The Pros of `cons`*:*
Programming with Pairs and Lists

CS251 Programming Languages
Fall 2016, Lyn Turbak

Department of Computer Science
Wellesley College

Racket Values

- booleans: `#t, #f`
- numbers:
  - integers: `42, 0, -273`
  - rationals: `2/3, -251/17`
  - floating point (including scientific notation): `98.6, -6.125, 3.141592653589793, 6.023e23`
  - complex: `3+2i, 17-23i, 4.5-1.4142i`

  Note: some are *exact*, the rest are *inexact*. See docs.
- strings: `"cat", "CS251", "αβγ", "To be\nor not\nto be"`
- characters: `#\a, #\A, #\5, #\space, #\tab, #\newline`
- anonymous functions: `(lambda (a b) (+ a (* b c)))`

What about compound data?

`cons` Glues Two Values into a Pair

A new kind of value:

- pairs (a.k.a. `cons` cells): `(cons` ***v1 v2***`)`

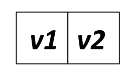
  e.g.,
  - `(cons 17 42)`
  - `(cons 3.14159 #t)`
  - `(cons "CS251" (λ (x) (* 2 x))`
  - `(cons (cons 3 4.5) (cons #f #\a))`

In Racket, type Command-\ to get λ char

Can glue any number of values into a `cons` tree!

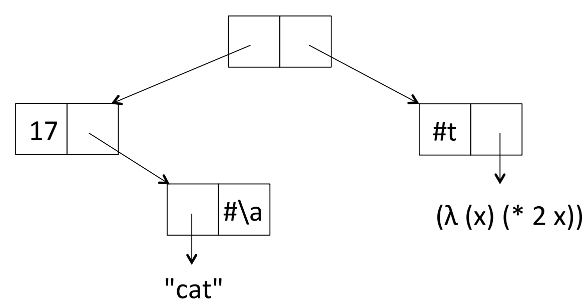
Box-and-pointer diagrams for `cons` trees

`(cons` ***v1 v2*****`)`**



Convention: put "small" values (numbers, booleans, characters) inside a box, and draw a pointers to "large" values (functions, strings, pairs) outside a box.

```
(cons (cons 17 (cons "cat" #\a))
      (cons #t (λ (x) (* 2 x))))
```



Evaluation Rules for `cons`

Big step semantics:

$$\frac{e1 \downarrow v1 \qquad e2 \downarrow v2}{(\texttt{cons}\ e1\ e2) \downarrow (\texttt{cons}\ v1\ v2)}\ \textbf{(cons)}$$

Small-step semantics:

(cons *e1* *e2*)

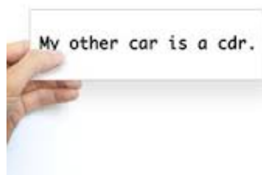
⇒* **(cons *v1* *e2*)** ; first evaluate ***e1*** to ***v1*** step-by-step

⇒* **(cons *v1* *v2*)** ; then evaluate ***e2*** to ***v2*** step-by-step

`cons` evaluation example

```
(cons (cons (+ 1 2) (< 3 4))
      (cons (> 5 6) (* 7 8)))
⇒ (cons (cons 3 (< 3 4))
        (cons (> 5 6) (* 7 8)))
⇒ (cons (cons 3 #t) (cons (> 5 6) (* 7 8)))
⇒ (cons (cons 3 #t) (cons #f (* 7 8)))
⇒ (cons (cons 3 #t) (cons #f 56))
```


`car` and `cdr`



- `car` extracts the left value of a pair

  `(car (cons 7 4))` ⇒ `7`

- `cdr` extract the right value of a pair

  `(cdr (cons 7 4))` ⇒ `4`

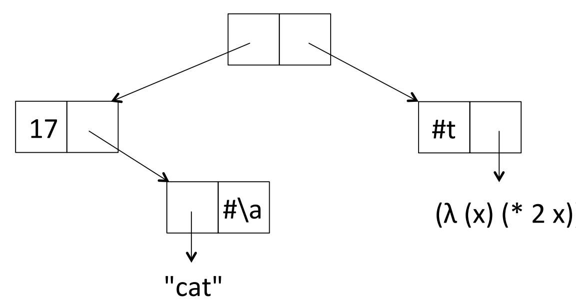
Why these names?

