Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

What links here
Related changes

Collatz conjecture

From Wikipedia, the free encyclopedia

The **Collatz conjecture** is a conjecture in mathematics that concerns a sequence defined as follows: start with any positive integer *n*. Then each term is obtained from the previous term as follows: if the previous term is even, the next term is one half the previous term. If the previous term is odd, the next term is 3 times the previous term plus 1. The conjecture is that no matter what value of *n*, the sequence will always reach 1.

Unsolved problem in mathematics:

Does the Collatz sequence eventually reach 1 for all positive integer initial values?

(more unsolved problems in mathematics)

The conjecture is named after Lothar Collatz, who introduced the idea in 1937, two years after receiving his doctorate.[1] It is also known as the **3*n* + 1 conjecture**, the **Ulam conjecture** (after Stanisław Ulam), **Kakutani's problem** (after Shizuo Kakutani), the **Thwaites conjecture** (after Sir Bryan Thwaites), **Hasse's algorithm** (after Helmut Hasse), or the **Syracuse problem**;[2][4] the sequence of numbers involved is referred to as the **hailstone sequence** or **hailstone numbers** (because the values are usually subject to multiple descents and ascents like hailstones in a cloud),[5][6] or as **wondrous numbers**.[7]

Paul Erdős said about the Collatz conjecture: "Mathematics may not be ready for such problems."[8] He also offered $500 for its solution.[9] Jeffrey Lagarias in 2010 claimed that based only on known information about this problem, "this is an extraordinarily difficult problem, completely out of reach of present day mathematics."[10]

Reminder: Racket functions

```
(define (int-average a b)
  (quotient (+ a b) 2))
```

(define (*function-name* *parameter*$_1$ … *parameter*$_n$)
    body)

Boolean functions

Write two functions:

odd? Takes a number and returns `true` if it is odd; `false` otherwise.

even? Takes a number and returns `true` if it is even; `false` otherwise.

You can assume that the functions take integers; you don't have to check.

Racket's modulo function may be useful.

Firstname Lastname **T. 10 / 9**

(Your response)

Boolean functions

Write two functions:

odd? Takes a number and returns true if it is odd; false otherwise.

even? Takes a number and returns true if it is even; false otherwise.

You can assume that the functions take integers; you don't have to check.

Racket's modulo function may be useful.

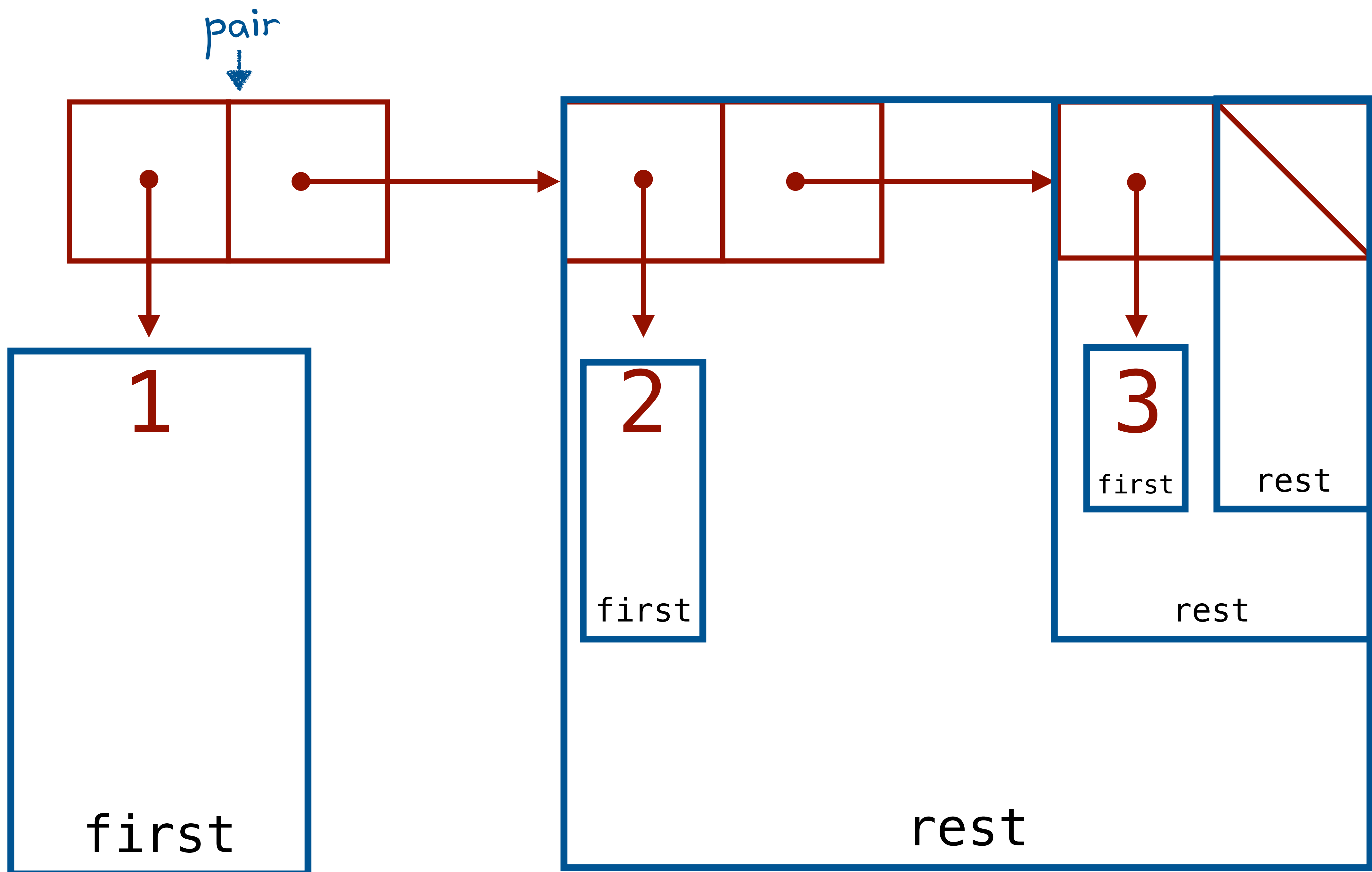
Firstname Lastname **T. 10 / 9**

```
(define (odd? n)
  (= (modulo n 2) 1))

(define (even? n)
  (not (odd? n)))
```

Note: odd? and even? are built-in Racket functions.

Reminder: Racket lists

```
(cons 1    (cons 2    (cons 3 '())))
```



Watch out!

