The Pros of \textit{cons}:
Programming with Pairs and Lists
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):** \( (\text{cons } v_1 \ v_2) \)

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

Can glue any number of values into a cons tree!
Box-and-pointer diagrams for \texttt{cons} trees

\[(\text{cons } v1 \ v2) \quad v1 \ v2\]

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

\[(\text{cons } (\text{cons } 17 \ (\text{cons } \text{"cat" } \#\text{\textbackslash a})))
\quad (\text{cons } \#\text t \ (\lambda (x) \ (* \ 2 \ x)))\]
Evaluation Rules for \texttt{cons}

Big step semantics:

\[
\begin{array}{c}
e_1 \downarrow v_1 \\
e_2 \downarrow v_2 \\
\hline
(\text{cons } e_1 e_2) \downarrow (\text{cons } v_1 v_2)
\end{array}
\]

Small-step semantics:

\[
(\text{cons } e_1 e_2) \\
\Rightarrow^* (\text{cons } v_1 e_2); \text{ first evaluate } e_1 \text{ to } v_1 \text{ step-by-step} \\
\Rightarrow^* (\text{cons } v_1 v_2); \text{ then evaluate } e_2 \text{ to } v_2 \text{ step-by-step}
\]
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
  
  \[(\text{car} \ (\text{cons} \ 7 \ 4)) \Rightarrow 7\]

- **cdr** extract the right value of a pair
  
  \[(\text{cdr} \ (\text{cons} \ 7 \ 4)) \Rightarrow 4\]

**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)))))
```

```
17
  ---
 #\a
  "cat"

#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:

\[
\begin{align*}
\text{e} & \downarrow (\text{cons } v1 \ v2) \\
\text{(car e)} & \downarrow v1 \\
\text{(cdr e)} & \downarrow v2
\end{align*}
\]

Small-step semantics:

\[
\begin{align*}
\text{(car e)} & \Rightarrow^* \text{(car (cons } v1 \ v2) ) ; \text{first evaluate e to pair step-by-step} \\
\text{\Rightarrow v1} & ; \text{then extract left value of pair} \\
\text{(cdr e)} & \Rightarrow^* \text{(car (cons } v1 \ v2) ) ; \text{first evaluate e to pair step-by-step} \\
\text{\Rightarrow v2} & ; \text{then extract right value of pair}
\end{align*}
\]
Semantics Puzzle

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

\[(\text{car} \ (\text{cons} \ (+ \ 2 \ 3) \ (* \ 5 \ #t)))\]

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

\[
\text{(define (swap-pair pair) }
  \text{(cons (cdr pair) (car pair)))}
\]

\[
\text{(define (sort-pair pair) }
  \text{(if (< (car pair) (cdr pair))}
    \text{pair}
    \text{(swap pair))})
\]

What are the values of these expressions?

- \((\text{swap-pair (cons 1 2)})\)
- \((\text{sort-pair (cons 4 7)})\)
- \((\text{sort-pair (cons 8 5)})\)
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:

Notation for null in box-and-pointer diagrams
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:

\[(\text{list } (+ 1 2) (* 3 4) (< 5 6))\]

\textbf{desugars to } \(\text{cons } (+ 1 2) (\text{cons } (* 3 4) (\text{cons } (< 5 6) \text{ null})))\)

\[\Rightarrow (\text{cons } 3 (\text{cons } (* 3 4) (\text{cons } (< 5 6) \text{ null})))\]

\[\Rightarrow (\text{cons } 3 (\text{cons } 12 (\text{cons } (< 5 6) \text{ null})))\]

\[\Rightarrow (\text{cons } 3 (\text{cons } 12 (\text{cons } \#t \text{ null})))\]

\textbf{resugars to } \(\text{list } 3 12 \#t\)

\textbf{Heck, let’s informally write this as:}

\[(\text{list } (+ 1 2) (* 3 4) (< 5 6))\]

\[\Rightarrow (\text{list } 3 (* 3 4) (< 5 6))\]

\[\Rightarrow (\text{list } 3 12 (\text{cons } (< 5 6)))\]

\[\Rightarrow (\text{list } 3 12 \#t)\]
first, rest, and friends

• **first** returns the first element of a list:
  
  \[
  \text{(first (list 7 2 4)) } \Rightarrow 7
  \]
  
  (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:
  
  \[
  \text{(rest (list 7 2 4)) } \Rightarrow \text{(list 2 4)}
  \]
  
  (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 \( \text{recf} \) on a list argument \( L \) has two cases:

- **base case:** what does \( \text{recf} \) return when \( L \) is empty? (Use \( \text{null?} \) to test for an empty list)

- **recursive case:** if \( L \) is the nonempty list \( \text{cons} \ v\_\text{first} \ v\_\text{rest} \) how are \( v\_\text{first} \) and \( \text{recf} \ v\_\text{rest} \) combined to give the result for \( \text{recf} \ L \)?

