SICP section 3.2

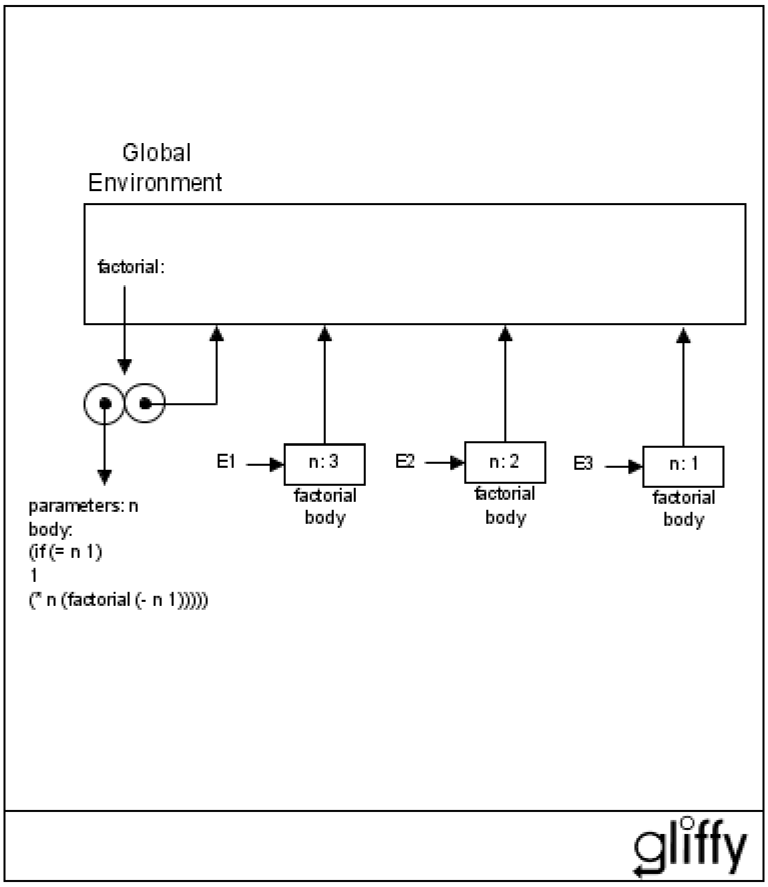
September 28, 2007 at 18:06 Tags SICP

I’m using the excellent online Visio clone Gliffy to draw the diagrams for this section. It takes quite a long time to draw them, but understanding these diagrams is very important. If you’re still not 100% clear on what is exactly meant by *lexical scope*, read this section several times until you understand.