Don't do these things (and if you accidentally do, know how to recognize them)

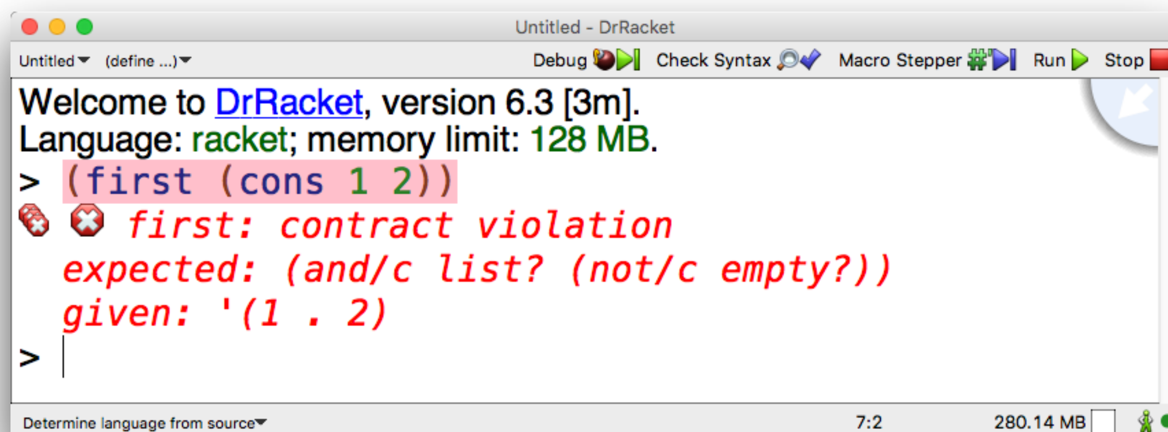
```
> (cons 1 2)
'(1 . 2)
```

←not a list!

This expression builds a *pair.*

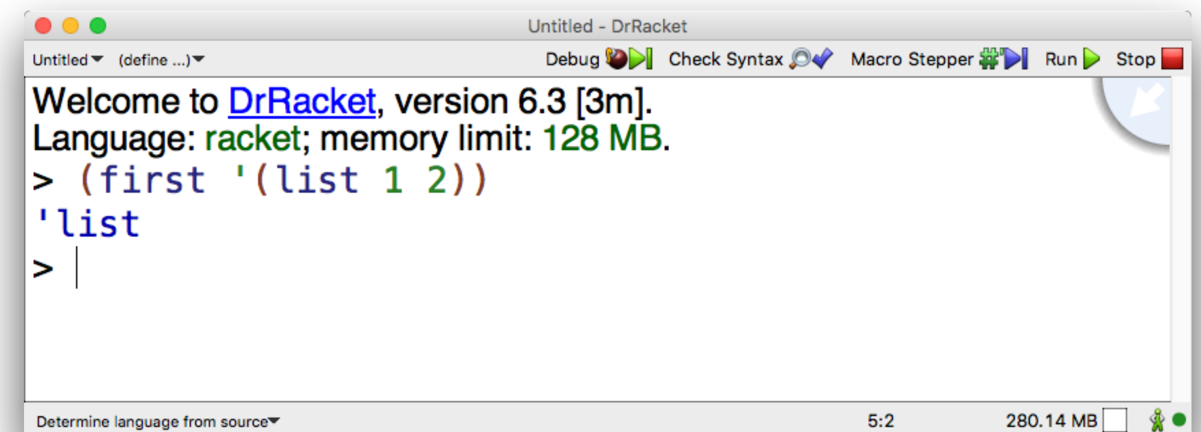
A pair is **not** a list.

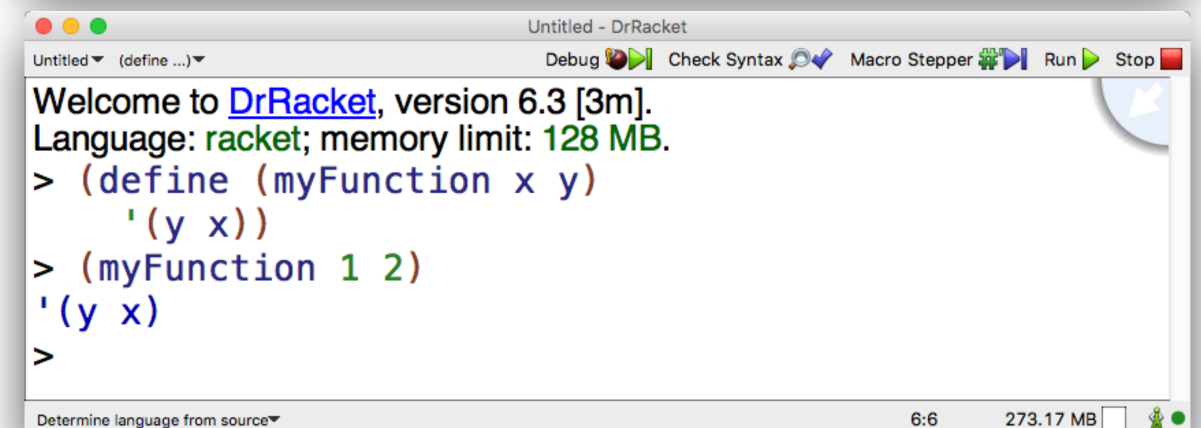
You can't call `first` on it.
You can't call `rest` on it.




```
> '(list 1 2)
'(list 1 2)
```

This expression builds a list whose first element is `'list`!





For more info, see: docs.racket-lang.org/reference/pairs.html

Recursive functions over lists

```
;; len
;;    inputs: a list, L
;;    outputs: the number of elements in the list
(define (len L)



                                     )
```

Recursive functions over lists

```
;; len
;;   inputs: a list, L
;;   outputs: the number of elements in the list
(define (len L)



                                    )


; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '((1 2 3))) 1)
```

Recursive functions over lists

```
;; len
;;   inputs: a list, L
;;   outputs: the number of elements in the list
(define (len L)

                                    )

; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '((1 2 3))) 1)
```

base case

recursive step

Recursive functions over lists

```
;; len
;;   inputs: a list, L
;;   outputs: the number of elements in the list
(define (len L)
  (if (empty? L)                          base case

                                  ))      recursive step


; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '((1 2 3))) 1)
```

Recursive functions over lists

```
;; len
;;   inputs: a list, L
;;   outputs: the number of elements in the list
(define (len L)
  (if (empty? L)
      0
                                ))
```

base case

recursive step

```
; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '((1 2 3))) 1)
```

Recursive functions over lists

```
;; len
;;   inputs: a list, L
;;   outputs: the number of elements in the list
(define (len L)
  (if (empty? L)
      0
      (+ 1 (len (rest L)))))      recursive step

; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '((1 2 3))) 1)
```

base case

recursive step

Let’s practice: `sum`

```
;; sum
;;   inputs: a list of integers, L
;;   outputs: the sum of the integers in L
(define (sum L)





; tests
(check-equal? (sum '()) 0)
(check-equal? (sum '(1)) 1)
(check-equal? (sum '(1 2 1)) 4)
(check-equal? (sum '(1 2 3 4)) 10)
```

Let's practice: `sum`

```
;; sum
;;   inputs: a list of integers, L
;;   outputs: the sum of the integers in L
(define (sum L)
  (if (empty? L)
      0
      (+ (first L) (sum (rest L)))))


; tests
(check-equal? (sum '()) 0)
(check-equal? (sum '(1)) 1)
(check-equal? (sum '(1 2 1)) 4)
(check-equal? (sum '(1 2 3 4)) 10)
```

