The Pros of cons: Programming with Pairs and Lists

CS251 Programming Languages
Fall 2017, 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\nnto 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 pointers to “large” values (functions, strings, pairs) outside a box.

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

In Racket, type Command-\ to get λ char

17 #\a "cat" #t

(λ (x) (* 2 x))
Evaluation Rules for `cons`

Big step semantics:

\[
\begin{array}{c}
E1 \downarrow V1 \\
E2 \downarrow V2 \\
(\text{cons } E1 E2) \downarrow (\text{cons } V1 V2)
\end{array}
\]

Small-step semantics:

`cons` has no special evaluation rules. Its two operands are evaluated left-to-right until a value `(cons V1 V2)` is reached:

\[
\begin{align*}
(\text{cons } E1 E2) \\
\Rightarrow^* (\text{cons } V1 \{E2\}) \; \text{; first evaluate } E1 \text{ to } V1 \text{ step-by-step} \\
\Rightarrow^* (\text{cons } V1 V2) \; \text{; then evaluate } E2 \text{ to } V2 \text{ step-by-step}
\end{align*}
\]

```
Pairs and Lists
5
```

```
Pairs and Lists
6
```

```
Pairs and Lists
7
```

```
Pairs and Lists
8
```

**cons evaluation example**

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

**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”

```
Pairs and Lists
5
```

```
Pairs and Lists
6
```

```
Pairs and Lists
7
```

```
Pairs and Lists
8
```

**Practice with car and cdr**

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

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

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
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 (car 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:

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

Small-step semantics:

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

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?

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!
Display Notation and Dotted Pairs

- The display notation for \((\text{cons} \ V1 \ V2)\) 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.)

Functions Can Take and Return Pairs

```scheme
(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 null, whose display notation is ()
- A nonempty list \((\text{cons} \ Vfirst \ Vrest)\) whose
  - first element is \(Vfirst\)
  - and the rest of whose elements are the sublist \(Vrest\)

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:

V1 → V2 → ... → Vn → \[∵\]

Notation for null in box-and-pointer diagrams

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

For example:

7 → 2 → 4 → \[∵\]

7 → 2 → 4 /

**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)))

⇒ (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 just informally write this as:

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

⇒ (list 3 (((* 3 4)) (< 5 6))

⇒ (list 3 12 (((< 5 6))

⇒ (list 3 12 #t)
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 \( (\text{cons} \ V\text{first} \ V\text{rest}) \) how are \( V\text{first} \) and \( (\text{recf} \ V\text{rest}) \) combined to give the result for \((\text{recf} \ L)\)?

Note that we 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 \text{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:
\[
(\text{define} \ (\text{sum} \ \text{nums}))
\quad ... \ (+ \ (\text{first} \ \text{nums}) \ (\text{sum} \ (\text{rest} \ \text{nums}))) \quad ...)
\]

Recursive List Functions: Divide/Conquer/Glue (DCG)

strategy for the general case [in diagram]
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:

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

Divide: what value \(V\text{null}\) should function return for empty list?

\[
(\text{sum } '()) \Rightarrow ^* V\text{null}
\]

In this case, \(V\text{null}\) 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 \(\text{nums}\)

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

Understanding \(\text{sum}\): Approach #1

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

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

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

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

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

\[
\Rightarrow (+ 7 6)
\]

\[
\Rightarrow 13
\]

We’ll call this the recursive accumulation pattern

Understanding \(\text{sum}\): Approach #2
Generalizing \textit{sum}: Approach #1

\begin{verbatim}
(recf (list 7 2 4))
\end{verbatim}

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

\begin{verbatim}
(cons 7 (cons 2 (cons 4 null)))
\end{verbatim}

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

\begin{verbatim}
(combine 7 (combine 2 (combine 4 nullval)))
\end{verbatim}

Generalizing the \textit{sum} definition

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

Your turn

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

- \texttt{(product ns)} returns the product of the numbers in \texttt{ns}
- \texttt{(min-list ns)} returns the minimum of the numbers in \texttt{ns}
  \textit{Hint: use} \texttt{min} \textit{and} \texttt{+inf.0} (positive infinity)
- \texttt{(max-list ns)} returns the minimum of the numbers in \texttt{ns}
  \textit{Hint: use} \texttt{max} \textit{and} \texttt{-inf.0} (negative infinity)
- \texttt{(all-true? bs)} returns \#t if all the elements in \texttt{bs} are truthy; otherwise returns \#f. \textit{Hint: use} \texttt{and}
- \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}
- \texttt{(my-length xs)} returns the length of the list \texttt{xs}
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)

```scheme
(define (map-double ns)
  (if (null? ns)
      #f
      (cons (F v1) (mapF rest ns)))))
```

Understanding map-double

(map-double '(7 2 4))

We’ll call this the mapping pattern

Generalizing map-double

(mapF (list V1 V2 ... Vn))

```scheme
(define (mapF xs)
  (if (null? xs)
      null
      (cons (F (first xs)) (mapF (rest xs)))))
```
Expressing \( \text{mapF} \) as an accumulation

\[
\begin{align*}
(\text{define } (\text{mapF} \ \text{xs}) \\
(\text{if } (\text{null?} \ \text{xs}) \\
\quad \text{null} \\
\quad ((\lambda (\text{fst} \ \text{subres}) \\
\quad \quad ; \text{Flesh this out} \\
\quad \quad (\text{first} \ \text{xs}) \\
\quad \quad (\text{mapF} \ (\text{rest} \ \text{xs}))))))
\end{align*}
\]

Some Recursive Listfuns Need Extra Args

\[
(\text{define } (\text{map-scale} \ \text{factor} \ \text{ns}) \\
(\text{if } (\text{null?} \ \text{ns}) \\
\quad \text{null} \\
\quad (\text{cons} \ (* \ \text{factor} \ (\text{first} \ \text{ns})) \\
\quad \quad (\text{map-scale} \ \text{factor} \ (\text{rest} \ \text{ns})))))
\]

Filtering Example: \text{filter-positive}

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

\[
\begin{align*}
> (\text{filter-positive} \ (\text{list} \ 7 \ -2 \ -4 \ 8 \ 5)) \\
'(7 \ 8 \ 5)
\end{align*}
\]

\[
(\text{define } (\text{filter-positive} \ \text{ns}) \\
(\text{if } (\text{null?} \ \text{ns}) \\
\quad ; \text{Flesh out base case} \\
\quad ; \text{Flesh out recursive case} \\
\quad ))
\]

Understanding \text{filter-positive}

\( (\text{filter-positive} \ (\text{list} \ 7 \ -2 \ -4 \ 8 \ 5)) \)

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

\begin{align*}
\text{(filterP} & \text{ (list } V_1 \ V_2 \ldots \ V_n) \\
\text{filterP} & \text{ (list } \ldots )
\end{align*}

Expressing \textit{filterP} as an accumulation

\begin{align*}
\text{(define (filterP} \text{ xs}) \\
\text{null} & \text{ null} \\
\text{(lambda (fst subres)} & \text{ (filterP (rest xs)))})
\end{align*}

Your turn:
Define these using Divide/Conquer/Glue

\begin{align*}
> & \text{(snoc 11 } '(7 2 4)) \\
& '(7 2 4 11) \\
> & \text{(my-append } '(7 2 4) \ '(5 8)) \\
& '(7 2 4 5 8) \\
> & \text{(append-all } '((7 2 4) 9) 5 8)) \\
& '(7 2 4 9 5 8) \\
> & \text{(my-reverse } '(5 7 2 4)) \\
& '(4 2 7 5)
\end{align*}