  also called `reduce`, `accum`, a "catamorphism")

The power of lambda

Using first-class functions to enlarge your vocabulary

- List computations
- Cheap functions from other functions

Supported in many languages: Haskell, ML, Python, JavaScript

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? number? '(1 2 zoo))
#t
-> (exists? number? '(apple orange))
#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)
```

Defining and running map

```
-> (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 (lambda(x)(* x x)) '(5 6 7))
(25 36 49)
```

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  ))))
```

Why the redundancy?

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

Answer: Redundancy supports specificity, which makes code eaiser to reason about.

Currying: The motivation

Remember me?

```
q-with-y? = (lambda (z) (q? y z))
```

Happens so often, there is a function for it:

```
q-with-y? = ((curry q?) y)
```

Called a partial application (one now, one later)

Map/search/filter love curried functions

```
-> (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
```

The idea of currying

- Input: a binary function `f(x,y)`
- Output: a function `f'`
  - Input: argument `x`
  - Output: a function `f''`
    - Input: argument `y`
    - Output: `f(x,y)`

What is the benefit?

- Functions like `exists?`, `all?`, `map`, and `filter` expect a function of one argument.

To get there, we use currying and partial application.

Slogan: *Curried functions take their arguments "one-at-a-time."*

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

From law to code

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

Composing Functions

In math, what is the following equal to?

```
(f o g)(x) == ???
```

Another algebraic law, another function:

```
(f o g) (x) = f(g(x))
(f o g) = lambda x. (f (g (x)))
```

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
```

Proofs about functions

Function consuming *A* is related to proof about *A*

- **Q: How to prove two lists are equal?**
  A: Prove they are both '() or that they are both cons cells cons-ing equal car's to equal cdr's
- **Q: How to prove two *functions* equal?**
  A: Prove that when applied to equal arguments they produce equal results.