- `car` from "contents of address register"
- `cdr` from "contents of decrement register"

Practice with `car` and `cdr`

Write expressions using `car`, `cdr`, and `tr` that extract the five leaves of this tree:

```
(define tr
  (cons (cons 17 (cons "cat" #\a))
        (cons #t (λ (x) (* 2 x)))))
```



`cadr` and friends

- `(caar` ***e***`)` means `(car (car` ***e***`))`
- `(cadr` ***e***`)` means `(car (cdr` ***e***`))`
- `(cdar` ***e***`)` means `(cdr (car` ***e***`))`
- `(cddr` ***e***`)` means `(cdr (cdr` ***e***`))`
- `(caaar` ***e***`)` means `(car (car (car` ***e***`)))`

  ⋮
- `(cddddr` ***e***`)` means `(cdr (cdr (cdr (cdr` ***e***`))))`

Evaluation Rules for `car` and `cdr`

Big-step semantics:

$$\frac{e \downarrow (\texttt{cons}\ v1\ v2)}{(\texttt{car}\ e) \downarrow v1}\ (\text{car}) \qquad \frac{e \downarrow (\texttt{cons}\ v1\ v2)}{(\texttt{cdr}\ e) \downarrow v2}\ (\text{cdr})$$

Small-step semantics:

(car *e*)
⇒* **(car (cons *v1 v2*))** ; first evaluate ***e*** to pair step-by-step
⇒ ***v1*** ; then extract left value of pair

(cdr *e*)
⇒* **(cdr (cons *v1 v2*))** ; first evaluate ***e*** to pair step-by-step
⇒ ***v2*** ; then extract right value of pair

Semantics Puzzle

According to the rules on the previous page, what is the result of evaluating this expression?

```
(car (cons (+ 2 3) (* 5 #t)))
```

Note: there are two ``natural" answers. Racket gives one, but there are languages that give the other one!

Printed Representations in Racket Interpreter

```
> (lambda (x) (* x 2))
#<procedure>

> (cons (+ 1 2) (* 3 4))
'(3 . 12)

> (cons (cons 5 6) (cons 7 8))
'((5 . 6) 7 . 8)

> (cons 1 (cons 2 (cons 3 4)))
'(1 2 3 . 4)
```

What's going on here?

Display Notation and Dotted Pairs

- The **display notation** for `(cons` ***v1 v2***`)` is `(`***dn1*** `.` ***dn2***`)`, where ***dn1*** and ***dn2*** are the display notations for ***v1*** and ***v2***
- In display notation, a dot "eats" a paren pair that follows it directly:

  `((5 . 6) . (7 . 8))`

  becomes `((5 . 6) 7 . 8)`

  `(1 . (2 . (3 . 4)))`

  becomes `(1 . (2 3 . 4))`

  becomes `(1 2 3 . 4)`

  Why? Because we'll see this makes lists print prettily.
- The Racket interpreter puts a single quote mark before the display notation of a top-level pair value. (We'll say more about quotation later.)

`display` vs. `print` in Racket

```
> (display (cons 1 (cons 2 null)))
(1 2)

> (display (cons (cons 5 6) (cons 7 8)))
((5 . 6) 7 . 8)

> (display (cons 1 (cons 2 (cons 3 4))))
(1 2 3 . 4)

> (print(cons 1 (cons 2 null)))
'(1 2)

> (print(cons (cons 5 6) (cons 7 8)))
'((5 . 6) 7 . 8)

> (print(cons 1 (cons 2 (cons 3 4))))
'(1 2 3 . 4)
```


Functions Can Take and Return Pairs

```
(define (swap-pair pair)
   (cons (cdr pair) (car pair)))

(define (sort-pair pair)
   (if (< (car pair) (cdr pair))
       pair
       (swap pair)))
```

What are the values of these expressions?

