

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



**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):
`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?

Pairs and Lists 2

`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!

In Racket,
type `Command-\`
to get `λ char`

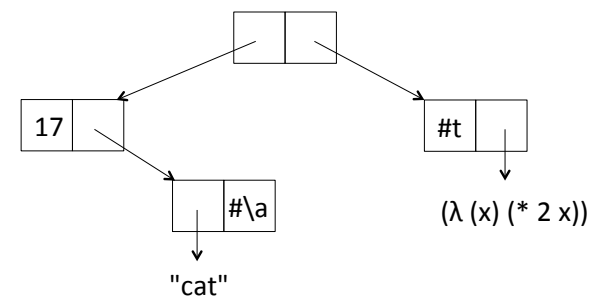
Box-and-pointer diagrams for `cons` trees

`(cons V1 V2)`

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.

```
(cons (cons 17 (cons "cat" #\a))
      (cons #t (λ (x) (* 2 x))))
```



Pairs and Lists 3

Pairs and Lists 4

Evaluation Rules for cons

Big step semantics:

$$\frac{\begin{array}{l} E1 \downarrow V1 \\ E2 \downarrow V2 \end{array}}{(\text{cons } E1 \ E2) \downarrow (\text{cons } V1 \ V2)} \text{ [cons]}$$

Small-step semantics:

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

(cons E1 E2)

\Rightarrow^* **(cons V1 E2)**; first evaluate **E1** to **V1** step-by-step

\Rightarrow^* **(cons V1 V2)**; then evaluate **E2** to **V2** step-by-step

Pairs and Lists 5

cons evaluation example

```
(cons (cons {(+ 1 2)} (< 3 4))
      (cons (> 5 6) (* 7 8)))
⇒ (cons (cons 3 {(< 3 4)})
        (cons (> 5 6) (* 7 8)))
⇒ (cons (cons 3 #t) (cons {(> 5 6)} (* 7 8)))
⇒ (cons (cons 3 #t) (cons #f {(* 7 8)}))
⇒ (cons (cons 3 #t) (cons #f 56))
```

Pairs and Lists 6

car and cdr

- **car** extracts the left value of a pair

`(car (cons 7 4)) ⇒ 7`

- **cdr** extract the right value of a pair

`(cdr (cons 7 4)) ⇒ 4`

Why these names?

- **car** from “contents of address register”
- **cdr** from “contents of decrement register”

Pairs and Lists 7



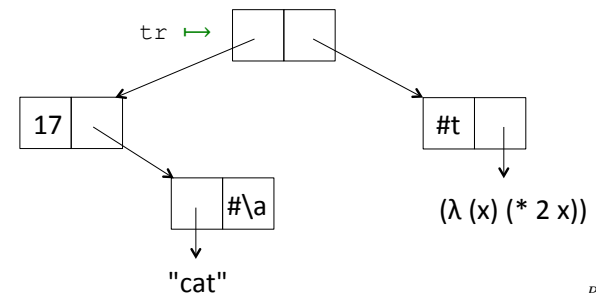
Practice with car and cdr



Write expressions using **car**, **cdr**, and **tr** that extract the five leaves of this tree:

```
(define tr (cons (cons 17 (cons "cat" #\a))
                 (cons #t (lambda (x) (* 2 x)))))
```

```
tr → (cons (cons 17 (cons "cat" #\a))
           (cons #t (lambda (x) (* 2 x)))) , ...
```



Pairs and Lists 8

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)))`
⋮
- `(cddddr E)` means `(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:

$$\frac{E \downarrow (\text{cons } V1 \ V2)}{(\text{car } E) \downarrow V1} \text{ [car]} \qquad \frac{E \downarrow (\text{cons } V1 \ V2)}{(\text{cdr } E) \downarrow V2} \text{ [cdr]}$$

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

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 `(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.)

Pairs and Lists 13

display vs. print in 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)
```

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

Pairs and Lists 14

Racket interpreter uses print (quoted) notation

```
> (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.