Let's practice: `remove`

```
;; remove
;;   inputs: an integer i
;;           a list of integers, L
;;   outputs: a new list with 1st occurrence
;;            of i removed
(define (remove e L)
```

hint: two base cases

```
; tests
(check-equal? (remove 1 '()) '())
(check-equal? (remove 1 '(1)) '())
(check-equal? (remove 1 '(1 1)) '(1))
(check-equal? (remove 4 '(1 2 3)) '(1 2 3))
```

Let's practice: `remove`

```
;; remove
;;   inputs: an integer i
;;           a list of integers, L
;;   outputs: a new list with 1st occurrence
;;            of i removed
(define (remove e L)
  (cond [(empty? L) '()]
        [(= (first L) e) (rest L)]
        [else (cons (first L) (remove e (rest L)))]))


; tests
(check-equal? (remove 1 '()) '())
(check-equal? (remove 1 '(1)) '())
(check-equal? (remove 1 '(1 1)) '(1))
(check-equal? (remove 4 '(1 2 3)) '(1 2 3))
```

theplatypusblog.files.wordpress.com/2013/05/close-up-of-a-female-duck-billed-platypus-with-two-eggs-ornithorhynchus-anatinus1.jpg

Implement these functions

Using list operations (`cons`, `empty`, `empty?`, `first`, and `rest`) and recursion

`plus-one`: given a list of integers L produces a new list L' where all the corresponding elements are one larger

```
> (plus-one '(1 2 3 4))
'(2 3 4 5)
```

`double`: given a list of integers L produces a new list L' where all the corresponding elements of L are doubled

```
> (double '(1 2 3 4))
'(2 4 6 8)
```

`odds`: given a list of integers L produces a new list L' that contains only the odd integers from L

```
> (odds '(1 2 3 4))
'(1 3)
```

`bigs`: given a list of integers L produces a new list L' that contains only the integers from L that are larger than 30.

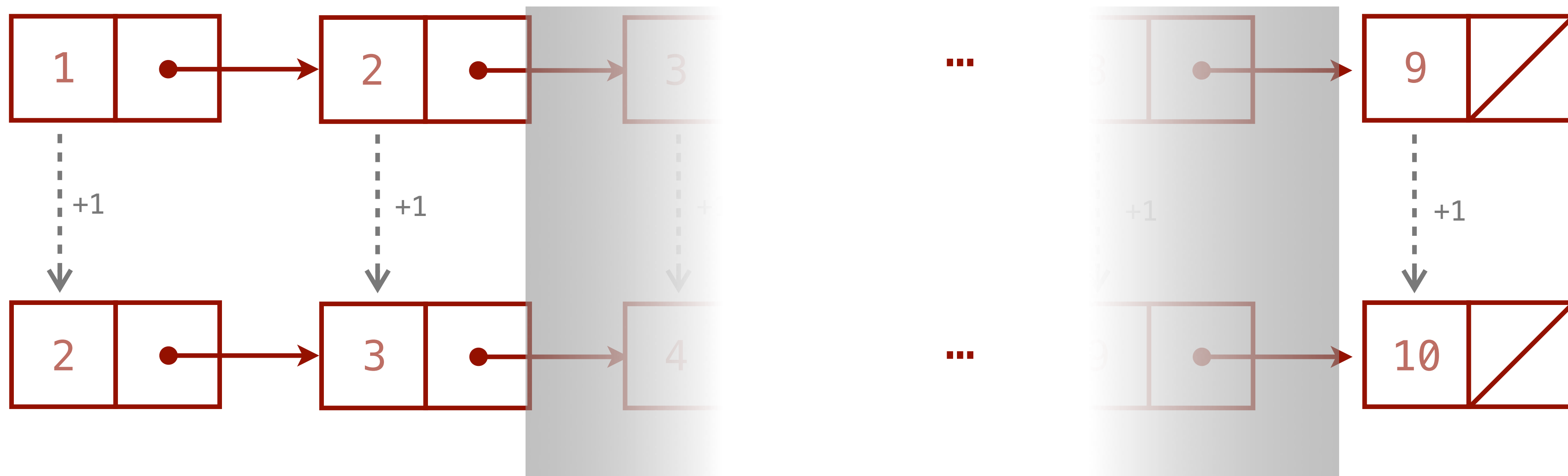
```
> (bigs '(100 1 42 35))
'(100 42 35)
```

`product`: given a list of integers L computes the product of all the integers.

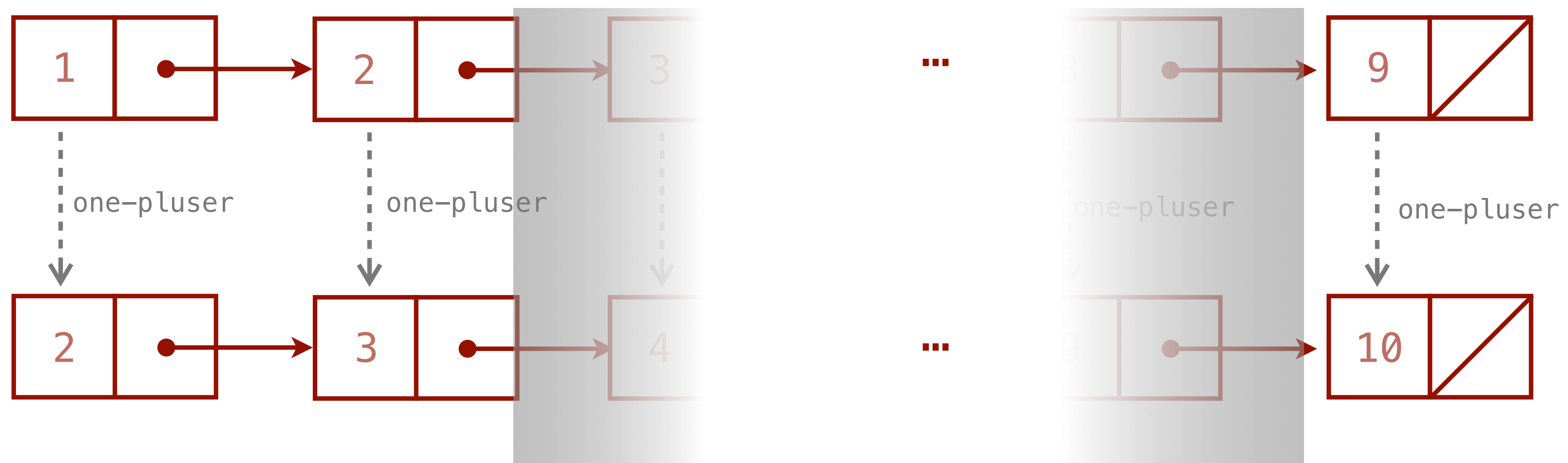
```
> (product '(1 2 3 4))
24
```

You can assume that the inputs to these functions are lists of integers.

