Section 6
What does mutation mean?
When do function bodies run?

Winter 2018

With thanks to: Dan Grossman / Eric Mullen
Set!

- Unlike ML, Racket really has assignment statements
  - But used only-when-really-appropriate!

\[ (\text{set! } x \ e) \]

- For the \( x \) in the current environment, subsequent lookups of \( x \) get the result of evaluating expression \( e \)
  - Any code using this \( x \) will be affected
  - Like \( x = e \) in Java, C, Python, etc.

- Once you have side-effects, sequences are useful:

\[ (\text{begin } e_1\ e_2\ \ldots\ e_n) \]
Example

Example uses `set!` at top-level; mutating local variables is similar

```
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4)) ; 7
(set! b 5)
(define z (f 4)) ; 9
(define w c) ; 7
```

Not much new here:

- Environment for closure determined when function is defined, but body is evaluated when function is called
- Once an expression produces a value, it is irrelevant how the value was produced
The truth about cons

cons just makes a pair
  – Often called a cons cell
  – By convention and standard library, lists are nested pairs that eventually end with null

(define pr (cons 1 (cons #t "hi"))) ; '(1 #t . "hi")
(define lst (cons 1 (cons #t (cons "hi" null))))
(define hi (cdr (cdr pr)))
(define hi-again (car (cdr (cdr lst))))
(define hi-another (caddr lst))
(define no (list? pr))
(define yes (pair? pr))
(define of-course (and (list? lst) (pair? lst)))

Passing an improper list to functions like length is a run-time error
The truth about cons

So why allow improper lists?
- Pairs are useful
- Without static types, why distinguish $(e_1, e_2)$ and $e_1::e_2$

Style:
- Use proper lists for collections of unknown size
- But feel free to use cons to build a pair
  - Though structs (like records) may be better

Built-in primitives:
- list? returns true for proper lists, including the empty list
- pair? returns true for things made by cons
  - All improper and proper lists except the empty list
**cons cells are immutable**

What if you wanted to mutate the *contents* of a cons cell?

- In Racket you cannot (major change from Scheme)
- This is good
  - List-aliasing irrelevant
  - Implementation can make `list?` fast since listness is determined when cons cell is created
Set! does not change list contents

This does not mutate the contents of a cons cell:

```
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
(define fourteen (car y))
```

- Like Java’s \texttt{x = new Cons(42,null)}, \texttt{not x.car = 42}
**mcons cells are mutable**

Since mutable pairs are sometimes useful (will use them soon), Racket provides them too:

- `mcons`
- `mcar`
- `mcdr`
- `mpair?`
- `set-mcar!`
- `set-mcdr!`

Run-time error to use `mcar` on a cons cell or `car` on an mcons cell
Delayed evaluation

For each language construct, the semantics specifies when subexpressions get evaluated. In ML, Racket, Java, C:

- Function arguments are *eager* (call-by-value)
  - Evaluated once before calling the function
- Conditional branches are not eager

It matters: calling `factorial-bad` never terminates:

```
(define (my-if-bad x y z) (if x y z))

(define (factorial-bad n) (my-if-bad (= n 0) 1 (* n (factorial-bad (- n 1)))))
```
We know how to delay evaluation: put expression in a function!

– Thanks to closures, can use all the same variables later

A zero-argument function used to delay evaluation is called a thunk

– As a verb: thunk the expression

This works (but it is silly to wrap if like this):

```
(define (my-if x y z)
  (if x (y) (z)))

(define (fact n)
  (my-if (= n 0)
    (lambda() 1)
    (lambda() (* n (fact (- n 1))))))
```
The key point

• Evaluate an expression e to get a result:

\[ e \]

• A function that \textit{when called}, evaluates e and returns result
  – Zero-argument function for “thunking”

\[ \text{(lambda () e)} \]

• Evaluate e to some thunk and then call the thunk

\[ (e) \]

• Next: Powerful idioms related to delaying evaluation and/or
  avoided repeated or unnecessary computations
  – Some idioms also use mutation in encapsulated ways
**Avoiding expensive computations**

Thunks let you skip expensive computations if they are not needed

Great if take the true-branch:

```scheme
(define (f th)
  (if (...) 0 (... (th) ...)))
```

But worse if you end up using the thunk more than once:

```scheme
(define (f th)
  (... (if (...) 0 (... (th) ...))
       (if (...) 0 (... (th) ...))
       ...
       (if (...) 0 (... (th) ...))))
```

In general, might not know many times a result is needed
Best of both worlds

Assuming some expensive computation has no side effects, ideally we would:

– Not compute it *until needed*
– *Remember the answer* so future uses complete immediately

Called *lazy evaluation*

Languages where most constructs, including function arguments, work this way are *lazy languages*

– Haskell

Racket predefines support for *promises*, but we can make our own

– Thunks and mutable pairs are enough… [Friday]