The Pros of cons:
Pairs and Lists in Racket

**SOLUTIONS**

CS251 Programming Languages
Spring 2019, 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):
    - `2.3e2`, `3.141592653589793`
    - complex: `3+2i`, `17-23i`, `1.4129i`
  
Note: some are exact, the rest are inexact. See docs.

- strings: `"cat"`, `"CS251"`, `"αβγ"`
- 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))`
- 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 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 \texttt{cons}

**Big step semantics:**

\[
\begin{array}{ll}
E1 & \downarrow V1 \\
E2 & \downarrow V2 \\
(\texttt{cons } E1 E2) & \downarrow (\texttt{cons } V1 V2)
\end{array}
\]

**Small-step semantics:**

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

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

---

\texttt{Pairs and Lists}

\texttt{car} and \texttt{cdr}

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

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

Why these names?

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

---

\texttt{cons evaluation example}

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

---

\texttt{Practice with car and cdr Solutions}

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

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

\[
\texttt{tr} \mapsto \ (\texttt{cons } (\texttt{cons } 17 (\texttt{cons } \texttt{"cat" } \#\texttt{a}))) \\
& \quad (\texttt{cons } \#t (\lambda (x) (* \ 2 x))))
\]

\[
\begin{align*}
17 : (\texttt{car } (\texttt{car } \texttt{tr})) \\
\texttt{#t} \\
\texttt{#f} : (\texttt{car } (\texttt{cdr } \texttt{tr}))
\end{align*}
\]

\[
\begin{align*}
\texttt{#\texttt{a}} \\
(\lambda (x) (* \ 2 x)) : (\texttt{cdr } (\texttt{cdr } (\texttt{car } \texttt{tr}))) \\
\texttt{#\texttt{a}} : (\texttt{cdr } (\texttt{cdr } (\texttt{car } \texttt{tr})))
\end{align*}
\]

\[
\texttt{"cat"} : (\texttt{car } (\texttt{cdr } (\texttt{car } \texttt{tr})))
\]
**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 (car (cdr (cdr (cdr (car \( E \))))))

Any sequence of up to four \( a \)s and \( d \)s between \( c \)...\( r \) is supported.

---

**Evaluation Rules for car and cdr**

**Big-step semantics:**

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

**Small-step semantics:**

\[
(\text{car } (\text{cons } V1 \ V2)) \Rightarrow V1 [\text{car}]
\]

\[
(\text{cdr } (\text{cons } V1 \ V2)) \Rightarrow V2 [\text{cdr}]
\]

---

**Semantics Puzzle  Solutions**

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

\[
(\text{car } (\text{cons } (+ 2 \ 3) (* 4 \ #t)))
\]

**Answer:**

\[
(\text{car } (\text{cons } ((+ 2 \ \ 3)) (* 4 \ #t)))
\]

\[
\Rightarrow (\text{car } (\text{cons } 5 (* 4 \ #t)))
\]

**Stuck at** \(( * 4 \ #t)\)

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

**Side note:** In so-called lazy languages like Haskell, (cons \( E1 \) \( E2 \)) is a value (even if \( E1 \) and \( E2 \) aren’t values) and car and cdr work as follows:

\[
(\text{car } (\text{cons } E1 \ E2)) \Rightarrow E1 [\text{lazy-car}]
\]

\[
(\text{cdr } (\text{cons } E1 \ E2)) \Rightarrow E2 [\text{lazy-cdr}]
\]

\[
((\text{car } (\text{cons } (+ 2 \ 3) (* 4 \ #t)))) \Rightarrow ((+ 2 \ 3)) [\text{lazy-car}]
\]

\[
\Rightarrow 5 [\text{addition}]
\]

---

**Printed Representations in Racket Interpreter**

\[
> (\text{lambda } (x) (* x 2))
\]

`#<procedure>`

\[
> (\text{cons } (+ 1 \ 2) (* 3 \ #t))
\]

`'(3 . 12)`

\[
> (\text{cons } (\text{cons } 5 \ 6) (\text{cons } 7 \ 8))
\]

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

\[
> (\text{cons } 1 (\text{cons } 2 (\text{cons } 3 \ #t)))
\]

`'(1 2 3 . 4)`

What’s going on here?
Display Notation, Print 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:
  
  
  
  
  becomes `((5 . 6) 7 . 8)`
  
  becomes `(1 . (2 . (3 . 4)))`
  
  becomes `(1 2 3 . 4)`

Why? Because we'll see this makes lists print prettily.