- `(swap-pair (cons 1 2))`
- `(sort-pair (cons 4 7))`
- `(sort-pair (cons 8 5))`

Lists

In Racket, a **list** is just a recursive pattern of pairs.

A list is either

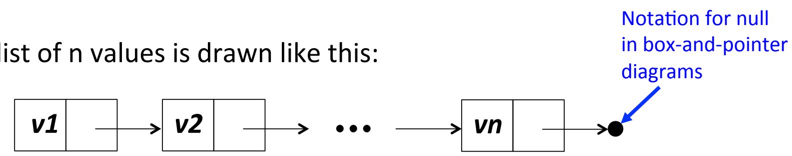
- The empty list `null`, whose display notation is ()
- A nonempty list `(cons` ***v_first v_rest***`)` whose
  - first element is ***v_first***
  - and the rest of whose elements are the sublist ***v_rest***

E.g., a list of the 3 numbers 7, 2, 4 is written

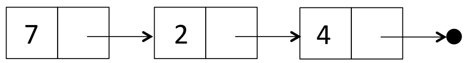
```
(cons 7 (cons 2 (cons 4 null)))
```

Box-and-pointer notation for lists

A list of n values is drawn like this:



For example:



`list` sugar

Treat `list` as syntactic sugar:

- `(list)` desugars to `null`
- `(list` ***e1*** `…)` desugars to `(cons` ***e1*** `(list …))`

For example:

`(list (+ 1 2) (* 3 4) (< 5 6))`
desugars to `(cons (+ 1 2) (list (* 3 4) (< 5 6)))`
desugars to `(cons (+ 1 2) (cons (* 3 4) (list (< 5 6))))`
desugars to `(cons (+ 1 2) (cons (* 3 4) (cons (< 5 6) (list))))`
desugars to `(cons (+ 1 2) (cons (* 3 4) (cons (< 5 6) null)))`

* This is a white lie, but we can pretend it's true for now

Display Notation for Lists

The "dot eats parens" rule makes lists display nicely:

`(list 7 2 4)`
desugars to `(cons 7 (cons 2 (cons 4 null))))`
displays as (before rule) `(7 . (2 . (4 . ())))`
displays as (after rule) `(7 2 4)`
prints as `'(7 2 4)`

In Racket:

```
> (display (list 7 2 4))
(7 2 4)

> (display (cons 7 (cons 2 (cons 4 null))))
(7 2 4)
```


`list` and small-step evaluation

It is sometimes helpful to both desugar and resugar with `list`:

`(list (+ 1 2) (* 3 4) (< 5 6))`
desugars to `(cons (+ 1 2) (cons (* 3 4) (cons (< 5 6) null)))`
⇒ `(cons 3 (cons (* 3 4) (cons (< 5 6) null)))`
⇒ `(cons 3 (cons 12 (cons (< 5 6) null)))`
⇒ `(cons 3 (cons 12 (cons #t null)))`
resugars to `(list 3 12 #t)`

Heck, let's informally write this as:

```
(list (+ 1 2) (* 3 4) (< 5 6))
⇒ (list 3 (* 3 4) (< 5 6))
⇒ (list 3 12 (cons (< 5 6))
⇒ (list 3 12 #t)
```

`first`, `rest`, and friends

- `first` returns the first element of a list:

  `(first (list 7 2 4)) ⇒ 7`

  (`first` is almost a synonym for `car`, but requires its argument to be a list)
- `rest` returns the sublist of a list containing every element but the first:

  `(rest (list 7 2 4)) ⇒ (list 2 4)`

  (`rest` is almost a synonym for `cdr`, but requires its argument to be a list)
- Also have `second`, `third`, …, `ninth`, `tenth`

Recursive List Functions

Because lists are defined recursively, it's natural to process them recursively.

Typically (but not always) a recursive function `recf` on a list argument `L` has two cases:

- **base case:** what does `recf` return when L is empty? (Use `null?` to test for an empty list)
- **recursive case:** if L is the nonempty list `(cons` ***v_first v_rest***`)` how are ***v_first*** and `(recf` ***v_rest***`)` combined to give the result for `(recf L)`?

  Note that we ``blindly" apply `recf` to ***v_rest***!

