2-digit elements revisited

Laws:

(2digit-elements '()) == '()
(2digit-elements (cons y ys)) =
  (cons y (2digit-elements ys)), when (2digit? y)
(2digit-elements (cons y ys)) =
  (2digit-elements ys), when not (2digit? y)

How to select

• Odd elements?
• Prime elements?
• Numeric elements?
Generalize 2-digit elements

Laws:

(2digit-elements '()) == '()
(2digit-elements (cons y ys)) =
    (cons y (2digit-elements ys)), when (2digit? y)
(2digit-elements (cons y ys)) =
    (2digit-elements ys), when not (2digit? y)

Generalize selection; make predicate a parameter:

(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)

Predicate p? could come from curry
Your turn

Common computations on linked lists
Defining exists?

; (exists? p? '()) == #f
; (exists? p? (cons y ys)) == (p? y) or (exists p? ys)
-> (define exists? (p? xs)
   (if (null? xs)
     #f
     (if (p? (car xs))
       #t
       (exists? p? (cdr xs)))))
-> (exists? pair? '(1 2 3))
#f
-> (exists? pair? '(1 2 (3)))
#t
-> (exists? zero? '(1 2 3))
#f
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:** \( \lambda+ . \lambda^0 . x_1 + \cdots + x_n + 0 \)

\[
(foldr (\text{plus} \ 0 \ '()) \quad = \quad \text{zero}
\]

\[
(foldr (\text{plus} \ 0 \ (\text{cons} \ y \ ys))) = \\
(\text{plus} \ y \ (foldr \ \text{plus} \ 0 \ ys))
\]

**Note:** Binary operator \(+\) associates to the right.

**Note:** \(\text{zero}\) might be identity of \(\text{plus}\).
Code for foldr

**Idea:** $\lambda+. \lambda 0. x_1 + \ldots + x_n + 0$

-> (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

\[ (1 \ 2 \ 3 \ 4) = (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ '()')))) \]

\[(\text{foldr} \ + \ 0 \ '(1 \ 2 \ 3 \ 4)) \]
\[= (+ \ 1 \ (+ \ 2 \ (+ \ 3 \ (+ \ 4 \ 0 )))) \]

\[(\text{foldr} \ f \ z \ '(1 \ 2 \ 3 \ 4)) \]
\[= (f \ 1 \ (f \ 2 \ (f \ 3 \ (f \ 4 \ z )))) \]
Your turn

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

$\Rightarrow$ (define combine (x a) (+ 1 a))
$\Rightarrow$ (foldr combine 0 '(2 3 4 1))

???
Wait for it
Answer

Idea: \( \lambda+. \lambda 0. x_1 + \cdots + x_n + 0 \)

\[
\begin{align*}
\rightarrow & \quad (\text{define combine (x a) (+ 1 a)}) \\
\rightarrow & \quad (\text{foldr combine 0 '(2 3 4 1)}) \\
& 4
\end{align*}
\]

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 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`?

\[ \ldots (\text{curry } f) \ldots = \ldots f \ldots \]

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

\[ (((\text{curry } f) \ x) \ y) = (f \ x \ y) \]

Three applications: so implementation will have three lambdas
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

\[
\begin{align*}
&\rightarrow (\text{define } o \ (f \ g) \ (\lambda x \ (f \ (g \ x)))) \\
&\rightarrow (\text{define } \text{even?} \ (n) \ (= \ 0 \ (\text{mod} \ n \ 2))) \\
&\rightarrow (\text{val } \text{odd?} \ (o \ \text{not} \ \text{even?})) \\
&\rightarrow (\text{odd?} \ 3) \\
&\quad \#t \\
&\rightarrow (\text{odd?} \ 4) \\
&\quad \#f
\end{align*}
\]
Next up: proving facts about functions