- The **print notation** for pairs adds a single quote mark before the display notation. (We’ll say more about quotation later.)

Racket interpreter uses print (quoted) notation

```racket
> (cons 1 (cons 2 null))
'(1 2)
> (cons (cons 5 6) (cons 7 8))
'((5 . 6) 7 . 8)
> (cons 1 (cons 2 (cons 3 4)))
'(1 2 3 . 4)
```

Why? Because, as we’ll see later, quoted values evaluate to themselves, and so are an easy way to specify a compound data value. Without the quote, the parentheses would indicate function calls and would generate errors.

Functions Can Take and Return Pairs Solutions

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

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

What are the values of these expressions?

- `(swap-pair (cons 1 2))` ⇒* `'(2 . 1)`
- `(sort-pair (cons 4 7))` ⇒* `'(4 . 7)`
- `(sort-pair (cons 8 5))` ⇒* `'(5 . 8)`
Lists

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

A list is either

• The empty list null, a new value whose display notation is ()

• A nonempty list \((\text{cons } V_{\text{first}} \text{ } V_{\text{rest}})\) 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
\[
(\text{cons } 7 \text{ (cons 2 (cons 4 null)))}
\]


list sugar

Treat list as syntactic sugar:*  
• (list) desugars to null
• (list \(E_1\) ...) desugars to (cons \(E_1\) (list ...))

For example:
\[
\begin{align*}
(\text{list } (+ 1 2) \text{ (* 3 4) (< 5 6)}) \\
\text{desugars to } (\text{cons } (+ 1 2) \text{ (list } (* 3 4) \text{ (< 5 6)))} \\
\text{desugars to } (\text{cons } (+ 1 2) \text{ (cons } (* 3 4) \text{ (list } (< 5 6)))} \\
\text{desugars to } (\text{cons } (+ 1 2) \text{ (cons } (* 3 4) \text{ (cons } (< 5 6) \text{ (list))))} \\
\text{desugars to } (\text{cons } (+ 1 2) \text{ (cons } (* 3 4) \text{ (cons } (< 5 6) \text{ null))))}
\end{align*}
\]

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

Box-and-pointer notation for lists

A list of \(n\) values is drawn like this:

For example:

\[
\begin{array}{c}
\text{V1} \\
\rightarrow \\
\text{V2} \\
\rightarrow \\
\cdots \\
\text{Vn} \\
\rightarrow \\
\end{array}
\]

Display Notation for Lists

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

\[
\begin{align*}
\text{(list 7 2 4)} \\
\text{desugars to } (\text{cons } 7 \text{ (cons 2 (cons 4 null)))}) \\
\text{displays as (before rule) } (7 . (2 . (4 . ()))) \\
\text{displays as (after rule) } (7\ 2\ 4) \\
\text{prints as } ' (7\ 2\ 4)
\end{align*}
\]

In Racket:

\[
\begin{align*}
> \text{(cons } 7 \text{ (cons 2 (cons 4 null)))} \\
\text{'(7 2 4)} \\
> \text{(list } 7\ 2\ 4) \\
\text{'(7 2 4)}
\end{align*}
\]
In small-step derivations, it’s helpful to both desugar and resugar with list:

\[
\begin{align*}
\text{list} & \rightarrow \text{cons} \ \text{((+ 1 2) \ (* 3 4) \ (< 5 6))} \\
& \Rightarrow \text{((cons \ (+ 1 2) \ (* 3 4) \ (< 5 6) \ null)))} \\
& \Rightarrow \text{(cons \ 3 \ ((* 3 4) \ (< 5 6) \ null)))} \\
& \Rightarrow \text{(cons \ 3 \ (cons \ 12 \ ((< 5 6) \ null)))} \\
\text{resugar} & \rightarrow \text{list} \ 3 \ 12 \ #t
\end{align*}
\]

Heck, let’s just informally write this as:

\[
\begin{align*}
\text{list} & \rightarrow \text{(cons \ ((+ 1 2) \ (* 3 4) \ (< 5 6))} \\
& \Rightarrow \text{(list \ 3 \ ((* 3 4) \ (< 5 6))} \\
& \Rightarrow \text{(list \ 3 \ 12 \ ((< 5 6))} \\
& \Rightarrow \text{(list \ 3 \ 12 \ #t)}
\end{align*}
\]

