The Pros of **cons**: Pairs and Lists in Racket

CS251 Programming Languages
Fall 2018, 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\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**): (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

**\lambda** char

In Racket, type Command-\ to get \λ char
Evaluation Rules for \texttt{cons}

\textbf{Big step semantics:}

\[
\begin{array}{c}
E_1 \downarrow V_1 \\
E_2 \downarrow V_2 \\
\text{(cons } E_1 E_2) \downarrow (\text{cons } V_1 V_2) \\
[\text{cons}] \\
\end{array}
\]

\textbf{Small-step semantics:}

\texttt{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:

\[
\begin{align*}
(\text{cons } E_1 E_2) & \Rightarrow^* (\text{cons } V_1 \{E_2\}); \text{ first evaluate } E_1 \text{ to } V_1 \text{ step-by-step} \\
(\text{cons } V_1 V_2) & \Rightarrow^* (\text{cons } V_1 \text{ V2}); \text{ then evaluate } e_2 \text{ to } v_2 \text{ step-by-step}
\end{align*}
\]

\textbf{Pairs and Lists}

\textbf{car and cdr}

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

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

Why these names?

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

\textbf{cons evaluation example}

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

\textbf{Practice with car and cdr}

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

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

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

\[
\begin{array}{c}
17 \\
\#\a \\
\text{"cat"} \\
\text{(\lambda} \ x \ (* \ 2 \ x)) \\
\#t
\end{array}
\]
**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 \( \text{car} \ (\text{car} \ (\text{car } e)) \) 
- \( (\text{cddddr } e) \) means \( \text{cdr} \ (\text{cdr} \ (\text{cdr} \ (\text{cdr } e))) \) 

**Evaluation Rules for car and cdr**

**Big-step semantics:**

\[
E \downarrow (\text{cons } V1 \ V2) \quad \begin{array}{c} \text{[car]} \end{array} \\
(\text{car } E) \downarrow V1 \\
E \downarrow (\text{cons } V1 \ V2) \quad \begin{array}{c} \text{[cdr]} \end{array} \\
(\text{cdr } e) \downarrow V2
\]

**Small-step semantics:**

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

**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**

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

\#<procedure>

\[
> (\text{cons} \ (+ \ 1 \ 2) \ (* \ 3 \ 4))
\]

'\(3 . 12)\)

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

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

\[
> (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ 4)))
\]

'(1 2 3 . 4)

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

- The **display notation** for \((\text{cons } V_1 \ V_2)\) is \((\text{DN}_1 \ . \ \text{DN}_2)\), where \(\text{DN}_1\) and \(\text{DN}_2\) are the display notations for \(V_1\) and \(V_2\).
- 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 **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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{cons } 1 \ \text{cons } 2 \ \text{null})))</td>
<td>'1 2</td>
</tr>
<tr>
<td>((\text{cons } (\text{cons } 5 \ 6) \ (\text{cons } 7 \ 8))))</td>
<td>'((5 . 6) 7 . 8)</td>
</tr>
<tr>
<td>((\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 3 \ 4)))))</td>
<td>'1 2 3 . 4</td>
</tr>
</tbody>
</table>

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

```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 null, whose display notation is ()
- A nonempty list (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

(cons 7 (cons 2 (cons 4 null)))

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

Box-and-pointer notation for lists

A list of n values is drawn like this:

V1 ———> V2 ———> ⋯ ———> Vn ———> ⋅

For example:

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:

> (cons 7 (cons 2 (cons 4 null)))

'(7 2 4)

> (list 7 2 4)

'(7 2 4)
**list and small-step evaluation**

In small-step derivations, it’s 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)))`
  - 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)`

**first, rest, and friends**

- **first** returns the first element of a list:
  - `(first (list 7 2 4)) ⇒ 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:
  - `(rest (list 7 2 4)) ⇒ (list 2 4)` (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.**

**first, rest, and friends examples**

```
> (define L '(10 20 (30 40 50 60)))
> (first L) 10
> (second L) 20
> (third L) '(30 40 50 60)
> (fourth L) fourth: list contains too few elements
  list: '(10 20 (30 40 50 60))
> (fourth (third L)) first: contract violation
  expected: (and/c list? (not/c empty?))
  given: '(1 2 3 . 4)
> (fourth (third L))
  (30 40 50 60)
> (rest (third L)) 60
> (rest (third L))
  '() 60
> (first (rest (third L)))
  fourth: list contains too few elements
  list: '(10 20 (30 40 50 60))
```

**length**

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

```
> (length (list 7 2 4))
  3
> (length '((17 19) (23) () (111 230 235 251 301)))
  4
> (length '())
  0
> (length '(()))
  1
> (length '())
  . . length: contract violation
  expected: list?
  given: '(1 2 3 . 4)
```
List exercise

```
(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?
- Give expressions involving LOL that return the following values:
  - 19
  - 23
  - 57
  - 251
  - '(235 251 301)
- What is the value of
  
  (+ (length LOL) (length (third LOL)) (length (second (third LOL))))

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 '(17) 19) '(23 42 57) '(111) () '(230 235 235 301)
'(17 19 23 42 57 111 230 235 235 301)
> (append '((0 1) 2 (3 4 5)) '((0) (6 7 8) 9))
'((0 1) 2 (3 4 5)) ((0) (6 7 8) 9))
> (append '(0 1) 2 (3 4 5))
  . . append: contract violation
    expected: list?
    given: 2
```

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  L2
 /   \
8   7  \
   /   \
   3   2  \
   /     \
   4     \
```

- 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 exercise**

Suppose you are given:

```lisp
(define L1 '(7 2 4))
(define L2 '(8 3 5))
```

For each of the following three expressions:
1. Draw the box-and-pointer structure for its value
2. Write the quoted notation for its value
3. Determine the length of its value

```lisp
(cons L1 L2)
(list L1 L2)
(append L1 L2)
```

**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:

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

Rather than this:

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