The Pros of \texttt{cons}:
Pairs and Lists in Racket

CS251 Programming Languages
Spring 2019, Lyn Turbak
Department of Computer Science
Wellesley College

\texttt{cons} Glues Two Values into a Pair

A new kind of value:
• pairs (a.k.a. \texttt{cons} cells): \texttt{(cons V1 V2)}
  e.g.,
  - \texttt{(cons 17 42)}
  - \texttt{(cons 3.14159 \#t)}
  - \texttt{(cons "CS251" (\lambda (x) (* 2 x))} \)
  - \texttt{(cons (cons 3 4.5) (cons \#f \#\a))}
• Can glue any number of values into a \texttt{cons} tree!

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 \texttt{exact}, the rest are \texttt{inexact}. See docs.
• strings: "cat", "CS251", "αβγ",
  "To be\ not\ to be"
• characters: \#\a, \#\A, \#\5, \#\space, \#\tab, \#\newline
• anonymous functions: \texttt{(lambda (a b) (+ a (* b c)))}

What about compound data?

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

\textbf{Example:}
\texttt{(cons V1 V2)} \texttt{V1 V2}

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

\texttt{(cons (cons 17 (cons "cat" \#\a)) (cons \#t (\lambda (x) (* 2 x)))))}
Evaluation Rules for cons

Big step semantics:

\[
\begin{align*}
E1 & \downarrow V1 \\
E2 & \downarrow V2 \\
(\text{cons } E1 \ E2) & \downarrow (\text{cons } V1 \ V2) \\
\end{align*}
\]

Small-step semantics:

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

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

cons evaluation example

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

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

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

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

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

\[
\begin{array}{c}
17 \\
#\text{\a}
\end{array} \text{ 17} \text{ #t} \\
\begin{array}{c}
\text{"cat"} \\
(\lambda \ (x) (* \ 2 \ x))
\end{array}
\]

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

Any sequence of up to four as and ds between c...r is supported.

Evaluation Rules for car and cdr

Big-step semantics:

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

Small-step semantics:

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

Semantics Puzzle

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

(car (cons (+ 2 3) (* 4 #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, Print 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 **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

- \(> (\text{print } (\text{cons 1 (cons 2 null))})\)
  \[
  '(1 2)
  \]
- \(> (\text{print } (\text{cons (cons 5 6) (cons 7 8)}))\)
  \[
  '((5 . 6) 7 . 8)
  \]
- \(> (\text{print } (\text{cons 1 (cons 2 (cons 3 4))}))\)
  \[
  '(1 2 3 . 4)
  \]

**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

\[
(\text{define (swap-pair pair})
  \]
\[
(\text{cons (cdr pair) (car pair))})
  \]
\[
(\text{define (sort-pair pair})
  \]
\[
(\text{if (< (car pair) (cdr pair))})
  \]
\[
(\text{pair})
  \]
\[
(\text{(swap-pair pair))})
  \]

**What are the values of these expressions?**
- \((\text{swap-pair (cons 1 2))}\)
- \((\text{sort-pair (cons 4 7))}\)
- \((\text{sort-pair (cons 8 5))}\)
Lists

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

A list is either

- The empty list null, a new value 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

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

**list sugar**

Treat **list** as syntactic sugar:

- (list) desugars to null
- (list *E1* ...) desugars to (cons *E1* (list ...))

For example:

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

* 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:

\[
\begin{align*}
V1 & \rightarrow V2 \rightarrow \cdots \rightarrow Vn \\
\end{align*}
\]

**Display Notation for Lists**

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

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

In Racket:

\[
\begin{align*}
> \text{(cons 7 (cons 2 (cons 4 null)))} & \quad \text{’(7 2 4)} \\
> \text{(list 7 2 4)} & \quad \text{’(7 2 4)} \\
\end{align*}
\]
list and small-step evaluation

In small-step derivations, it's helpful to both desugar and resugar with list:

\[
\text{(list (+ 1 2) (* 3 4) (< 5 6))}
\]
\[
\text{desugars to (cons \{(+ 1 2)\} (cons (* 3 4) (cons (< 5 6) null)))}
\]
\[
\Rightarrow (\text{cons 3 (cons \{(* 3 4)\} (cons (< 5 6) null)))}
\]
\[
\Rightarrow (\text{cons 3 (cons 12 (cons \{(< 5 6)\} null)))}
\]
\[
\Rightarrow (\text{cons 3 (cons 12 (cons #t null))})
\]
\[
\text{resugars to (list 3 12 #t)}
\]

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

• first returns the first element of a list:
  
  \[
  \text{(first (list 7 2 4)) \Rightarrow 7}
  \]
  
  (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:
  
  \[
  \text{(rest (list 7 2 4)) \Rightarrow (list 2 4)}
  \]
  
  (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.

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

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

\[
\text{expected: list? (not/c empty?)}
\]
given: '(1 2 3 . 4)

\[
> (\text{length '(1 2 3 . 4)})
\]
length: contract violation

\[
\text{expected: list?}
\]
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) '())
'(17 19 23 42 57 111)

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

> (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

8 ——> 3

(append L1 L2)

8 ——> 3

L2

7 ——> 2 ——> 4

- This fact is 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:

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

For each of the following three lists:
1. Draw the box-and-pointer structure for its value
2. Indicate the number of conses created for its value
3. Write the quoted notation for its value
4. Determine the length of its value

```
(define L3 (cons L1 L2))
(define L4 (list L1 L2))
(define L5 (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:

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

Rather than this:

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