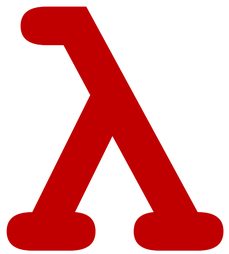
CSCI 275: Programming Abstractions

Lecture 04: Testing, Style & Modules

Spring 2025

Stephen Checkoway, **Oberlin College**
Slides gratefully borrowed from Molly Q Feldman



Goals for Today's Class

We'll have the basic tools to be able to write Racket programs.

Today:

How can we write them *well* and test them *effectively?*

List/Pair Wrap Up

Lists

Lists are the ***most important*** data type in Racket

A list is one of two things:

- The empty list (`empty`, `null` and `'()` produce the empty list)

- A pair `(x . y)` where `x` is an expression and `y` is a list

We can create a list with `(list 3 (+ 8 5) #f)` which gives `'(3 13 #f)`

Two (Deeper) Questions

1. While we can construct lists with list, they print out with a quotation mark. Why?

2. We said that lists were pairs `(x . y)` where `x` is an expression and `y` is a list. What is a pair?

Quoting in Racket

Placing a `'` before an s-expression "quotes" it

- The quoted expression is treated as ***data***, *not code*
- DrRacket displays lists with the quote

Quoting in Racket

Placing a `'` before an s-expression "quotes" it

- The quoted expression is treated as ***data***, *not code*
- DrRacket displays lists with the quote

`'(1 4 5)` is a 3-element list

We saw `(list (* 2 3) (and #t #f) 8)` produces

```
'(6 #f 8)
```

`'((* 2 3) (and #t #f) 8)` produces

```
'((* 2 3) (and #t #f) 8)
```

Quoting, in general, is how we represent data

Quoting a number, boolean, or string returns that number, boolean, or string

- `'35` gives `35`
- `'#t` gives `#t`
- `'"Hello!"` gives `"Hello!"`

Quoting a variable gives a symbol

- `+` and `string-append` are variables whose values are procedures
- `'+` and `'string-append` are symbols

Quoting a list gives a list of quoted elements

- `'(1 2 x y)` is the same as `(list '1 '2 'x 'y)`
- `'(() (1) (1 2 3))` is the same as `(list '() '(1) '(1 2 3))`

Guidelines for creating lists

If you want to evaluate some expressions and have the resulting values be in the list, use `(list expr1 expr2 ... exprn)`

Example: `(list x (list x y z) z)`

If you want to create a list of literal numbers/strings/booleans/symbols, use `'(...)`

Example: `'(10 15 20 -3)`

Given variables `x` and `y`, how do we create a list containing the values of `x`, `y`, and `x + y`?

i.e., if `x` is 10 and `y` is 15, the list we want is `'(10 15 25)`.

A. `(list x y (+ x y))`

B. `(list 'x 'y (+ 'x 'y))`

C. `(list 'x 'y '(+ x y))`

D. `'(x y (+ x y))`

E. All of the above

Two (Deeper) Questions

1. While we can construct lists with list, they print out with a quotation mark. Why?

2. We said that lists were pairs `(x . y)` where `x` is an expression and `y` is a list. What is a pair?

Pair are the (traditional) data structure in Scheme

Pairs hold data. To create a pair you use the `cons` procedure, which takes two arguments: `(cons a b)`

Top Tip: If you evaluate a term and it prints with a `.` in the middle (i.e. `'(2 . 3)`) that is a *pair* not a *list*

`cons` means “create a pair”

- `(cons 'x 'y)` creates the pair `'(x . y)`
- `(cons 2 3)` creates the pair `'(2 . 3)`
- `(cons 5 null)` creates the list `'(5)`

Lists are simply (useful) special cases of pairs –
All operators for pairs also work with lists, but not vice versa

`cons` helps us build up lists, one-by-one

If we have a list `lst` and an element `x`, prepend `x` to `lst`: `(cons x lst)`

```
(cons "c" (list "a" "b")) => '("c" "a" "b")
```

This works because the second argument to `cons` is a list so the result is a list

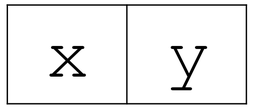
What if we want to append `x` to `lst`? Can we use `(cons lst x)`?

Will `(cons '(1 2 3) 4)` produce `'(1 2 3 4)`?

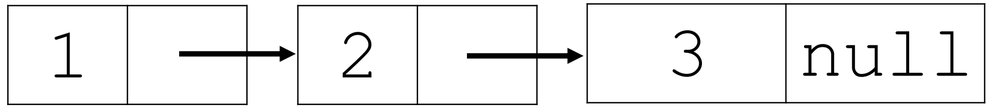
A. Yes
B. No

Cons cells

`(cons x y)` creates a *cons-cell*



`(cons 1 (cons 2 (cons 3 null)))` produces



You'll notice that this is a linked list!

