More on Racket: Bindings and Mutation

We discuss in this section the various options that Racket provides for creating bindings. We also discuss the tools it offers for mutation.

Bindings

In addition to define discussed earlier, there are four other let-style bindings in use in Racket, all with slightly different semantics, all with their uses.

The first of these is a let construct very similar to OCAML's:

```
(let ([x 3]
      [y (+ 2 3)])
  (+ x y))
```

In other words the first argument in the let expression is a parenthesized sequence of pairs, represented by the square brackets, each pair being a binding of a symbol to an expression. The second argument to the let is an expression that is evaluated in a context where all the bindings described by the first argument are in place. Notice also the use of square brackets; square brackets are actually equivalent to parentheses in Racket, but we tend to use them in some specific cases to identify the terms involved; in this case the various bindings.

It is important here to keep in mind that the bindings are evaluated all at once, and so a later binding cannot depend on a previous one. The following for example would signal an error:

```
(let ([x 3]
      [y (+ 2 x)])
  (+ x y))
```

The second construct, which "solves" that "problem", is called let*. It looks exactly like let, except that the bindings occur in order, and each future binding can use the bindings that came before it. You can think of a let* as a sequence of nested lets.

```
(let* ([x 3]
       [y (+ 2 x)])
  (+ x y))
```

A third construct is letrec. In letrec all the bindings are being evaluated in an environment where the bindings are already defined. This sounds very self-referential, and it is, but it makes sense if you want to create mutually recursive functions. For example, here is a rather inefficient way to determine if a positive number is even:

```
(define (isEven n)
  (letrec
      ([even? (lambda (x) (if (= 0 x)
                              #t
                              (odd? (- x 1))))]
       [odd? (lambda (x) (if (= 0 x)
                             #f
                             (even? (- x 1))))])
    (even? n)))
```

Here the functions even? and odd? depend on each other for their work. Situations like this make letrec a valuable feature.

It is really important to keep in mind that letrec allows expressions to refer to bindings that might only exist later in time. If those bindings are accessed too soon then problems arise. So the following would in fact give us an error, because the [y z] binding tries to set the value of y to equal z before z had a chance to be defined to equal 3; that hasn't happened yet.

```
(letrec ([y z]
         [z 3])
  y)
```

Variable Mutation

Racket actually supports what we named *variable mutation*. There is an operator that allows you to change the value of bindings:

```
(define b 3)
(define (f) b)
b
(f)
(set! b 5)
b
(f)
```

Notice the set! line, that changes the value of b. Notice also how that changes the behavior of the function f. It's not that we simply changed what b means in the future. We actually changed what b means in the closure of the function f that was defined prior to the set! line.

As another example, here's a simple function that keeps increasing an internal counter:

```
(define next
  (let ([counter 0])
    (lambda () (begin (set! counter (+ counter 1))
                      counter))))
(next)          ;;    <--- 1
(next)          ;;    <--- 2
```

We have used here the begin construct that allows us to sequence more than one expressions to evaluate in order, and we used let in order to not expose the variable counter to the rest of the program.

Before we move on, consider the following and explain what goes on:

```
(define (f x)
  (begin (set! x 5) x))
(define b 2)
(f b)       ;  <-- how much is this?
b           ;  <-- how much is this?
```

Structure Mutation

Racket also offers various forms of Structure Mutation, one is **mutable cons cells**. These cons cells must be constructed with a call to mcons instead of cons, their parts can be accessed with calls to mcar and mcdr, and their contents can be changed with a call to set-mcar! or set-mcdr!. As an example, we'll implement delayed evaluation as follows:

- We keep an unevaluated thunk in a mutable cons cell (mcons #f th).
- When we are forced to evaluate the thunk the first time, we change the cell to (mcons #t v) where v is the value we get from evaluating the thunk.

Here is for example code for a delay function and a force function:

```
(define (delay th)
  (mcons #f th))

(define (force cell)
  (if (mcar cell)
      (mcdr cell)
      (let ([v ((mcdr cell))])
        (begin (set-mcar! cell #t)
               (set-mcdr! cell v)
               v))))

(define delayed-b
  (delay (lambda ()
           (begin (print "Will only see this once!")
                  5))))

(force delayed-b)
(force delayed-b)
```

Practice

All these can be done with either mcons cells or set!, but were originally designed to be done with set!.

1. Write a function keepAdding that starts with an internal counter of 0. Each time it is called it takes as argument a number, updates the counter by adding this number and returns the result.
2. Write a function that takes some initial input, and returns a function recall that does the following: Each time it is called, with an argument, it returns the value that was used the last time it was called. For the first call we would return the initial input that was provided to the function that created recall.
3. Write a function stackup that each time it is called with an argument it adds this argument to the list of values from previous calls, starting with an empty list. Each call to the function increases the list length by 1.