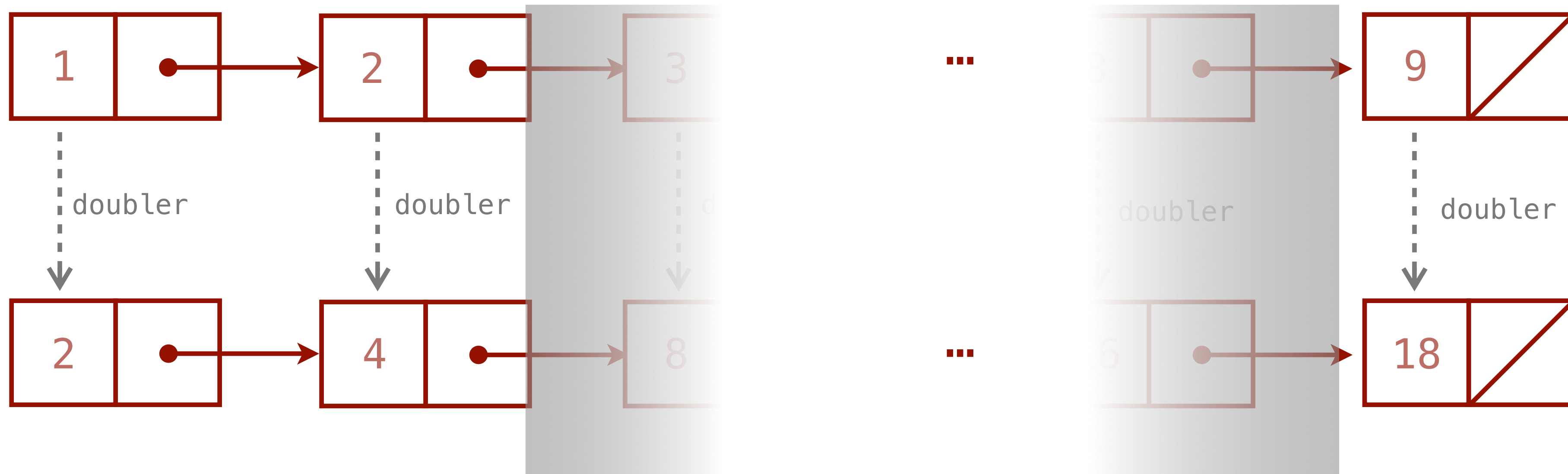
```
(define (plus-one L)
  (if (empty? L)
      '()
      (cons (+ 1 (first L)) (plus-one (rest L)))))
```



```
(define (one-pluser n) (+ n 1))

(define (plus-one L)
  (if (empty? L)
      '()
      (cons (one-pluser (first L)) (plus-one (rest L)))))
```



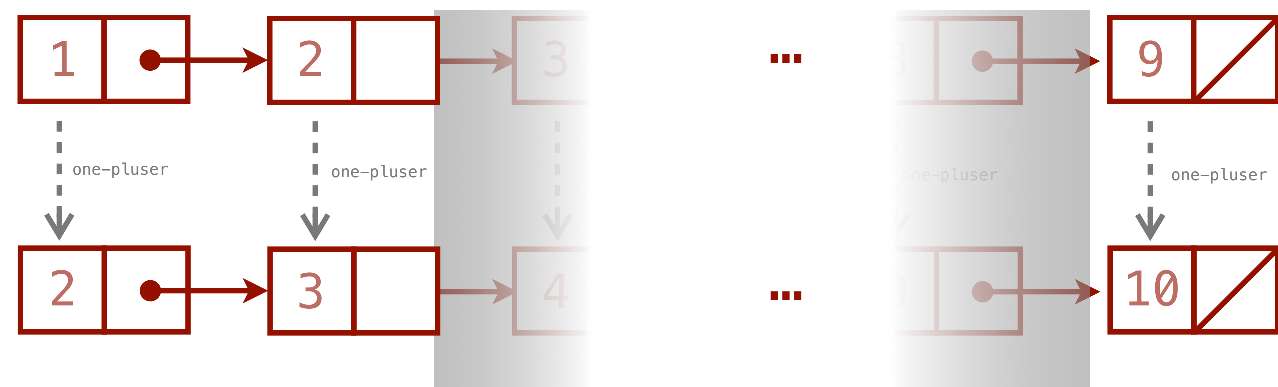
```
(define (doubler    n) (* n 2))

(define (double   L)
  (if (empty? L)
      '()
      (cons (doubler    (first L)) (double   (rest L)))))
```



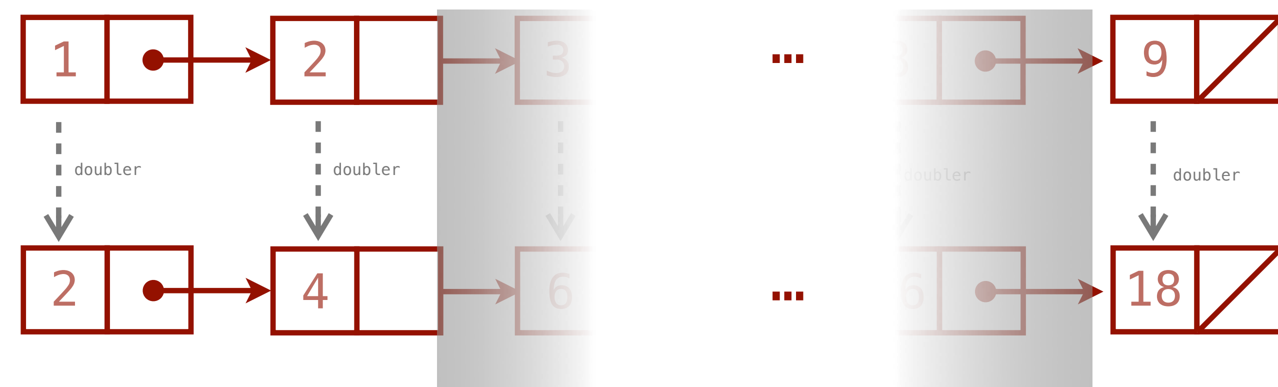
A common pattern

Apply a transformer function to each element of a list

```
(define (one-pluser n) (+ n 1))

(define (plus-one L)
  (if (empty? L)
      '()
      (cons (one-pluser (first L)) (plus-one (rest L)))))
```




```
(define (doubler n) (* n 2))

(define (double L)
  (if (empty? L)
      '()
      (cons (doubler (first L)) (double (rest L)))))
```