**first, rest, and friends**

- **first** returns the first element of a list:
  \[
  \text{first \ (list \ 7 \ 2 \ 4)} \Rightarrow 7 \quad \text{(first \ is \ almost \ a \ synonym \ for \ car, \ but \ requires \ its \ argument \ to \ be \ a \ list)}
  \]

- **rest** returns the sublist of a list containing every element but the first:
  \[
  \text{rest \ (list \ 7 \ 2 \ 4)} \Rightarrow \text{(list \ 2 \ 4)} \quad \text{(rest \ is \ almost \ a \ synonym \ for \ cdr, \ but \ requires \ its \ argument \ to \ be \ a \ list)}
  \]

- Also have **second**, **third**, ..., **ninth**, **tenth**

- Stylistically, **first**, **rest**, **second**, **third** preferred over **car**, **cdr**, **cadr**, **caddr** because emphasizes that argument is expected to be a list.

**length**

length returns the number of top-level elements in a list:

\[
\begin{align*}
\text{length} \ (\text{list \ 7 \ 2 \ 4)} & \Rightarrow 3 \\
\text{length} \ ((17 \ 19) \ (23) \ () \ (111 \ 230 \ 235 \ 251 \ 301)) & \Rightarrow 4 \\
\text{length} \ () & \Rightarrow 0 \\
\text{length} \ () & \Rightarrow 1 \\
\text{length} \ ((12 \ 3 \ . \ 4)) & \Rightarrow 4 \\
\text{length: \ contract \ violation} \\
\text{expected: \ list?} \ (\text{not/c \ empty?}) \\
given: \ '(1 \ 2 \ 3 \ . \ 4)
\end{align*}
\]
List exercise  Solutions

(define LOL
  (list (list 17 19)
        (list 23 42 57)
        (list 110 (list 111 230 235 251 301) 304 342)))

• What is the printed representation of LOL?
  '((17 19) (23 42 57) (110 (111 230 235 251 301) 304 342))

• Give expressions involving LOL that return the following values:
  o 19: (second (first LOL))
  o 23: (first (second LOL))
  o 57: (third (second LOL))
  o 251: (fourth (second (third LOL)))
  o '(235 251 301): (rest (rest (second (third LOL))))

• What is the value of
  (+ (length LOL) ; =>* 3
     (length (third LOL))) ; =>* 4
  (length (second (third LOL)))) ; =>* 5
  ) ; =>* 12

append
append takes any number of lists and returns a list that
combines all of the top-level elements of its argument lists.

> (append '(17 19) '(23 42 57))
'(17 19 23 42 57)

append and sharing
Given two lists L1 and L2, (append L1 L2) copies the list
structure of L1 but shares the list structure of L2.

For example:

L1
8 ---> 3

(append L1 L2)

This fact important when reasoning about number of cons-cells created by a program.
We’ll see why it’s true in the next lecture, when we see how append is implemented
Given more than two lists, append copies all but the last and only shares the last.

cons vs. list vs. append
cons, list, and append are the three most common ways to build lists.
They are very different! Since you will use them extensively in both Racket
and Standard ML, it’s important to master them now!

In the context of lists, (cons Eelt Elist) creates one new cons-cell and returns a list
whose length is 1 more then the length of its 2nd argument (assumed to be a list here).
(list Eelt1 Eelt2) creates a list of length 2 using two new cons-cells.
(list Eelt1 ... Eeltn) creates a list of length n
(append Elist1 Elist2) only makes sense if Elist1 and Elist2 denote lists.
It returns a list whose length is the sum of the length of the two lists.
append can be applied to any number of lists.
Use (cons Eval Elist) rather than (append (list Eval) Elist)

Although (cons Eval Elist) and (append (list Eval) Elist) return equivalent lists, the former is preferred stylistically over the latter (because the former creates only one cons-cell, but the latter creates two).

For example, use this:

```
> (cons (* 6 7) '(17 23 57))
'(42 17 23 57)
```

Rather than this:

```
> (append (list (* 6 7)) '(17 23 57))
'(42 17 23 57)
```