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

```
> (1 2)  
application: not a procedure;  
expected a procedure that can be  
applied to arguments  
given: 1  
arguments....:
```

Pairs and Lists 15

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 pair)))
```

What are the values of these expressions?

- `(swap-pair (cons 1 2))`
- `(sort-pair (cons 4 7))`
- `(sort-pair (cons 8 5))`

Pairs and Lists 16

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***

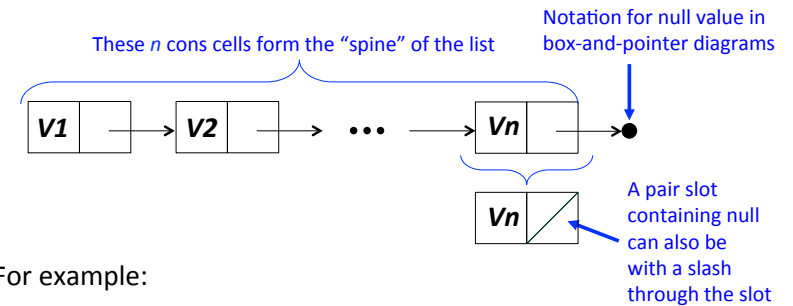
E.g., a list of the 3 numbers 7, 2, 4 is written

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

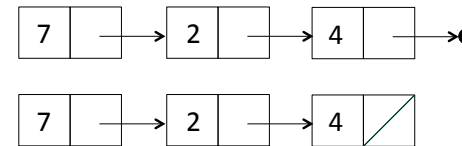
Pairs and Lists 17

Box-and-pointer notation for lists

A list of n values is drawn like this:



For example:



Pairs and Lists 18

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

Pairs and Lists 19

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)
```

Pairs and Lists 20

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)))
⇒ (cons 3 (cons 12 (cons #t 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)
```

Pairs and Lists 21

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.

Pairs and Lists 22

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 (third L))
60
> (rest (third L))
'(40 50 60)
```

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

Pairs and Lists 23

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 '(1 2 3 . 4))
length: contract violation
expected: list?
given: '(1 2 3 . 4)
```

Pairs and Lists 24

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) '() '(230 235 251 301))
'(17 19 23 42 57 111 230 235 251 301)

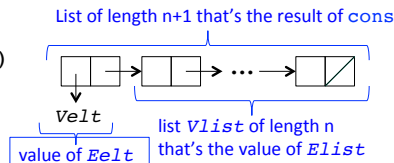
> (append '((0 1) 2 (3 (4 5))) '() (6 (7 8) 9))
'((0 1) 2 (3 (4 5)) () (6 (7 8) 9))

> (append '(0 1) 2 '(3 (4 5)))
append: contract violation
  expected: list?
  given: 2
```

cons vs. list vs. append

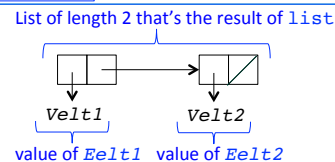
cons, list, and append are the three most common ways to build lists. They are very different! Since you will use them extensively in both Racket and Standard ML, it's important to master them **now!**

In the context of lists, (cons Eelt Elist) creates one new cons-cell and returns a list whose length is 1 more than the length of its 2nd argument (assumed to be a list here).



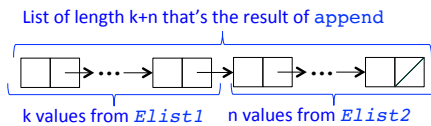
(list Eelt1 Eelt2) creates a list of length 2 using two new cons-cells.

(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.

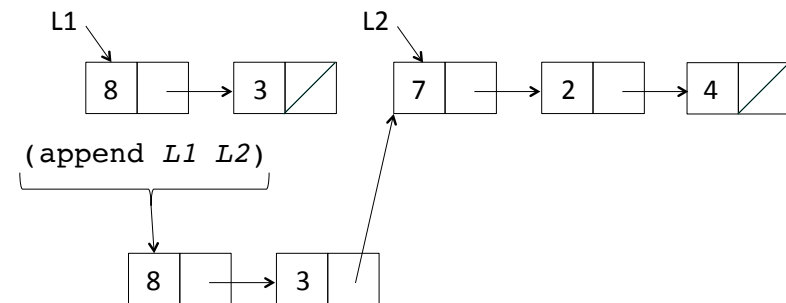
append can be applied to any number of lists.



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:



- 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:

```
(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)
```