This is the same list that's produced by `(list 1 2 3)`

Lists are either `null` or pairs whose second element is a list. We can create a pair using `(cons x y)`. How can we use `cons` to create the 3-element list `'(#t #f 89)`?

A. `(cons #t (cons #f (cons 89 null)))`

B. `(cons (cons (cons (#t #f) 89 null)`

C. `(cons #t (cons #f 89))`

D. `(cons (cons #t #f) 89)`

E. More than one of the above (which?)

Get the first element from a pair

`car` (Contents of the Address part of a Register*)

Returns the first element of a pair (or the head of a list)

`(car (cons 5 8))` (equivalently `(car '(5 . 8))`) returns `5`

`(car '(1 2 3 4))` returns `1`

`(car (1 2 3 4))` is an error because `(1 2 3 4)` is invalid

* This terminology comes from the IBM 704, an ancient computer

Get the second element of the pair

`cdr` (Contents of the Decrement part of a Register*)

Returns the second element of a pair (or the tail of a list); pronounced "could-er"

`(cdr (cons 5 8))` (equivalently `(cdr '(5 . 8))`) returns 8

`(cdr '(1 2 3 4))` returns the list `'(2 3 4)`

`(cdr '(5))` returns the empty list, DrRacket will display `'()`

Note: `cdr` is equivalent to `rest`, **not second** in Racket terminology

* This terminology comes from the IBM 704, an ancient computer

Recap

To create a list with a fixed number of elements: `(list x1 x2 … xn)`

`x1` … `xn` are arbitrary s-expressions that will be evaluated and their values put in a list

To create a list with a fixed number of literal values: `'(a b 5 3 (2 3) #f)`

To add an element `x` to the beginning of an existing list `lst`: `(cons x lst)`

This returns a new list! It doesn't modify anything

To get the first element of the list: `(first lst)`

To get the rest of the list (i.e., not the first element): `(rest lst)`

```
(define fun
  (lambda (lst1 lst2)
    (cond [(empty? lst1) lst2]
          [else (cons (first lst1)
                      (fun (rest lst1) lst2))])))
```

What is the result of `(fun '(1 2 3 4) '(a b c))`?

A. `'(1 2 3 4 a b c)`
B. `'(4 3 2 1 a b c)`
C. `'(1 2 3 4 c b a)`
D. `'(4 3 2 1 c b a)`
E. `'(a b c)`

About Testing

In this class, you'll be required to write test suites for each piece of Racket code you write.

Why do you think that's an expectation of this class?

Some Reasons to Test

- I can write tests *once*, and test my implementation as I go along *as many times as I want*
  - This helps me be a more efficient programmer

- Writing tests helps me *design my solutions*
  - I can identify edge cases *before I start* programming
  - I can immediately decide if my first attempt makes sense

- At-scale software in the real world is very complex
  - Testing is an important point of maintaining software quality in the real world

Relevant Quotes from Software Engineering at Google

"The ability for humans to manually validate every behavior in a system has been **unable to keep pace** with the explosion of features and platforms in most software."

"In addition to empowering companies to build great products quickly, testing is becoming critical **to ensuring the safety of important products** and services in our lives."

"Keep in mind that **tests derive their value from the trust engineers place in them**."

Related: Test Driven Development

- A software engineer philosophy
  - *Start* with tests
  - *Develop* according to their requirements

<u>Kent Beck from “Test Driven Development: By Example”:</u>
The two rules imply an order to the tasks of programming.

Red— Write a little test that doesn't work, and perhaps doesn't even compile at first.

Green— Make the test work quickly, committing whatever sins necessary in the process.

Refactor— Eliminate all of the duplication created in merely getting the test to work.

Let’s think about some good tests

`(remove-numbers lst)` — Remove all of the numbers from `lst`

In your small groups, think about some example lists that would help you (a) design this procedure and (b) test out a correct implementation?

Tests are worth points too!

- Especially for the later problems in HW0/HW1, it can be really useful to *write tests first* that explore what the problem is trying to do
- You get points for tests even if your implementation does not pass my tests

Aside: Print Debugging

This is *less common* and *less easily supported* in Racket than in other languages – has to do with evaluation methods in Racket

It can be useful and there are numerous Racket procedures:

`print, write, display`

I recommend using `println` and `displayln`

```
> (println "there")
"there"
> (displayln "there")
there
> (println 4)
4
> (displayln 5)
5
```