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

---

CS251 Programming Languages
Spring 2017, Lyn Turbak
Department of Computer Science
Wellesley College

---

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

---

**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?

---

**Box-and-pointer diagrams for **cons** trees**

`{(cons v1 v2) 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))))
```

```
(cons v1 v2)
  \a
  #t
  "cat"
  λ x (* 2 x)
```
**Evaluation Rules for cons**

**Big step semantics:**

\[
\begin{align*}
E_1 & \downarrow V_1 \\
E_2 & \downarrow V_2 \\
(\text{cons } E_1 E_2) & \downarrow (\text{cons } V_1 V_2) \\
\end{align*}
\]

**Small-step semantics:**

*cons* has no special evaluation rules. Its two operands are evaluated left-to-right until a value \((\text{cons } V_1 V_2)\) is reached:

\[
(\text{cons } E_1 E_2) \\
\Rightarrow^* (\text{cons } V_1 \{E_2\}) \; \text{first evaluate } E_1 \text{ to } V_1 \text{ step-by-step} \\
\Rightarrow^* (\text{cons } V_1 V_2) \; \text{then evaluate } E_2 \text{ to } V_2 \text{ step-by-step}
\]

**cons evaluation example**

\[
(\text{cons} (\text{cons} \{(\text{+ 1 2})\} \{< 3 4\}) \{\text{(} > 5 6 \text{)} \{(\text{* 7 8)}\} \}) \\
\Rightarrow (\text{cons} (\text{cons} 3 \{\{< 3 4\}\}) \{\text{(} > 5 6 \text{)} \{(\text{* 7 8)}\} \}) \\
\Rightarrow (\text{cons} (\text{cons} 3 \#t) \{\text{(} > 5 6 \text{)} \{(\text{* 7 8)}\} \}) \\
\Rightarrow (\text{cons} (\text{cons} 3 \#t) \{\text{\#f} \{(\text{* 7 8)}\} \}) \\
\Rightarrow (\text{cons} (\text{cons} 3 \#t) \{\text{\#f 56}\})
\]

**Pairs and Lists**

**car and cdr**

- **car** extracts the left value of a pair
  
  \[
  (\text{car} \ (\text{cons} \ 7 \ 4)) \Rightarrow 7
  \]

- **cdr** extracts 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:

\[
\text{(define tr} \ (\text{cons} \ (\text{cons} \ 17 \ (\text{cons} \ "\text{cat}" \ #\text{a}) \ {\text{\#t}} \ (\lambda \ (x) \ {(\text{*} 2 \ x)}) \})) \\
\text{tr} \leftrightarrow (\text{cons} \ (\text{cons} \ 17 \ (\text{cons} \ "\text{cat}" \ #\text{a}) \ {\text{\#t}} \ (\lambda \ (x) \ {(\text{*} 2 \ x)}) \))
\]

\[
\text{tr} \leftrightarrow (\text{cons} \ (\text{cons} \ 17 \ (\text{cons} \ "\text{cat}" \ #\text{a}) \ {\text{\#t}} \ (\lambda \ (x) \ {(\text{*} 2 \ x)}) \))
\]

17

#\text{a}

"cat"

\[
\text{tr} \leftrightarrow (\text{cons} \ (\text{cons} \ 17 \ (\text{cons} \ "\text{cat}" \ #\text{a}) \ {\text{\#t}} \ (\lambda \ (x) \ {(\text{*} 2 \ x)}) \))