Higher-order functions

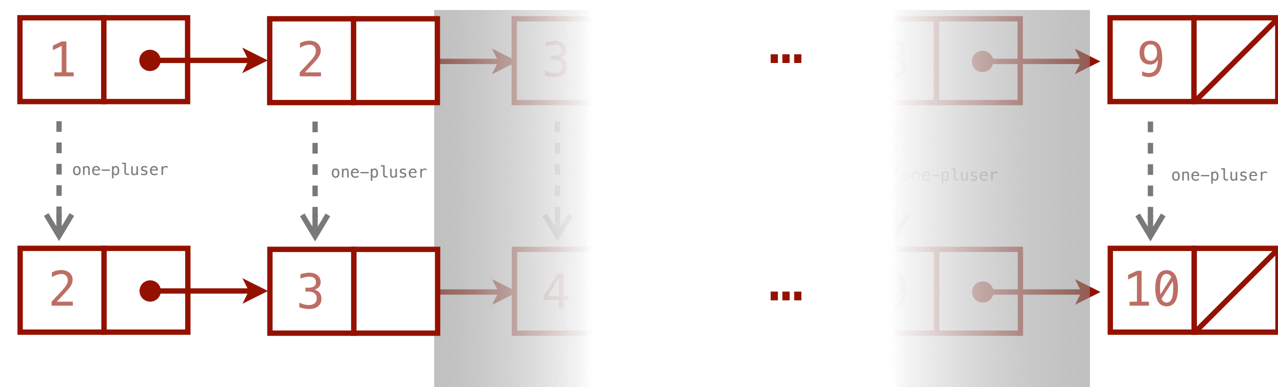
(functions as values)

A **higher-order function** is a function that takes at least one function as an argument *or* that returns a function as its result (or both).

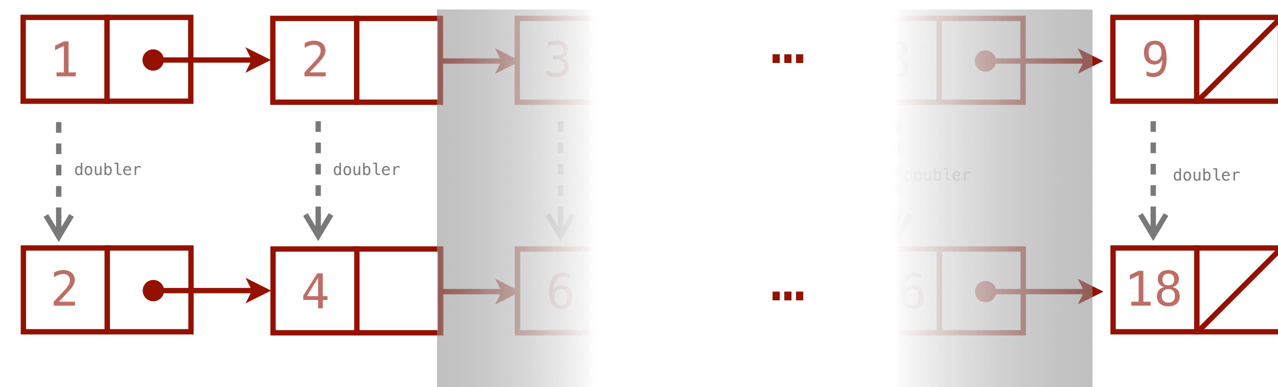
A common pattern

Apply a transformer function to each element of a list

```
(define (one-pluser n) (+ n 1))

(define (plus-one L)
  (if (empty? L)
      '()
      (cons (one-pluser (first L)) (plus-one (rest L)))))
```




```
(define (doubler n) (* n 2))

(define (double L)
  (if (empty? L)
      '()
      (cons (doubler (first L)) (double (rest L)))))
```