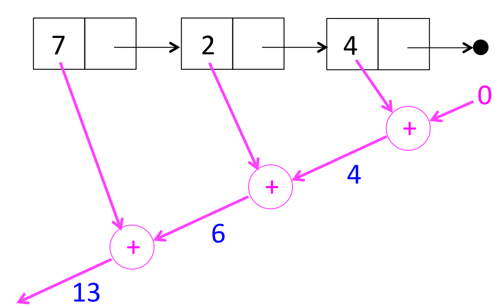
Example: `sum`

`(sum ns)` returns the sum of the numbers in the list ns

```
(define (sum ns)
  (if (null? ns)
      0
      (+ (first ns)
         (sum (rest ns)))))
```


Understanding `sum`: Approach #1

`(sum (list 7 2 4))`



We'll call this the **recursive accumulation** pattern

Understanding `sum`: Approach #2

In `(sum (list 7 2 4))`, the list argument to `sum` is

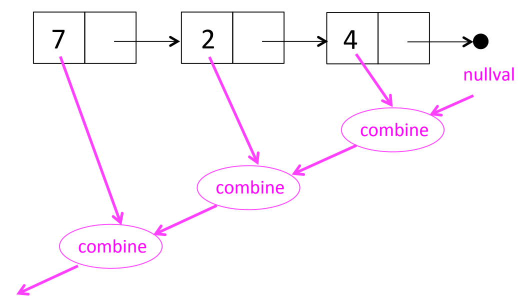
```
(cons 7 (cons 2 (cons 4 null))))
```

Replace `cons` by **+** and `null` by **0** and simplify:

```
(+ 7 (+ 2 (+ 4 0))))
⇒ (+ 7 (+ 2 4)))
⇒ (+ 7 6)
⇒ 13
```


Generalizing `sum`: Approach #1

```
(recf (list 7 2 4))
```



Generalizing `sum`: Approach #2

In `(recf (list 7 2 4))`, the list argument to `recf` is

```
(cons 7 (cons 2 (cons 4 null))))
```

Replace `cons` by **`combine`** and `null` by **`nullval`** and simplify:

```
(combine 7 (combine 2 (combine 4 nullval))))
```


Generalizing the `sum` definition

```
(define (recf ns)
  (if (null? ns)
      nullval
      (combine (first ns)
               (recf (rest ns)))))
```

Your turn

`(product ns)` returns the product of the numbers in `ns`

`(min-list ns)` returns the minimum of the numbers in `ns`
Hint: use `min` and `+inf.0` (positive infinity)

`(max-list ns)` returns the minimum of the numbers in `ns`
Hint: use `max` and `-inf.0` (negative infinity)

`(all-true? bs)` returns `#t` if all the elements in `bs` are truthy; otherwise returns `#f`. *Hint*: use `and`

`(some-true? bs)` returns a truthy value if at least one element in `bs` is truthy; otherwise returns `#f`. Hint: use `or`

`(my-length xs)` returns the length of the list `xs`

Recursive Accumulation Pattern Summary

	combine	nullval
sum	+	0
product	*	1
min-list	min	+inf.0
max-list	max	-inf.0
all-true?	and	#t
some-true?	or	#f
my-length	(λ (fst subres) (+ 1 subres))	0

Mapping Example: `map-double`

`(map-double ns)` returns a new list the same length as `ns` in which each element is the double of the corresponding element in `ns`.

```
> (map-double (list 7 2 4))
'(14 4 8)
```

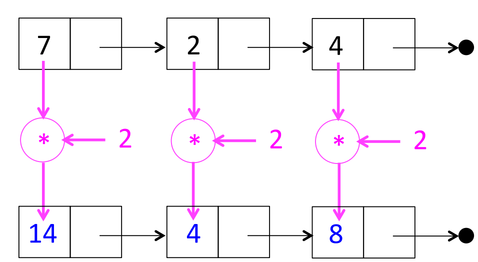
```
(define (map-double ns)
  (if (null? ns)
      ; Flesh out base case

      ; Flesh out recursive case

      ))
```