\]

#\text{t}

\[
\text{tr} \leftrightarrow (\text{cons} \ (\text{cons} \ 17 \ (\text{cons} \ "\text{cat}" \ #\text{a}) \ {\text{\#t}} \ (\lambda \ (x) \ {(\text{*} 2 \ x)}) \))
\]

\[
\text{tr} \leftrightarrow (\text{cons} \ (\text{cons} \ 17 \ (\text{cons} \ "\text{cat}" \ #\text{a}) \ {\text{\#t}} \ (\lambda \ (x) \ {(\text{*} 2 \ x)}) \))
\]

\[
\text{tr} \leftrightarrow (\text{cons} \ (\text{cons} \ 17 \ (\text{cons} \ "\text{cat}" \ #\text{a}) \ {\text{\#t}} \ (\lambda \ (x) \ {(\text{*} 2 \ x)}) \))
\]
**cadr and friends**

- \( (\text{caar } e) \) means \( (\text{car } (\text{car } e)) \)
- \( (\text{cadr } e) \) means \( (\text{car } (\text{cdr } e)) \)
- \( (\text{cdar } e) \) means \( (\text{cdr } (\text{car } e)) \)
- \( (\text{cddr } e) \) means \( (\text{cdr } (\text{cdr } e)) \)
- \( (\text{caaar } e) \) means \( (\text{car } (\text{car } (\text{car } e))) \)
- \( (\text{cddddd } e) \) means \( (\text{cdr } (\text{cdr } (\text{cdr } (\text{cdr } e)))) \)

**Evaluation Rules for car and cdr**

**Big-step semantics:**

\[
\begin{align*}
E \downarrow (\text{cons } V1 V2) & \quad \text{[car]} \\
(\text{car } E) \downarrow V1 & \\
(\text{cdr } e) \downarrow v2 & \quad \text{[cdr]}
\end{align*}
\]

**Small-step semantics:**

\[
\begin{align*}
(\text{car } (\text{cons } V1 V2)) & \Rightarrow V1 \quad \text{[car]} \\
(\text{cdr } (\text{cons } V1 V2)) & \Rightarrow V2 \quad \text{[cdr]}
\end{align*}
\]

**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**

\[
\begin{align*}
> & \ (\text{lambda } (x) (* x 2)) \\
& \ #<\text{procedure}> \\
> & \ (\text{cons } (+ 1 2) (* 3 4)) \\
& \ '((3 . 12) \\
> & \ (\text{cons } (5 6) (\text{cons } 7 8)) \\
& \ '((5 . 6) 7 . 8) \\
> & \ (\text{cons } 1 (\text{cons } 2 (\text{cons } 3 4))) \\
& \ '((1 \ 2 \ 3 . 4))
\end{align*}
\]

What’s going on here?
Display Notation and Dotted Pairs

- The **display notation** for \((\text{cons } V_1 \ V_2)\) is \((D_1N \ . \ D_2N)\), where \(D_1N\) and \(D_2N\) are the display notations for \(V_1\) and \(V_2\).
- In display notation, a dot “eats” a paren pair that follows it directly:
  1. \((5 \ . \ 6) \ (7 \ . \ 8))\)
     becomes \(((5 \ . \ 6) \ 7 \ . \ 8)\)
   2. \((1 \ . \ (2 \ . \ (3 \ . \ 4)))\)
     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.)

### Functions Can Take and Return Pairs

