The Pros of \texttt{cons:} Programming with Pairs and Lists

CS251 Programming Languages
Fall 2016, Lyn Turbak
Department of Computer Science
Wellesley College

\texttt{cons} Glues Two Values into a Pair

A new kind of value:
\begin{itemize}
  \item \texttt{pairs (a.k.a. cons cells):} \((\texttt{cons } v1 \ v2)\)
    \begin{itemize}
      \item \((\texttt{cons 17 42})\)
      \item \((\texttt{cons 3.14159 \#t})\)
      \item \((\texttt{cons "CS251" (\lambda(x) \(*\ 2\ x\))})\)
      \item \((\texttt{cons (cons 3 4.5) (cons \#f \#\alpha)})\)
    \end{itemize}
\end{itemize}

Can glue any number of values into a \texttt{cons} tree!

Racket Values

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

Note: some are \textit{exact}, the rest are \textit{inexact}. See docs.

- \texttt{strings:} "cat", "CS251", "αβγ", "To be\ or not\ into"
- \texttt{characters:} \#\alpha, \#\A, \#\\5, \#space, \#\tab, \#\newline
- \texttt{anonymous functions:} \((\lambda a b (\texttt{\texttt{\texttt{(lambda}} a b (\texttt{\texttt{\texttt{+}} a \ (** b c))}}))\)

What about compound data?

Box-and-pointer diagrams for \texttt{cons} trees

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

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

\[(\texttt{cons (cons 17 (cons "cat" \#\alpha)) (cons \#t (\lambda(x) (** 2 x))))}\]

\[
\begin{array}{c}
17 \\
\#\alpha \\
"cat" \\
(\lambda(x)(**2x)) \rightarrow \#t
\end{array}
\]
Evaluation Rules for \texttt{cons}

Big step semantics:

\[
\begin{align*}
\text{e}_1 & \downarrow v_1 \\
\text{e}_2 & \downarrow v_2 \\
(\text{cons } \text{e}_1 \text{ e}_2) & \downarrow (\text{cons } v_1 \ v_2)
\end{align*}
\]

Small-step semantics:

\[(\text{cons } \text{e}_1 \text{ e}_2) \\
\Rightarrow^* (\text{cons } v_1 \text{ e}_2); \text{ first evaluate } \text{e}_1 \text{ to } v_1 \text{ step-by-step} \\
\Rightarrow^* (\text{cons } v_1 \ v_2); \text{ then evaluate } \text{e}_2 \text{ to } v_2 \text{ step-by-step} \]

\textbf{car and cdr}

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

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

Why these names?

- \textbf{car} from “contents of address register”
- \textbf{cdr} from “contents of decrement register”

\textbf{cons evaluation example}

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

\textbf{Practice with car and cdr}

Write expressions using \texttt{car}, \texttt{cdr}, and \texttt{tr} that extract the five leaves of this tree:

\[
\begin{align*}
(\text{define } \text{tr} \\
(\text{cons } (\text{cons } 17 \ (\text{cons } "\text{cat" } \ #\text{a}))) \\
(\text{cons } #t \ (\lambda \ (x) \ (* \ 2 \ x))))
\end{align*}
\]
**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 \((\text{cdr } (\text{cdr } (\text{car } e))))\)

**Evaluation Rules for car and cdr**

Big-step semantics:

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

Small-step semantics:

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

**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)`
  - 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

```
(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:

\[
\begin{array}{c}
v1 \\
\rightarrow \\
v2 \\
\rightarrow \\
\cdots \\
\rightarrow \\
v_n
\end{array}
\]

For example:

\[
\begin{array}{c}
7 \\
\rightarrow \\
2 \\
\rightarrow \\
4 \\
\rightarrow
\end{array}
\]

Display Notation for Lists

The “dot eats parens” rule makes lists display nicely:

\[
\text{(list 7 2 4)}
\]

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

\[
\text{displays as (before rule) (7 . (2 . (4 . ())))}
\]

\[
\text{displays as (after rule) (7 2 4)}
\]

\[
\text{prints as \'(7 2 4)}
\]

In Racket:

\[
> \text{(display (list 7 2 4))}
\]

\[
(7 2 4)
\]

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

\[
(7 2 4)
\]

list sugar

Treat list as syntactic sugar:

- (list) desugars to null
- (list \(e_1\) ...) desugars to (cons \(e_1\) (list ...))

For example:

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

\[
\begin{array}{l}
\text{desugars to} \quad \text{(cons (+ 1 2) (list (* 3 4) (< 5 6)))}
\end{array}
\]

\[
\begin{array}{l}
\text{desugars to} \quad \text{(cons (+ 1 2) (cons (* 3 4) (list (< 5 6))))}
\end{array}
\]

\[
\begin{array}{l}
\text{desugars to} \quad \text{(cons (+ 1 2) (cons (* 3 4) (cons (< 5 6) (list)))))}
\end{array}
\]

\[
\begin{array}{l}
\text{desugars to} \quad \text{(cons (+ 1 2) (cons (* 3 4) (cons (< 5 6) null)))}
\end{array}
\]

* 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:

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

\[
\begin{array}{l}
\text{desugars to} \quad \text{(cons (+ 1 2) (cons (* 3 4) (cons (< 5 6) null)))}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow \quad \text{(cons 3 (cons (* 3 4) (cons (< 5 6) null)))}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow \quad \text{(cons 3 (cons 12 (cons (< 5 6) null)))}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow \quad \text{(cons 3 (cons 12 (cons #t null)))}
\end{array}
\]

\[
\begin{array}{l}
\text{resugars to} \quad \text{(list 3 12 #t)}
\end{array}
\]

Heck, let’s informally write this as:

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

\[
\begin{array}{l}
\Rightarrow \quad \text{(list 3 (* 3 4) (< 5 6))}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow \quad \text{(list 3 12 (cons (< 5 6))}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow \quad \text{(list 3 12 #t)}
\end{array}
\]
**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 \( \text{recf} \) 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: sum**

\[(\text{sum} \ ns) \text{ returns the sum of the numbers in the list } ns\]

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

---

**Understanding sum: Approach #1**

\[(\text{sum} \ (\text{list} \ 7 \ 2 \ 4))\]

\[
7 \rightarrow 2 \rightarrow 4 \rightarrow \bullet
\]

\[
+ \ 13
\]

\[
+ \ 6
\]

\[
+ \ 4
\]

\[
+ \ 0
\]

We’ll call this the **recursive accumulation** pattern
Understanding \textit{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 \texttt{+} and \text{null} by \texttt{0} and simplify:
\[
\begin{align*}
&\text{(+} \ 7 \ \text{(+} \ 2 \ \text{(+} \ 4 \ 0)\text{))} \\
\Rightarrow &\text{(+} \ 7 \ \text{(+} \ 2 \ 4)\text{)} \\
\Rightarrow &\text{(+} \ 7 \ 6\text{)} \\
\Rightarrow &13
\end{align*}
\]

Generalizing \textit{sum}: Approach \#1

\((\text{recf} \ (\text{list} \ 7 \ 2 \ 4))\)

\begin{equation}
\begin{tikzpicture}
\draw[->] (0,0) -- (2,0) node[midway, above] {7};
\draw[->] (2,0) -- (4,0) node[midway, above] {2};
\draw[->] (4,0) -- (6,0) node[midway, above] {4};
\end{tikzpicture}
\end{equation}

Generalizing \textit{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 \texttt{combine} and \text{null} by \texttt{nullval} and simplify:
\[
\begin{align*}
&\text{combine} \ 7 \ (\text{combine} \ 2 \ (\text{combine} \ 4 \ \text{nullval}))\text{)}
\end{align*}
\]

Generalizing the \textit{sum} definition

\begin{verbatim}
(define (recf ns)
  (if (null? ns)
      \texttt{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>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 recursive case
    ))

Understanding map-double

(map-double (list 7 2 4))

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

\( \text{mapF} \ (\text{list } v_1 \ v_2 \ \ldots \ v_n) \)

\begin{align*}
& v_1 \quad \text{---} \quad v_2 \quad \text{---} \quad \cdots \quad v_n \quad \text{---} \\
& \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
& \quad F \quad F \quad F \\
& (F \ v_1) \quad (F \ v_2) \quad (F \ v_n) \\
& \quad \text{---} \quad \text{---} \quad \text{---} \\
\end{align*}

\begin{verbatim}
(define (mapF xs)
  (if (null? xs)
      null
      (cons (F (first xs))
            (mapF (rest xs)))))
\end{verbatim}

Pairs and Lists 33

Expressing \textit{mapF} as an accumulation

\begin{verbatim}
(define (mapF xs)
  (if (null? xs)
      null
      ((λ (fst subres)
         ; Flesh this out
         (first xs)
         (mapF (rest xs))))))
\end{verbatim}

Pairs and Lists 34

Some Recursive Listfuns Need Extra Args

\begin{verbatim}
(define (map-scale factor ns)
  (if (null? ns)
      null
      (cons (* factor (first ns))
            (map-scale factor (rest ns)))))
\end{verbatim}

Pairs and Lists 35

Filtering Example: \textit{filter-positive}

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

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

Pairs and Lists 36
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))

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)