Understanding `map-double`

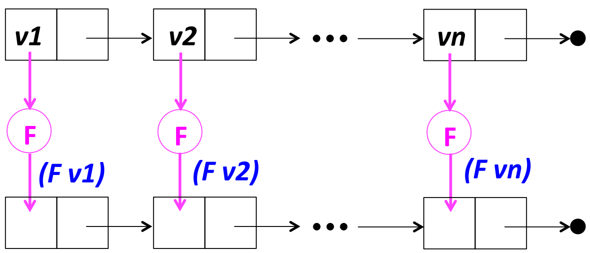
`(map-double (list 7 2 4))`



We'll call this the **mapping** pattern

Generalizing `map-double`

`(mapF (list v1 v2 … vn))`




```
(define (mapF xs)
  (if (null? xs)
      null
      (cons (F (first xs))
            (mapF (rest xs)))))
```


Expressing `mapF` as an accumulation

```
(define (mapF xs)
  (if (null? xs)
      null
      ((λ (fst subres)
                        ) ; Flesh this out
       (first xs)
       (mapF (rest xs)))))
```


Some Recursive Listfuns Need Extra Args

```
(define (map-scale factor ns)
  (if (null? ns)
      null
      (cons (* factor (first ns))
            (map-scale factor (rest ns)))))
```


Filtering Example: `filter-positive`

`(filter-positive ns)` returns a new list that contains only the positive elements in the list of numbers `ns`, in the same relative order as in `ns`.

```
> (filter-positive (list 7 -2 -4 8 5))
'(7 8 5)
```

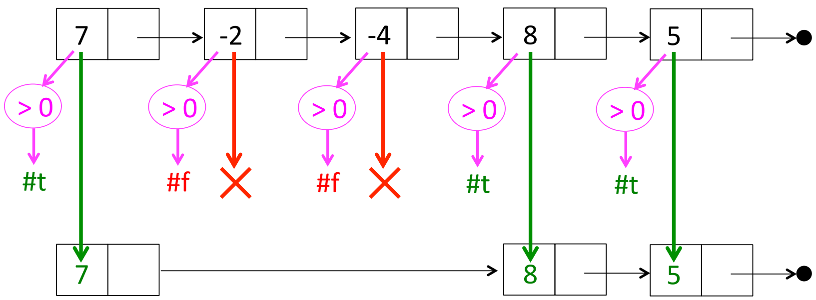
```
(define (filter-positive ns)
  (if (null? ns)
      ; Flesh out base case

      ; Flesh out recursive case

      ))
```

Understanding `filter-positive`

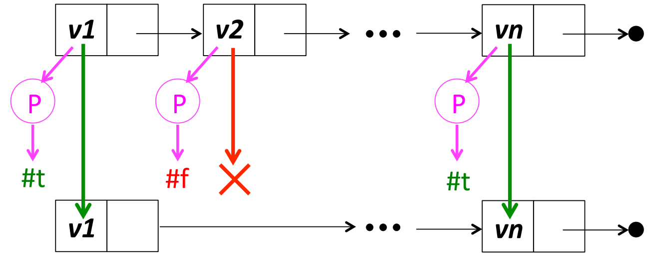
```
(filter-positive (list 7 -2 -4 8 5))
```



We'll call this the **filtering** pattern

Generalizing `filter-positive`

```
(filterP (list v1 v2 … vn))
```




```
(define (filterP xs)
  (if (null? xs)
      null
      (if (P (first xs))
          (cons (first xs) (filterP (rest xs)))
          (filterP (rest xs)))))
```


Expressing `filterP` as an accumulation

```
(define (filterP xs)
  (if (null? xs)
      null
      ((lambda (fst subres)


                             ) ; Flesh this out
       (first xs)
       (filterP (rest xs)))))
```


Your turn:
Define these using Divide/Conquer/Glue

```
> (snoc 11 '(7 2 4))
'(7 2 4 11)

> (my-append '(7 2 4) '(5 8))
'(7 2 4 5 8)

> (append-all '((7 2 4) (9) () (5 8)))
'(7 2 4 9 5 8)

> (my-reverse '(7 2 4))
'(4 2 7)
```