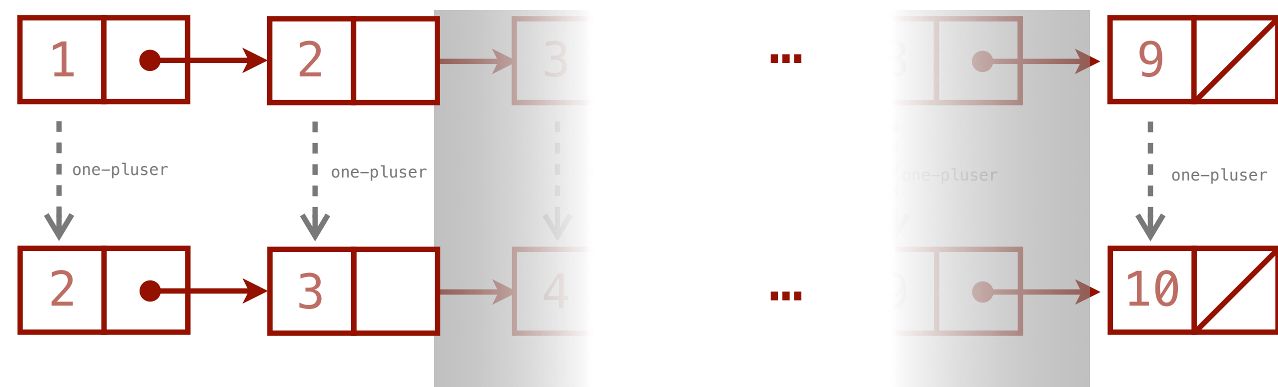


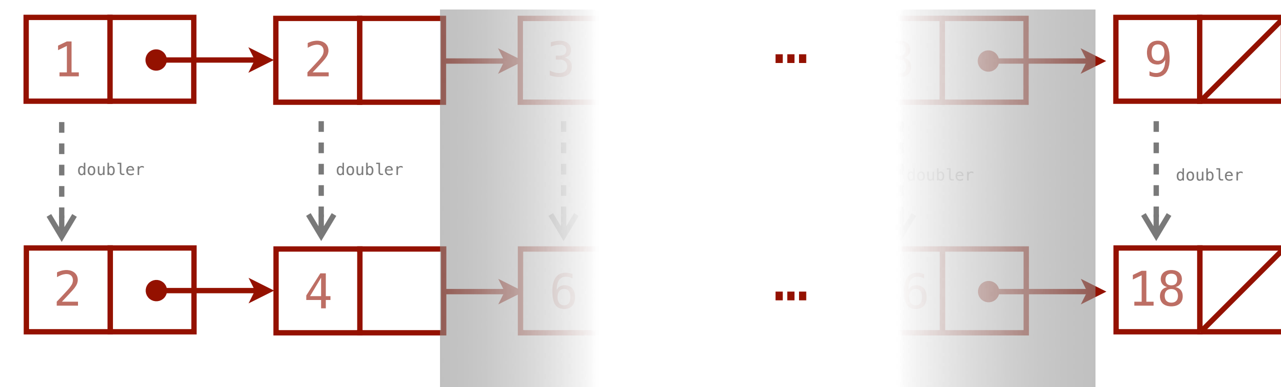
map

Apply a transformer function to each element of a list

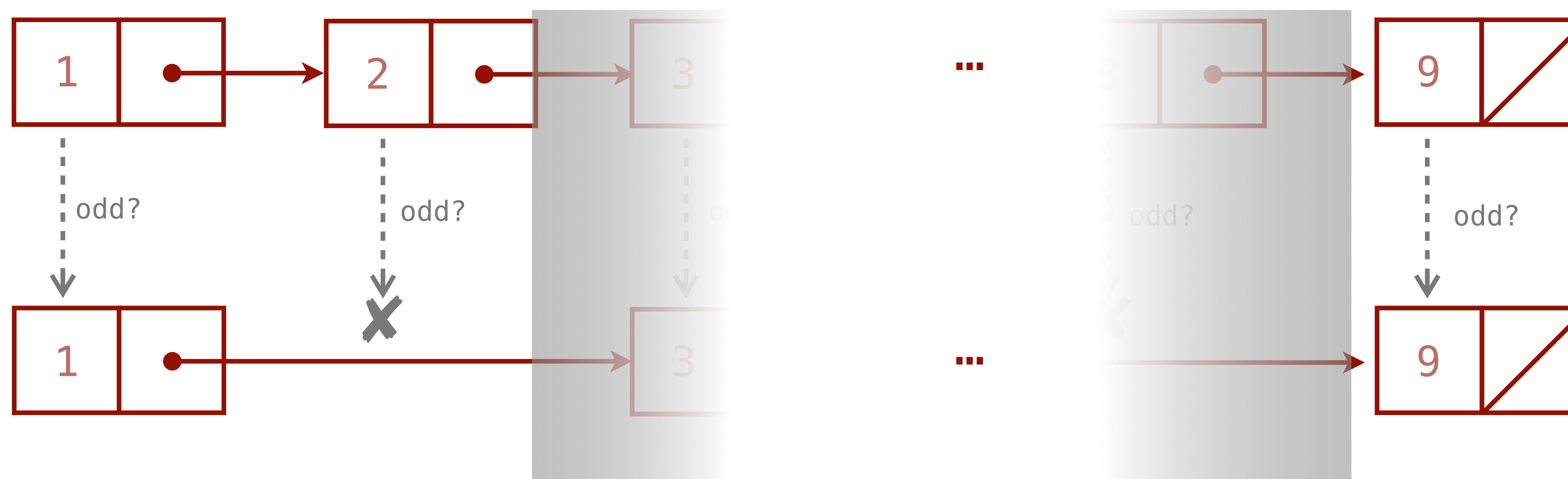
```
(define (one-pluser n) (+ n 1))

(define (plus-one L)
   (map one-pluser L))
```

```
(define (doubler n) (* n 2))

(define (double L)
   (map doubler L))
```

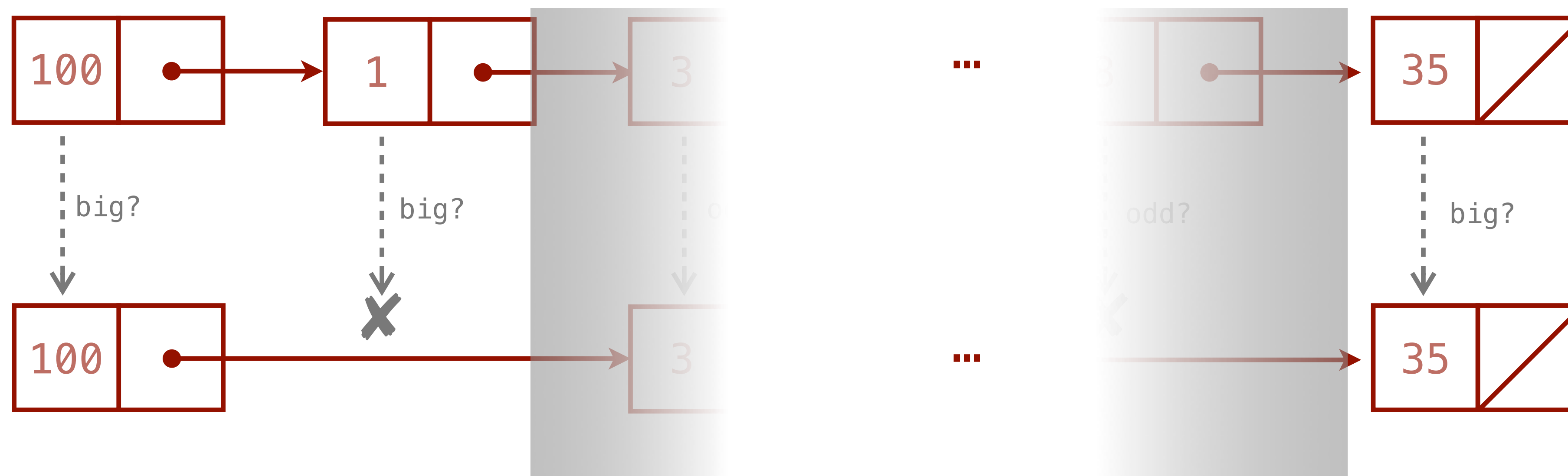




```
(define (odd?  n) (= (modulo n 2) 1))

(define (odds  L)
  (cond [(empty? L) '()]
        [(odd?  (first L)) (cons (first L) (odds  (rest L)))]
        [else (odds  (rest L))]))
```



```
(define (big?  n) (> n 30))

(define (bigs  L)
  (cond [(empty? L) '()]
        [(big?  (first L)) (cons (first L) (big  (rest L)))]
        [else (big  (rest L))]))
```

