Racket I/O

Racket includes various input and output statements, including terminal-based I/O and file I/O. Note that all of these have side effects!

`read` is a function that reads one Racket object from the standard input and returns it.

Example:

```
(define clam (read))
```

There are some other functions to to read characters without parsing them into Racket objects.

Racket provides three different ways to print an instance of a built-in value. According to the [Reading and Writing Racket Data] section of the Guide, these are:

- `print`, which prints a value in the same way that is it printed for a REPL result
- `write`, which prints a value in such a way that read on the output produces the value back
- `display`, which tends to reduce a value to just its character or byte content -- at least for those datatypes that are primarily about characters or bytes, otherwise it falls back to the same output as write.

Read the Guide for details if you need them for an assignment or if you are curious. (You won't need to memorize the differences among them for 341 exams.) One thing to remember is that, for simple datatypes, `print` works so that if you type the output back into the interaction window, you get the same value back. For example, a symbol `squid` will print as

```
'squid
```

so that if you typed in `'squid` in the interaction window (the Read-Eval-Print-Loop) you get the symbol back. Similarly a list with three sea creatures prints as

```
'(octopus squid clam)
```

On the other hand, if you use `write`, Racket doesn't put a quote in front of them, for example

```
squid
```

```
(octopus squid clam)
```

`(newline)` does the same thing as `(display "\n")`.

Multiple Forms

Many of the constructs we've discussed so far, including `define`, `let`, and `cond`, allow multiple forms in their bodies. All but the last form is evaluated just for its side effect, and the value of the last form is used. For example:

```
(define (starfish x)
  (display "this function doesn't do much\n")
  (display "but it does illustrate the point\n")
  (+ x x))
```

Here the display expressions are evaluated (for their effect) and then the last expression `(+ x x)` is evaluated and its value returned as the value of the function. Similarly, there can be multiple forms in the body of a `let`, or the consequent part of a clause in a `cond`. However, `if` must have just a single form in its two arms. (Otherwise, how would Racket tell which was which?)

This capability hasn't been of any use in the past, since we haven't had any side effects -- clearly, the only time one would want to use this is if the forms other than the last have some side effect.