CSE341: Programming Languages

Section 6

What does mutation mean?
When do function bodies run?

Winter 2018

With thanks to: Dan Grossman / Eric Mullen

Agenda

- Let Expressions
- Mutation: Set!
- Delayed Evaluations: Thunks

---

**Let**

A let expression can bind any number of local variables
- Notice where all the parentheses are

The expressions are all evaluated in the environment from *before the let-expression*
- Except the body can use all the local variables of course
- This is **not** how ML let-expressions work
- Convenient for things like `(let ([x y] [y x]) ` ...

```
(define (silly-double x)
  (let ([x (+ x 3)]
        [y (+ x 2)])
    (+ x y -5)))
```

---

**Let***

*Syntactically, a* let* expression is a let-expression with 1 more character*

The expressions are evaluated in the environment produced from the *previous bindings*
- Can repeat bindings (later ones shadow)
- This is how ML let-expressions work

```
(define (silly-double x)
  (let* ([x (+ x 3)]
         [y (+ x 2)])
    (+ x y -8)))
```
**Letrec**

*Syntactically, a letrec expression is also the same*

The expressions are evaluated in the environment that includes all the bindings

\[
\begin{align*}
\text{(define (silly-triple x)} & \text{)} \\
\text{(letrec ([y (+ x 2)] (f (lambda(z) (+ z y w x))) [w (+ x 7)]) (f -9))}
\end{align*}
\]

- Needed for mutual recursion
- But expressions are still evaluated in order: accessing an uninitialized binding raises an error
  - Remember function bodies not evaluated until called

**More letrec**

- Letrec is ideal for recursion (including mutual recursion)

\[
\begin{align*}
\text{(define (silly-mod2 x)} & \text{)} \\
\text{(letrec ([even? (+=x (if (zero? x) \#t (odd? (- x 1))))] [odd? (+=x (if (zero? x) \#f (even? (- x 1))))]) (if (even? x) 0 1))}
\end{align*}
\]

- Do not use later bindings except inside functions
  - This example will raise an error when called

\[
\begin{align*}
\text{(define (bad-letrec x)} & \text{)} \\
\text{(letrec ([y z] [z 13]) (if x y z))}
\end{align*}
\]

**Local defines**

- In certain positions, like the beginning of function bodies, you can put defines
  - For defining local variables, same semantics as letrec

\[
\begin{align*}
\text{(define (silly-mod2 x)} & \text{)} \\
\text{(define (even? x) (if (zero? x) \#t (odd? (- x 1))))} \\
\text{(define (odd? x) (if (zero? x) \#f (even? (- x 1))))} \\
\text{(if (even? x) 0 1))}
\end{align*}
\]

- Local defines is preferred Racket style, but course materials will avoid them to emphasize let, let*, letrec distinction
  - You can choose to use them on homework or not

**Top-level**

The bindings in a file work like local defines, i.e., letrec

- Like ML, you can refer to earlier bindings
- Unlike ML, you can also refer to later bindings
- But refer to later bindings only in function bodies
  - Because bindings are evaluated in order
  - Get an error if try to use a not-yet-defined binding
- Unlike ML, cannot define the same variable twice in module
  - Would make no sense: cannot have both in environment
REPL

Unfortunate detail:
- REPL works slightly differently
  - Not quite let* or letrec
  - 😐
- Best to avoid recursive function definitions or forward references in REPL
  - Actually okay unless shadowing something (you may not know about) – then weirdness ensues
  - And calling recursive functions is fine of course

Optional: Actually…

- Racket has a module system
  - Each file is implicitly a module
  - Not really “top-level”
  - A module can shadow bindings from other modules it uses
    - Including Racket standard library
  - So we could redefine + or any other function
    - But poor style
    - Only shadows in our module (else messes up rest of standard library)

  - (Optional note: Scheme is different)

Set!

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

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

- 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:
  ///

  \[(\text{begin } e1\ e2\ \ldots\\text{ en})\]

Example

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

\[
(\text{define } b\ 3)\\
(\text{define } f\ (\text{lambda } (x)\ (*\ 1\ (+\ x\ b))))\\
(\text{define } c\ (+\ b\ 4))\quad;\ 7\\
(\text{set! } b\ 5)\\
(\text{define } z\ (f\ 4))\quad;\ 9\\
(\text{define } w\ c)\quad;\ 7
\]

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
Top-level

• Mutating top-level definitions is particularly problematic
  – What if any code could do set! on anything?
  – How could we defend against this?

• A general principle: If something you need not to change might change, make a local copy of it. Example:

```
(define b 3)
(define f (let ([b b]) (lambda (x) (* 1 (+ x b))))))
```

Could use a different name for local copy but do not need to

But wait…

• Simple elegant language design:
  – Primitives like + and * are just predefined variables bound to functions
  – But maybe that means they are mutable
  – Example continued:

```
(define f (let ([b b] [+ +] [* *]) (lambda (x) (* 1 (+ x b))))))
```

– Even that won’t work if f uses other functions that use things that might get mutated – all functions would need to copy everything mutable they used

No such madness

In Racket, you do not have to program like this
  – Each file is a module
  – If a module does not use set! on a top-level variable, then Racket makes it constant and forbids set! outside the module
  – Primitives like +, *, and cons are in a module that does not mutate them

Showed you this for the concept of copying to defend against mutation
  – Easier defense: Do not allow mutation
  – Mutable top-level bindings a highly dubious idea

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:
– \texttt{list?} returns true for proper lists, including the empty list
– \texttt{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 \texttt{list?} fast since listness is determined when cons cell is created

Set! does not change list contents

This does \textit{not} mutate the contents of a cons cell:

\begin{verbatim}
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
(define fourteen (car y))
\end{verbatim}

– 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:
– \texttt{mcons}
– \texttt{mcar}
– \texttt{mcdr}
– \texttt{mpair?}
– \texttt{set-mcar!}
– \texttt{set-mcdr!}

Run-time error to use \texttt{mcar} on a cons cell or \texttt{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:

```scheme
(define (factorial-bad n)
  (my-if-bad (= n 0)
           1
           (* n (factorial-bad (- n 1))))
)
```

Thunks delay

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):

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

The key point

- Evaluate an expression `e` to get a result:
  ```scheme
  e
  ```

- A function that *when called*, evaluates `e` and returns result
  - Zero-argument function for "thunking"
    ```scheme
    (lambda () e)
    ```

- Evaluate `e` to some thunk and then call the thunk
  ```scheme
  (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]