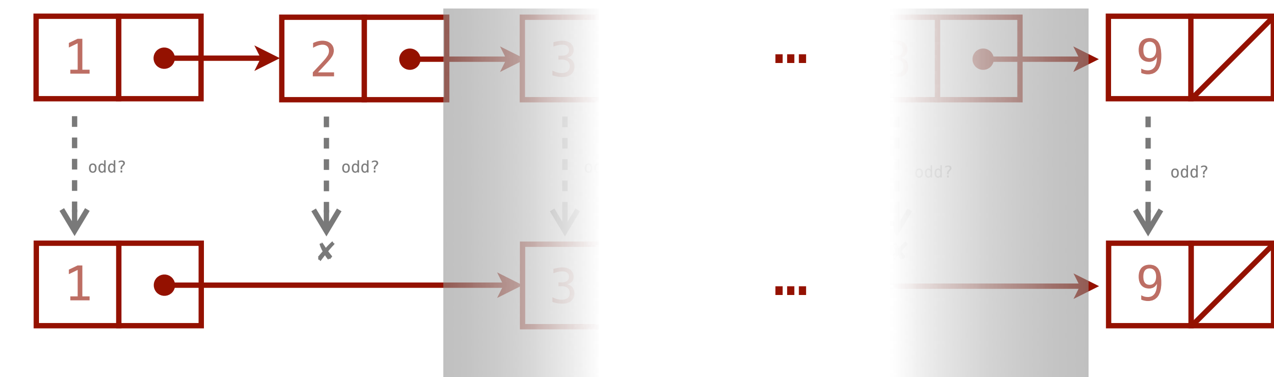


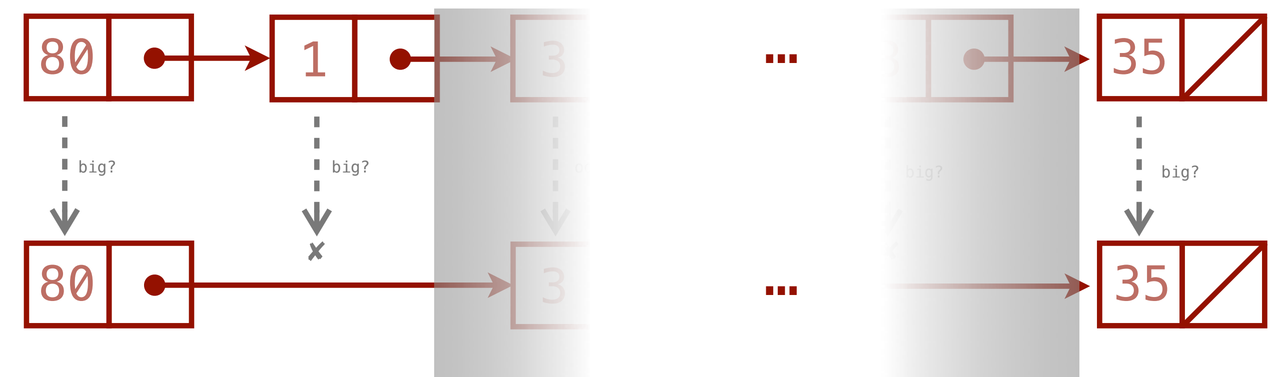
A common pattern

Use a predicate function to cull a list

```
(define (odd? n) (= (modulo n 2) 1))

(define (odds L)
  (cond [(empty? L) '()]
        [(odd? (first L)) (cons (first L) (odds (rest L)))]
        [else (odds (rest L))]))
```

```
(define (big? n) (> n 30))

(define (bigs L)
  (cond [(empty? L) '()]
        [(big? (first L)) (cons (first L) (bigs (rest L)))]
        [else (bigs (rest L))]))
```

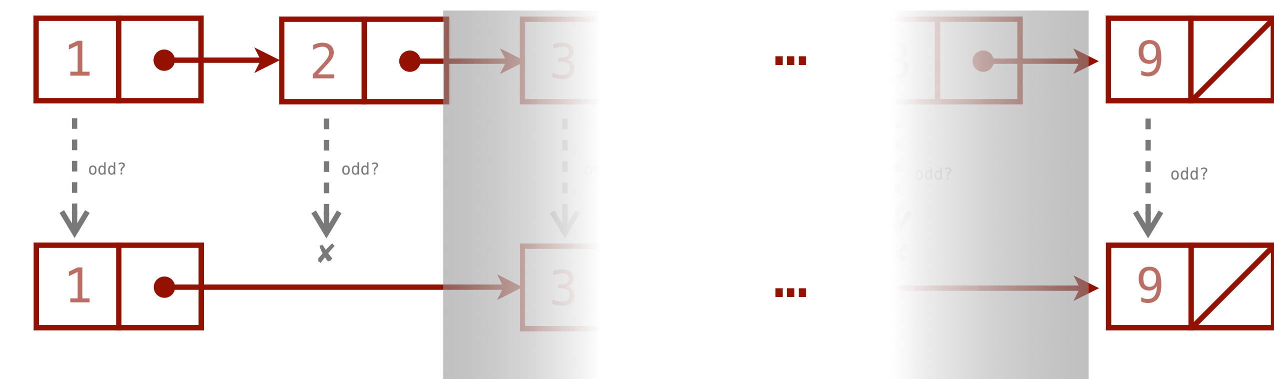




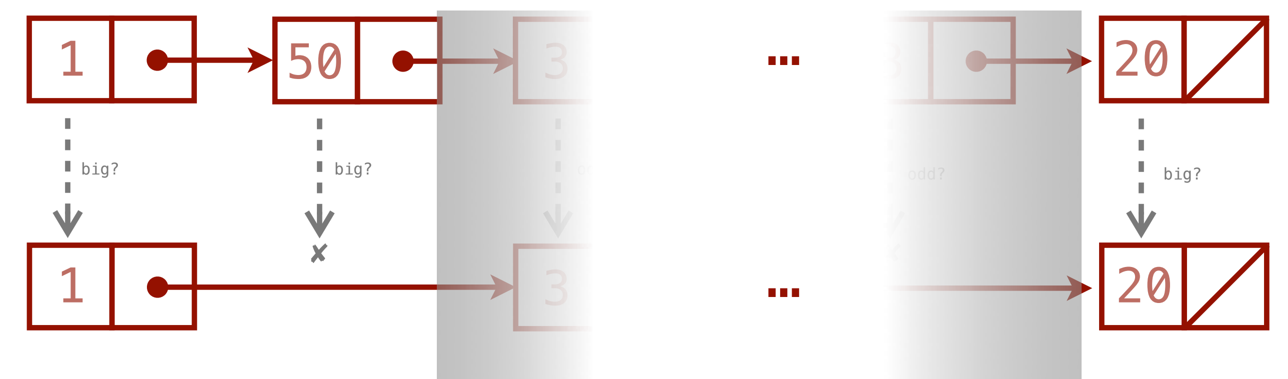
filter

Use a predicate function to cull a list

```
(define (odd? n) (= (modulo n 2) 1))

(define (odds L)
  (filter odd? L))
```




```
(define (big? n) (> n 30))

(define (big L)
  (filter big? L))
```



```
(define (sum      L)
  (if (empty? L)
      0
      (+ (first L) (sum      (rest L)))))
```

```
(define (product L)
  (if (empty? L)
      1
      (* (first L) (product (rest L)))))
```

A common pattern

Reduce a list to a value by accumulating the list's elements

```
(define (sum L)
  (if (empty? L)
      0
      (+ (first L) (sum (rest L)))))
```

```
(define (product L)
  (if (empty? L)
      1
      (* (first L) (product (rest L)))))
```

watch out for order of operations!

$$(\texttt{foldl f seed L}) \equiv (\texttt{f } v_n \texttt{ (... (f } v_0 \texttt{ seed) ...))}$$

foldl

Reduce a list to a value by accumulating the list's elements (start at the beginning of list and move towards end of the list)

```
(define (sum L)
  (foldl + 0 L))
```

```
(define (product L)
  (foldl * 1 L))
```

foldr

Reduce a list to a value by accumulating the list's elements (start at the end of list and move towards the beginning of the list)

watch out for order of operations!

```
(foldr f seed L) ≡ (f v0 (… (f vn seed) … ))
```

