Eval and Apply in Racket

`eval`

The `eval` function takes a Racket object evaluates it. Here are some examples. (Try these in the bottom evaluation pane in Racket.)

```
(define fn '*)
(define x 3)
(define y (list '+ x 5))
(define z (list fn 10 y))
x  =>  3
y  => '(+ 3 5)
z  => '(* 10 (+ 3 5))
(eval '(+ 6 6))  => 12
(eval y)  => 8
(eval z)  => 80
```

An example of variables whose values are atoms:

```
(define a 'b)
(define b 'c)
(define c 50)
a => 'b
(eval a)  => 'c
(eval (eval a))  => 50
```

Numbers just evaluate to themselves, so:

```
(eval (eval (eval a)))  => 50
(eval (eval (eval (eval a))))  => 50
```

Quote suppresses evaluation; `eval` causes evaluation. They can cancel each other out.

```
(define x 3)
x  =>  3
'x => 'x
(eval 'x)  => 3
```

Namespaces

The `eval` function used with just one argument evaluates that argument using the top-level bindings for names. You can also call it with an additional optional namespace argument, to cause evaluation in some different environment. See the eval section of the Racket Guide for details. The key point to remember for 341 is that evaluation takes place within some environment that determines the bindings for names.

A catch: when you run Racket in interactive mode, `eval` uses the exports of the `racket` module, so that basic functions like `cons` and `+` will be defined. If you use it directly (e.g. in a definition), however, the initial namespace starts out empty. You can fix this by supplying a namespace argument explicitly:

```
(eval '(+ 3 5) (make-base-namespace))
```

Here's an example of using namespaces in conjunction with defining and using a new variable:

```
(define ns (make-base-namespace))
(eval '(define squid 100) ns)
(eval '(+ squid 8) ns)
```

The last `eval` returns 108.

Notice that both calls to `eval` are using the same namespace -- if we used a different namespace the second time (for example by calling `(make-base-namespace)` again) `squid` wouldn't be defined.

`apply`

The `apply` function applies a function to a list of its arguments. Examples:

```
(apply even? '(3))   => #f
(apply + '(1 2 3 4))  => 10
```

A useful programming trick is to use apply to define a function that takes a list of arguments, if you have available a function that arbitrary number of arguments. Example:

```
(define (sum s) (apply + s))
(sum '(1 2 3))  => 6
```

The read-eval-print-loop

The interactive prompt in Racket (and Scheme) gives you a so-called read-eval-print-loop (REPL). Racket uses the `read` function to read in a Racket object, evaluates it using `eval`, and prints it using `print`. Then prompt again. You'll see the term read-eval-print-loop often in discussing programming language environments (not just for Racket).

The functions `eval` and `apply` interact in a fundamental way in the Racket interpreter. If we are evaluating an ordinary function call (which is represented as a list, of course), we evaluate the first element of the list to get the functions, evaluate the remaining elements to get the arguments, and then use `apply` to apply the function to the arguments. (Notice that this is using call by value semantics, not lazy evaluation as in Haskell -- the arguments get evaluated before applying the function to them.)