```racket
(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?

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

Lists

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

- A list is either
  - The empty list \(\text{null}\), whose display notation is ()
  - A nonempty list \((\text{cons } V_{\text{first}} \ V_{\text{rest}})\) whose
    - first element is \(V_{\text{first}}\)
    - and the rest of whose elements are the sublist \(V_{\text{rest}}\)

  E.g., a list of the 3 numbers 7, 2, 4 is written
  
  
  \((\text{cons } 7 \ (\text{cons } 2 \ (\text{cons } 4 \ \text{null}))))\)
### Box-and-pointer notation for lists

A list of n values is drawn like this:

A pair slot containing null can also be with a slash through the slot.

For example:

```
7  2  4  \rightarrow  \cdot
```

### Notation for null in box-and-pointer diagrams

```
V1  \rightarrow  V2  \rightarrow  \cdots  \rightarrow  Vn  \rightarrow  \cdot
```

### 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 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

### 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)))
- desugars to (cons 3 (cons ((+ 3 4)) (cons (< 5 6) null)))
- desugars to (cons 3 (cons 12 (cons ((< 5 6)) null)))
- desugars to (cons 3 (cons 12 (cons #t null)))

resugars to (list 3 12 #t)

 Heck, let’s just informally write this as:

```
(list ((+ 1 2)) (* 3 4) (< 5 6))
```

- desugars to (list 3 ((+ 3 4)) (< 5 6))
- desugars to (list 3 12 ((< 5 6)))
- desugars to (list 3 12 #t)
first, rest, and friends

- first returns the first element of a list:
  \[ \text{first} \ (\text{list} \ 7 \ 2 \ 4) \]  \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:
  \[ \text{rest} \ (\text{list} \ 7 \ 2 \ 4) \]  \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 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 Vfirst Vrest) how are Vfirst and (recf Vrest) combined to give the result for (recf L)?

Note that we always "blindly" apply recf to Vrest!

Recursive List Functions: Divide/Conquer/Glue (DCG) strategy for the general case [in words]

**Step 1 (concrete example):** pick a concrete input list, typically 3 or 4 elements long. What should the function return on this input?

E.g. A sum function that returns the sum of all the numbers in a list:

\[ \text{(sum '}(5 \ 7 \ 2 \ 4)) \]  \Rightarrow \!* \ 18

**Step 2 (divide):** without even thinking, always apply the function to the rest of the list. What does it return?

\[ \text{(sum '}(7 \ 2 \ 4)) \]  \Rightarrow \!* \ 13

**Step 3 (glue):** How to combine the first element of the list (in this case, 5) with the result from processing the rest (in this case, 13) to give the result for processing the whole list (in this case, 18)? 5 + \text{(sum '}(7 \ 2 \ 4)) \]  \Rightarrow \!* \ 18

**Step 4 (generalize):** Express the general case in terms of an arbitrary input:

\[
\text{(define (sum nums)} \hfill ...
\text{(+ (first nums) (sum (rest nums)) ...) )}
\]

Recursive List Functions: Divide/Conquer/Glue (DCG) strategy for the general case [in diagram]

\[
\text{(sum '}(5 \ 7 \ 2 \ 4)) \]  \Rightarrow \!* \ 18
\]

\[
\text{Divide: what should function return for rest of list? (wishful thinking!)}
\]

\[
\text{(sum '}(7 \ 2 \ 4)) \]  \Rightarrow \!* \ 13
\]

\[
\text{Glue: how to combine the first element of the list with the result of recursively processing rest of the list to get the desired result for the whole list?}
\]

\[
\text{Solution for concrete example: 5 + (sum '}(7 \ 2 \ 4))
\]

\[
\text{Generalization of concrete solution: (first nums) + (sum (rest nums))}
\]
Recursive List Functions: base case via singleton case

Deciding what a recursive list function should return for the empty list is not always obvious and can be tricky. E.g. what should \( \text{sum ('())} \) return?

If the answer isn’t obvious, consider the “penultimate case” in the recursion, which involves a list of one element:

\[
\text{sum ('(4))} \Rightarrow 4
\]

In this case, \textbf{Vnull} should be 0, which is the identity element for addition.

But in general it depends on the details of the particular combiner determined from the general case. So solve the general case before the base case!

Putting it all together: base & general cases

\( \text{sum nums} \) returns the sum of the numbers in the list \( \text{nums} \)

\[
\text{(define (sum ns)}
\text{  (if (null? ns)
  0
  (+ (first ns)
    (sum (rest ns)))))}
\]

Understanding \textit{sum}: Approach #1

\( \text{(sum '(7 2 4))} \)

We’ll call this the \textit{recursive accumulation} pattern

Understanding \textit{sum}: Approach #2

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

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

Replace cons by + and null by 0 and simplify:

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

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

\[
\Rightarrow (+ 7 6)
\]

\[
\Rightarrow 13
\]
Generalizing \textit{sum}: Approach \#1

\begin{center}
\begin{tabular}{c}
\texttt{(recf (list 7 2 4))}
\end{tabular}
\end{center}

