Circle 6

Doing Global Assignments

Heretics imprisoned in flaming tombs inhabit Circle 6.

A global assignment can be performed with `'<<-'`:

```
> x <- 1
> y <- 2
> fun
function () {
   x <- 101
   y <<- 102
}
> fun()
> x
[1] 1
> y
[1] 102
```

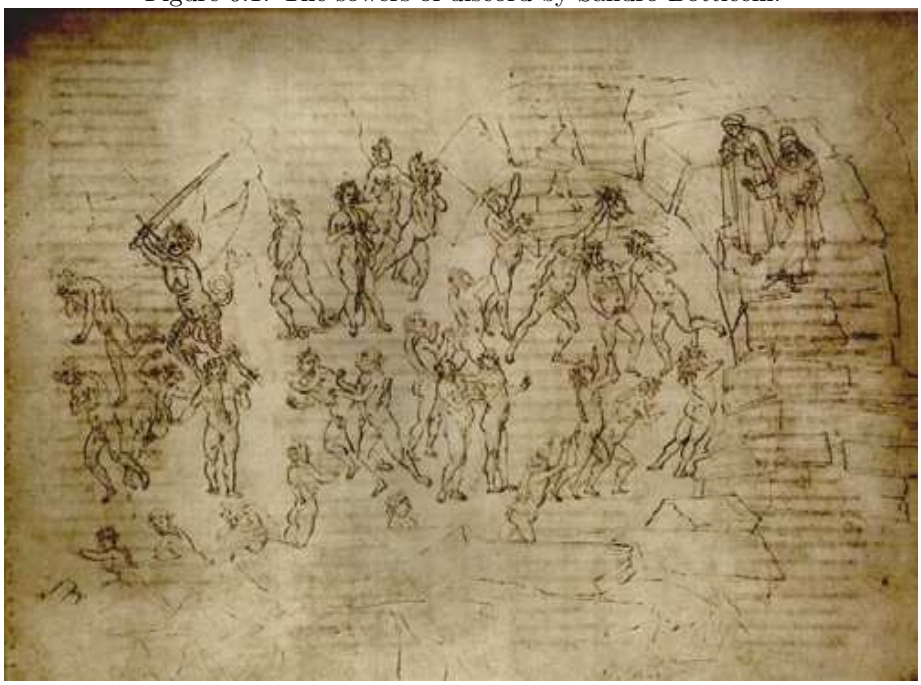
This is life beside a volcano.

If you think you need `'<<-'`, think again. If on reflection you still think you need `'<<-'`, think again. Only when your boss turns red with anger over you not doing anything should you temporarily give in to the temptation. There have been proposals (no more than half-joking) to eliminate `'<<-'` from the language. That would not eliminate global assignments, merely force you to use the `assign` function to achieve them.

What's so wrong about global assignments? Surprise.
Surprise in movies and novels is good. Surprise in computer code is bad.

Except for a few functions that clearly have side effects, it is expected in R that a function has no side effects. A function that makes a global assignment violates this expectation. To users unfamiliar with the code (and even to the writer of the code after a few weeks) there will be an object that changes seemingly by magic.

Figure 6.1: The sowers of discord by Sandro Botticelli.



A particular case where global assignment is useful (and not so egregious) is in memoization. This is when the results of computations are stored so that if the same computation is desired later, the value can merely be looked up rather than recomputed. The global variable is not so worrisome in this case because it is not of direct interest to the user. There remains the problem of name collisions—if you use the same variable name to remember values for two different functions, disaster follows.

In R we can perform memoization by using a locally global variable. ("locally global" is meant to be a bit humorous, but it succinctly describes what is going on.) In this example of computing Fibonacci numbers, we are using the `<<-` operator but using it safely:

```
fibonacci <- local({
   memo <- c(1, 1, rep(NA, 100))
   f <- function(x) {
      if(x == 0) return(0)
      if(x < 0) return(NA)
      if(x > length(memo))
         stop("'x' too big for implementation")
      if(!is.na(memo[x])) return(memo[x])
      ans <- f(x-2) + f(x-1)
      memo[x] <<- ans
```

```
        ans
    }
})
```

So what is this mumbo jumbo saying? We have a function that is just implementing memoization in the naive way using the `'<<-'` operator. But we are hiding the memo object in the environment local to the function. And why is `fibonacci` a function? The return value of something in curly braces is whatever is last. When defining a function we don't generally name the object we are returning, but in this case we need to name the function because it is used recursively.

Now let's use it:

```
> fibonacci(4)
[1] 3
> head(get('memo', envir=environment(fibonacci)))
[1] 1 1 2 3 NA NA
```

From computing the Fibonacci number for 4, the third and fourth elements of `memo` have been filled in. These values will not need to be computed again, a mere lookup suffices.

R always passes by value. It never passes by reference.

There are two types of people: those who understand the preceding paragraph and those who don't.

If you don't understand it, then R is right for you—it means that R is a safe place (notwithstanding the load of things in this document suggesting the contrary). Translated into humanspeak it essentially says that it is dreadfully hard to corrupt data in R. But ingenuity knows no bounds ...

If you do understand the paragraph in question, then you've probably already caught on that the issue is that R is heavily influenced by functional programming—side effects are minimized. You may also worry that this implies hideous memory inefficiency. Well, of course, the paragraph in question is a lie. If it were literally true, then objects (which may be very large) would always be copied when they are arguments to functions. In fact, R attempts to only copy objects when it is necessary, such as when the object is changed inside the function. The paragraph is conceptually true, but not literally true.