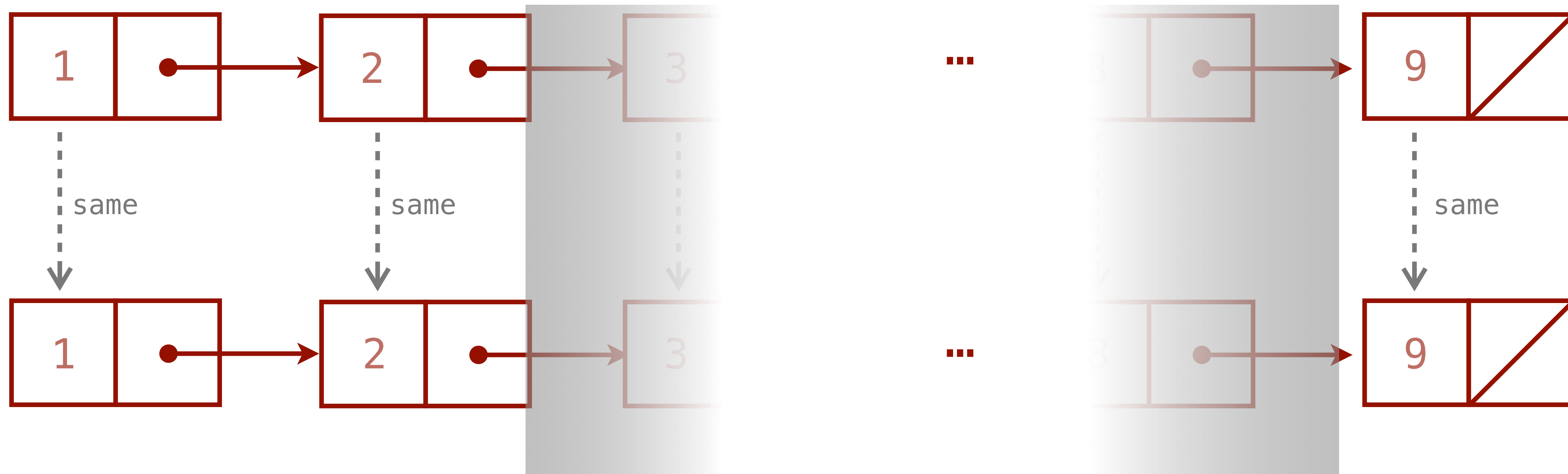
```
(define (sum L)
  (foldr + 0 L))
```

```
(define (product L)
   (foldr * 1 L))
```

A common pattern

Define one function to be used as an argument to another function

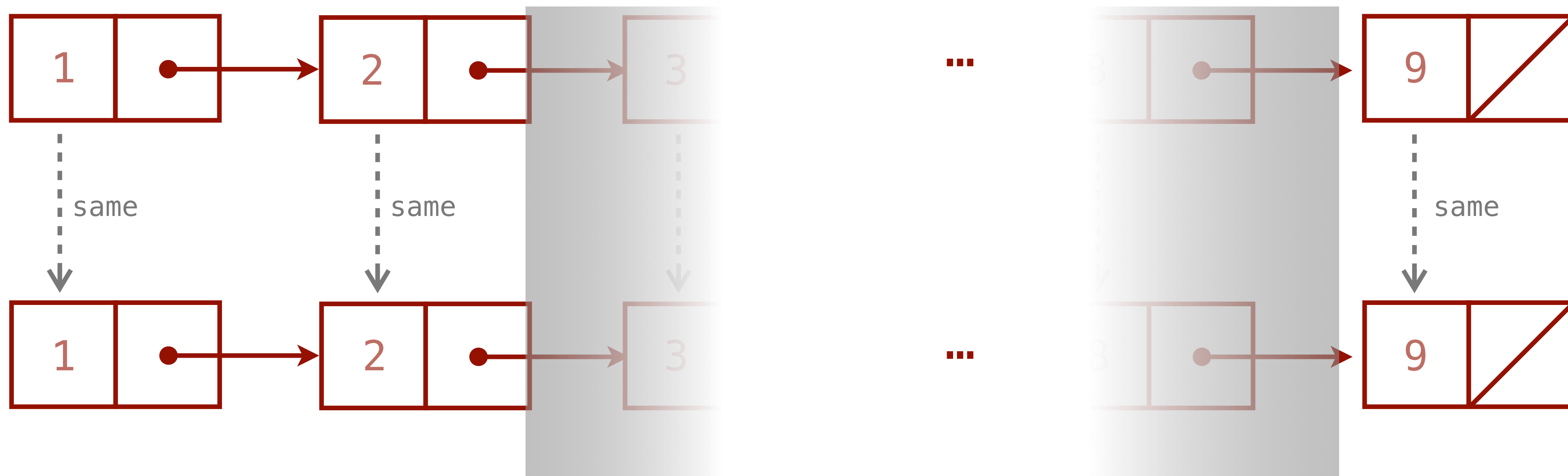
```
(define (one-pluser n) (+ n 1))

(define (plus-one L)
   (map one-pluser L))
```



Anonymous functions (λ)

Define one function to be used as an argument to another function

```
(define (plus-one L)
   (map (lambda (x) (+ x 1)) L))
```



Anonymous functions (λ)

```
(define (one-pluser x) (+ x 1))
```

≡

```
(define one-pluser (lambda (x) (+ x 1)))
```

Implement these functions

Using `map`, `filter`, `foldl`, `foldr`, and `lambda`

`times3`: multiplies each number in a list L by 3.

```
> (times3 '(1 2 3 4))
'(3 6 9 12)
```

`9multiples`: filters a list L to remove any numbers that aren't divisible by 9.

```
> (9multiples '(8 9 10))
'(9)
```

`sum-of-times10`: multiplies each element of a list L by 10, then sums the results

```
> (sum-of-times10 '(1 2 3))
60
```

`count-ones`: counts the number of times the value 1 appears in a list L

write one version using a combination of `filter` and `foldr`
write another version using only `foldr`

```
> (count-ones '(1 2 1))
2
```

click here for solutions

Bonus

`copy`, **`reverse`**

with `map`? with `foldl` or `foldr`?

`map`, **`filter`**, **`foldr`**, **`foldl`**

You can assume that the inputs to these functions are lists of integers.

A higher-order puzzle

Define a function called `divisible-by` that can be used as follows:

```
> (filter (divisible-by 2) (range 10))
'(0 2 4 6 8)
> (filter (divisible-by 3) (range 10))
'(0 3 6 9)
> (filter (divisible-by 4) (range 10))
'(0 4 8)
> (filter (divisible-by 5) (range 10))
'(0 5)
```

It may help to know the following functions:

```
> (range 10)
'(0 1 2 3 4 5 6 7 8 9)

> (modulo 10 2)
0
```

```
(define (divisible-by k)
  (lambda (n) (= 0 (modulo n k))))
```

The divisible-by function returns another function!

Assignment

Ciphers as a way to study lists and HOFs

Can you decrypt this message?

```
xjwtmhjxmjnfyzyjzftunjjksdumqswrytjqndtuff
irunssfhiwtyyxttgflgujjxynnsxqsntnlcnmtsdn
jiswjwysjsyfytfjfhjjlkxmsnkiryjjskztjitxfn
jxtytskxtszulftkvwiuhkbhymskhjnfnnrjmfyirt
zsryjzwjyhxzraxjhsxjcnthxudysmhnxutrnnqzfy
jjhzwsujhjfsfitfxjxtstcvhqxkqhxyxyknd
```

Strings can be interpreted as lists of characters

How to write a single character in Racket

```
> (string->list "Hello world!")
'(#\H #\e #\l #\l #\o #\space #\w #\o #\r #\l #\d #\!)

> (list->string
   '(#\H #\e #\l #\l #\o #\space #\w #\o #\r #\l #\d #\!))
"Hello world!"
```