In \texttt{(recf (list 7 2 4))}, the list argument to \texttt{recf} is
\begin{center}
\begin{tabular}{c}
\texttt{(cons 7 (cons 2 (cons 4 null)))}
\end{tabular}
\end{center}

Replace \texttt{cons} by \texttt{combine} and \texttt{null} by \texttt{nullval} and simplify:
\begin{center}
\begin{tabular}{c}
\texttt{(combine 7 (combine 2 (combine 4 nullval)))}
\end{tabular}
\end{center}

Generalizing the \textit{sum} definition

\begin{center}
\begin{tabular}{c}
\texttt{(define (recf ns)}
\texttt{ (if (null? ns)}
\texttt{ \texttt{nullval}}
\texttt{ \texttt{(combine (first ns)}
\texttt{ \texttt{(recf (rest ns)))))}}
\end{tabular}
\end{center}

Your turn

Define the following recursive list functions and test them in Racket:

\begin{itemize}
\item \texttt{(product ns)} returns the product of the numbers in \texttt{ns}
\item \texttt{(min-list ns)} returns the minimum of the numbers in \texttt{ns}
\textit{Hint:} use \texttt{min} and \texttt{+inf.0} (positive infinity)
\item \texttt{(max-list ns)} returns the minimum of the numbers in \texttt{ns}
\textit{Hint:} use \texttt{max} and \texttt{-inf.0} (negative infinity)
\item \texttt{(all-true? bs)} returns \#t if all the elements in \texttt{bs} are truthy; otherwise returns \#f. \textit{Hint:} use \texttt{and}
\item \texttt{(some-true? bs)} returns a truthy value if at least one element in \texttt{bs} is truthy; otherwise returns \#f. \textit{Hint:} use \texttt{or}
\item \texttt{(my-length xs)} returns the length of the list \texttt{xs}
\end{itemize}
### Recursive Accumulation Pattern Summary

<table>
<thead>
<tr>
<th></th>
<th>combine</th>
<th>nullval</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>product</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>min-list</td>
<td>min</td>
<td>+inf.0</td>
</tr>
<tr>
<td>max-list</td>
<td>max</td>
<td>-inf.0</td>
</tr>
<tr>
<td>all-true?</td>
<td>and</td>
<td>#t</td>
</tr>
<tr>
<td>some-true?</td>
<td>or</td>
<td>#f</td>
</tr>
<tr>
<td>my-length</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 general case
 )

### Understanding map-double

(map-double '(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))))

Pairs and Lists 33

Pairs and Lists 34

Pairs and Lists 35

Pairs and Lists 36
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`.

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

Understanding filter-positive

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

We’ll call this the filtering pattern
```
Generalizing \texttt{filter-positive}

\begin{equation*}
\text{(filterP (list } V1 \ V2 \ldots \ Vn))
\end{equation*}

\begin{equation*}
\text{(define (filterP xs)}
\begin{array}{l}
(\text{if (null? xs)}
\begin{array}{l}
\text{null}
\end{array}
\begin{array}{l}
(\text{if (P (first xs)}
\begin{array}{l}
(\text{cons (first xs) (filterP (rest xs))})
\end{array}
\begin{array}{l}
\begin{array}{l}
\text{(filterP (rest xs)))})
\end{array}
\end{array}
\end{array}
\end{array}
\end{equation*}

\text{Your turn:}
\begin{equation*}
\text{Define these using Divide/Conquer/Glue}
\end{equation*}

\begin{itemize}
\item \texttt{(snoc 11 '(7 2 4))}
\texttt{'(7 2 4 11)}
\item \texttt{(my-append '(7 2 4) '(5 8))}
\texttt{'(7 2 4 5 8)}
\item \texttt{(append-all '((7 2 4) (9) () (5 8)))}
\texttt{'(7 2 4 9 5 8)}
\item \texttt{(my-reverse '(5 7 2 4))}
\texttt{'(4 2 7 5)}
\end{itemize}