The Pros of cons:
Pairs and Lists in Racket

CS251 Programming Languages
Spring 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
  not to 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

(cons V1 V2)

(cons (cons 17 (cons "cat" #\a))
     (cons #t (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:

\textbf{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:

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

cons evaluation example

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

Pairs and Lists

Pairs and Lists

Practice with \textbf{car} and \textbf{cdr}

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

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

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

Why these names?

\begin{itemize}
\item \textbf{car} from “contents of address register”
\item \textbf{cdr} from “contents of decrement register”
\end{itemize}
**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 (cdr e)))`
- `(caaar e)` means `(car (car (car e)))`
- `(cddddr e)` means `(cdr (cdr (cdr (cdr (cdr e)))))`

**Evaluation Rules for car and cdr**

**Big-step semantics:**

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

**Small-step semantics:**

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

**Semantics Puzzle**

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

`(car (cons (+ 2 3) (* 5 #t)))`

**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, 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:
  
  ```
  ((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

```racket
> (display (cons 1 (cons 2 null)))
(1 2)
> (display (cons (cons 5 6) (cons 7 8)))
(((5 . 6) 7 . 8)
> (display (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.

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

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?

- `(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 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)))

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

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:

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

\( \text{desugars to} \ (\text{cons} \ ((+ 1 2)) \ (\text{cons} \ (* 3 4) \ \text{null})) \)

\( \Rightarrow (\text{cons} \ 3 \ (\text{cons} \ ((* 3 4)) \ \text{null})) \)

\( \Rightarrow (\text{cons} \ 3 \ (\text{cons} \ 12 \ \text{null})) \)

\( \Rightarrow (\text{cons} \ 3 \ (\text{cons} \ #t \ \text{null})) \)

\( \text{resugars to} \ (\text{list} \ 3 \ 12 \ #t) \)

Heck, let's just informally write this as:

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

\( \Rightarrow (\text{list} \ 3 \ (\{(* 3 4)\} \ (< 5 6))) \)

\( \Rightarrow (\text{list} \ 3 \ 12 \ (\{< 5 6\})) \)

\( \Rightarrow (\text{list} \ 3 \ 12 \ #t) \)

first, rest, and friends

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

- \( \text{rest} \) 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 \( \text{cdr} \), 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:

\( > (\text{length} \ (\text{list} \ 7 \ 2 \ 4)) \)

3

\( > (\text{length} \ '((17 \ 19) \ (23) \ () \ (111 \ 230 \ 235 \ 251 \ 301))) \)

4

\( > (\text{length} \ '()) \)

0

\( > (\text{length} \ '(())) \)

1

\( > (\text{length} \ '(1 \ 2 \ 3 \ . \ 4)) \)

. . 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:
  o 19
  o 23
  o 57
  o 251
  o '(235 251 301)
• What is the value of
  (+ (length LOL)
     (length (third LOL))
     (length (second (third LOL))))?

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 '((0 1) 2 (3 4 5)) '() '(6 (7 8) 9))
'((0 1) 2 (3 4 5))

cons vs. list vs. append

ccons, 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 Eval EList) creates one new cons-cell and returns a list whose length is 1 more than the length of its 2nd argument (assumed to be a list here).

(list Eval1 Eval2) creates a list of length 2 using two new cons-cells.
(list Eval1 ... Evaln) 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.

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

(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 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

- `(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)
```