Note that we “blindly” apply \( \text{recf} \) to \( v\_\text{rest} \)!
Example: \textit{sum}

\textbf{(sum ns)} returns the sum of the numbers in the list \textit{ns}

\texttt{(define (sum ns)}
\texttt{  (if (null? ns)}
\texttt{    0}
\texttt{    (+ (first ns)}
\texttt{      (sum (rest ns)))))}
Understanding \texttt{sum}: Approach \#1

\texttt{(sum (list 7 2 4))}

We’ll call this the \texttt{recursive accumulation} pattern
Understanding \textbf{sum}: Approach #2

In \((\text{sum} \ (\text{list} \ 7 \ 2 \ 4))\), the list argument to \text{sum} is

\((\text{cons} \ 7 \ (\text{cons} \ 2 \ (\text{cons} \ 4 \ \text{null}))))\)

Replace \text{cons} by \text{+} and \text{null} by \text{0} and simplify:

\((+ \ 7 \ (+ \ 2 \ (+ \ 4 \ 0))))\)

\(\Rightarrow (+ \ 7 \ (+ \ 2 \ 4))\)

\(\Rightarrow (+ \ 7 \ 6)\)

\(\Rightarrow 13\)
Generalizing \texttt{sum}: Approach #1

\texttt{(recf (list 7 2 4))}
Generalizing \textbf{sum}: Approach #2

In \((\text{recf} \ (\text{list} \ 7 \ 2 \ 4))\), the list argument to \text{recf} is

\[(\text{cons} \ 7 \ (\text{cons} \ 2 \ (\text{cons} \ 4 \ \text{null})))\]

Replace \text{cons} by \textbf{combine} and \text{null} by \textbf{nullval} and simplify:

\[(\text{combine} \ 7 \ (\text{combine} \ 2 \ (\text{combine} \ 4 \ \text{nullval})))\]
Generalizing the \texttt{sum} definition

\begin{verbatim}
(define (recf ns)
  (if (null? ns)
      nullval
      (combine (first ns)
        (recf (rest ns))))
\end{verbatim}
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

<table>
<thead>
<tr>
<th></th>
<th>combine</th>
<th>nullval</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sum</strong></td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td><strong>product</strong></td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td><strong>min-list</strong></td>
<td>min</td>
<td>+inf.0</td>
</tr>
<tr>
<td><strong>max-list</strong></td>
<td>max</td>
<td>-inf.0</td>
</tr>
<tr>
<td><strong>all-true?</strong></td>
<td>and</td>
<td>#t</td>
</tr>
<tr>
<td><strong>some-true?</strong></td>
<td>or</td>
<td>#f</td>
</tr>
<tr>
<td><strong>my-length</strong></td>
<td>(λ (fst subres) (+ 1 subres))</td>
<td>0</td>
</tr>
</tbody>
</table>
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 \textit{map-double}

\texttt{(map-double (list 7 2 4))}

We’ll call this the \textit{mapping} pattern
Generalizing map-double

\((\text{map}^F (\text{list} \ v1 \ v2 \ \ldots \ \text{vn}))\)

\[
\begin{align*}
\text{v1} & \rightarrow \text{v2} \rightarrow \cdots \rightarrow \text{vn} \\
(F \ v1) & \rightarrow (F \ v2) \rightarrow (F \ \text{vn})
\end{align*}
\]

\[
(\text{define} \ (\text{map}^F \ \text{xs}) \\
(\text{if} \ (\text{null?} \ \text{xs}) \\
 \ \ \ \text{null} \\
 \ \ \ \ (\text{cons} \ ((F \ (\text{first} \ \text{xs}))) \\
 \ \ \ \ (\text{map}^F \ (\text{rest} \ \text{xs}))))))
\]
Expressing \texttt{map}\texttt{F} as an accumulation

\begin{verbatim}
(define (map\texttt{F} xs)
  (if (null? xs)
      null
      ((\lambda (fst subres)
         ; Flesh this out
         (first xs)
         (map\texttt{F} (rest xs))))))
\end{verbatim}
(define (map-scale factor ns)
  (if (null? ns)
      null
      (cons (* factor (first ns))
            (map-scale factor (rest ns)))))

Some Recursive Listfuns Need Extra Args
Filtering Example: \texttt{filter-positive}

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

\texttt{> (filter-positive (list 7 -2 -4 8 5))}
\texttt{'(7 8 5)}

\begin{verbatim}
(define (filter-positive ns)
  (if (null? ns)
      ;; Flesh out base case
      ;; Flesh out recursive case
      ))
\end{verbatim}
Understanding **filter-positive**

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

We’ll call this the **filtering** pattern
Generalizing filter-positive

\[(\text{filter}_P \ (\text{list} \ v_1 \ v_2 \ \ldots \ \ v_n))\]

\[
\text{(define (filter}_P \ \text{xs}) \\
\quad \text{(if (null? \ xs) \\
\quad \null \\
\quad \text{(if (P (first \ xs)) \\
\quad \quad \text{(cons (first \ xs) (filter}_P \ \text{(rest \ xs))}}) \\
\quad \text{(filter}_P \ \text{(rest \ xs))})) \\
\text{)}}
\]
Expressing \texttt{filterP} as an accumulation

\begin{verbatim}
(define (filterP xs)
  (if (null? xs)
      null
      ((lambda (fst subres)
        ) ; Flesh this out
        (first xs)
        (filterP (rest xs)))))
\end{verbatim}
More examples

• snoc/postpend
• append
• append-all
• sorted?
• merge