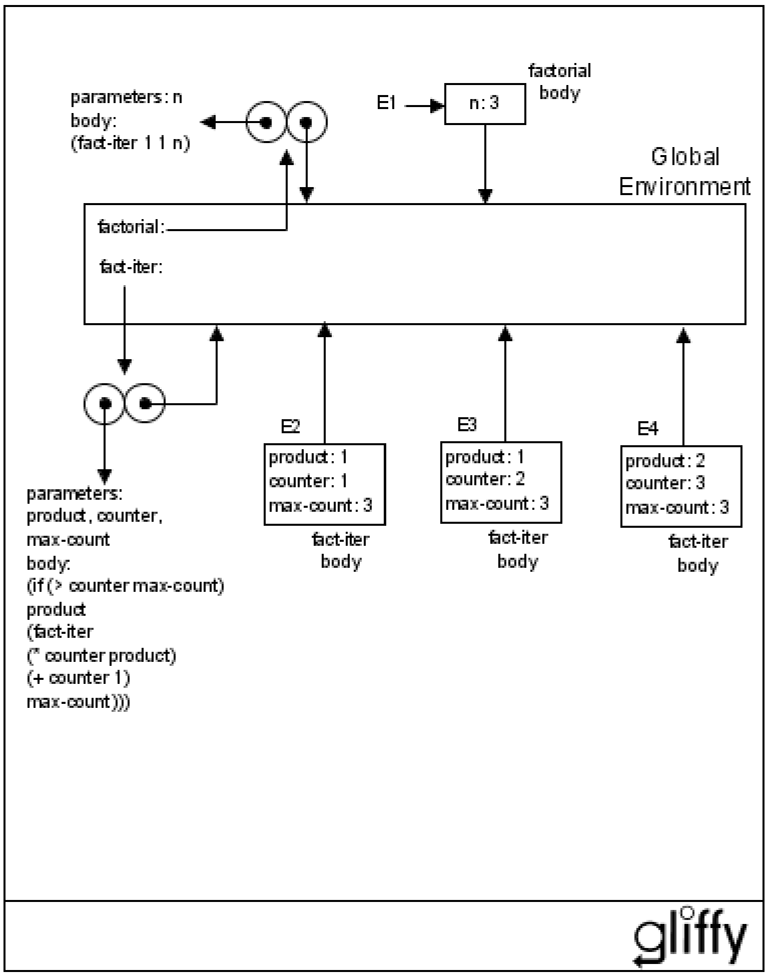
For those who already understand what it is and how the real evaluation model of Lisp works, constructing these diagrams is still important in order to understand the implementation of the Scheme evaluator in chapter 4 of the book.

Exercise 3.9

Since it’s tiresome to draw diagrams, I’ll assume the call is `(factorial 3)` instead of `(factorial 6)`. It illustrates the point just the same. Here’s the environment for the recursive call:



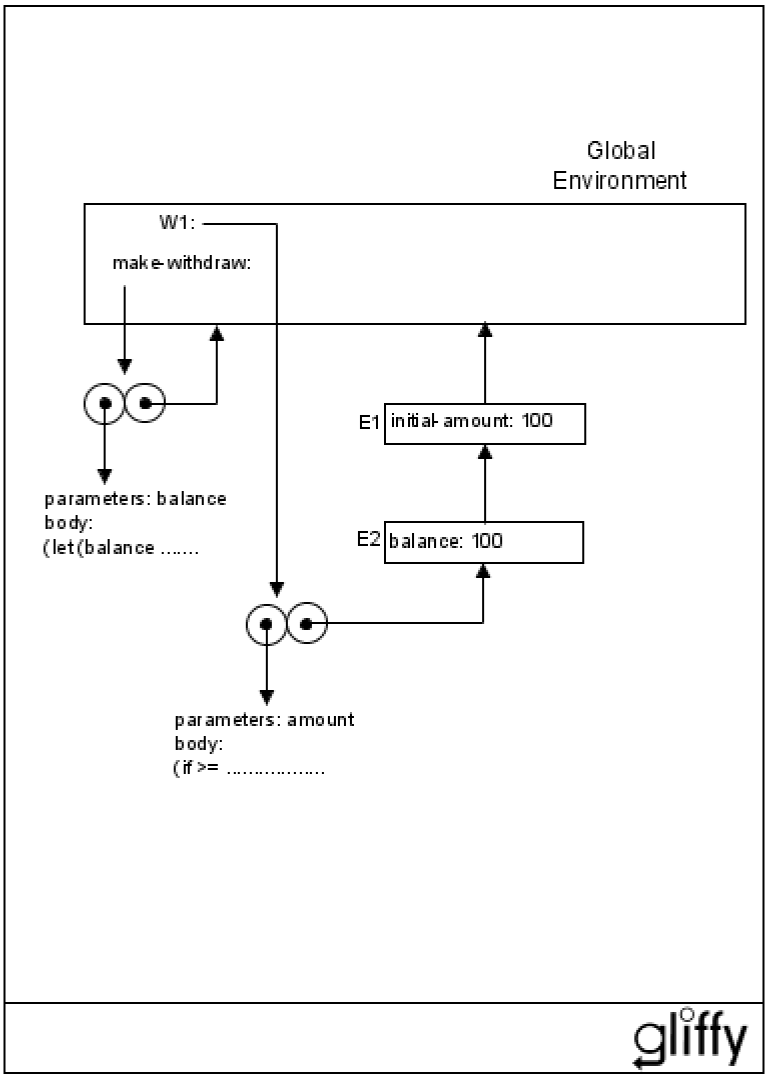
The iterative version produces the following environment diagram:



Exercise 3.10

The result of defining `make-withdraw` is similar to figure 3.6 in the book, except for the body of the `make-withdraw` function which is a little different (as defined in the exercise).

The two versions of `make-withdraw` create objects with the same behavior. To demonstrate the subtle difference, it is enough to consider what happens when
`(define W1 (make-withdraw 100))` is evaluated:



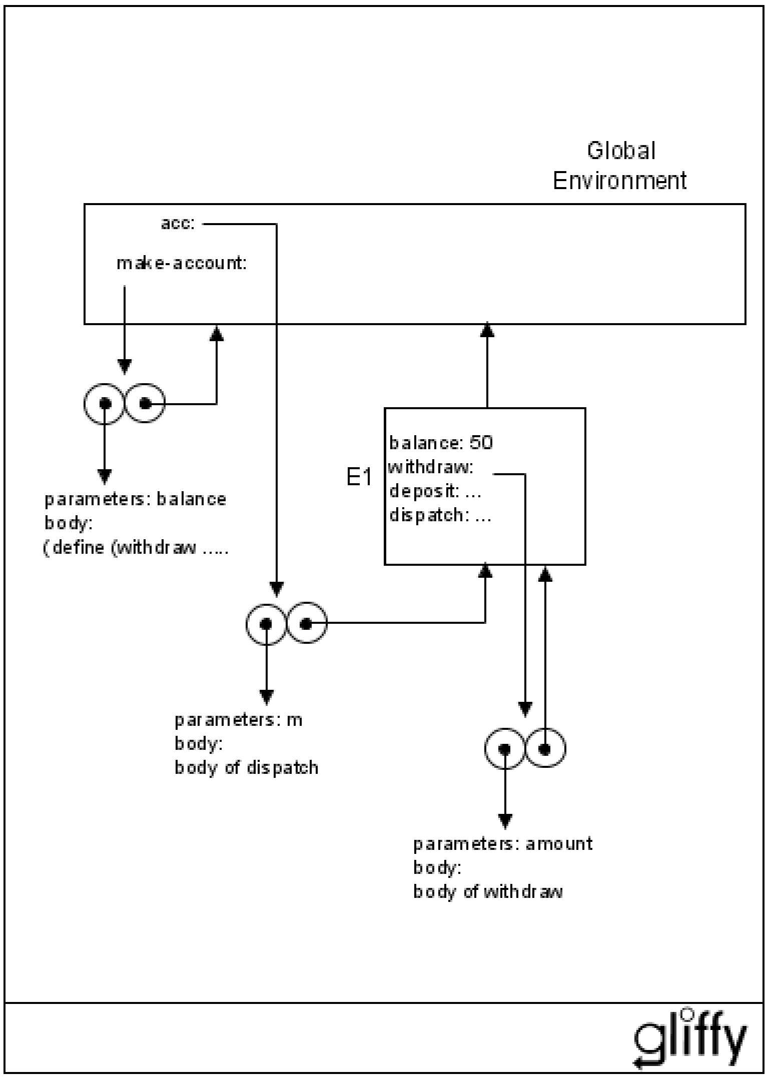
Since `let` is equivalent to `lambda`, another environment is created – in which `balance` is defined. Now the `W1` object has two “private” variables – `balance` and `initial-amount`, and both can be changed separately from one another.

Exercise 3.11

After the line:

```
(define acc (make-account 50))
```

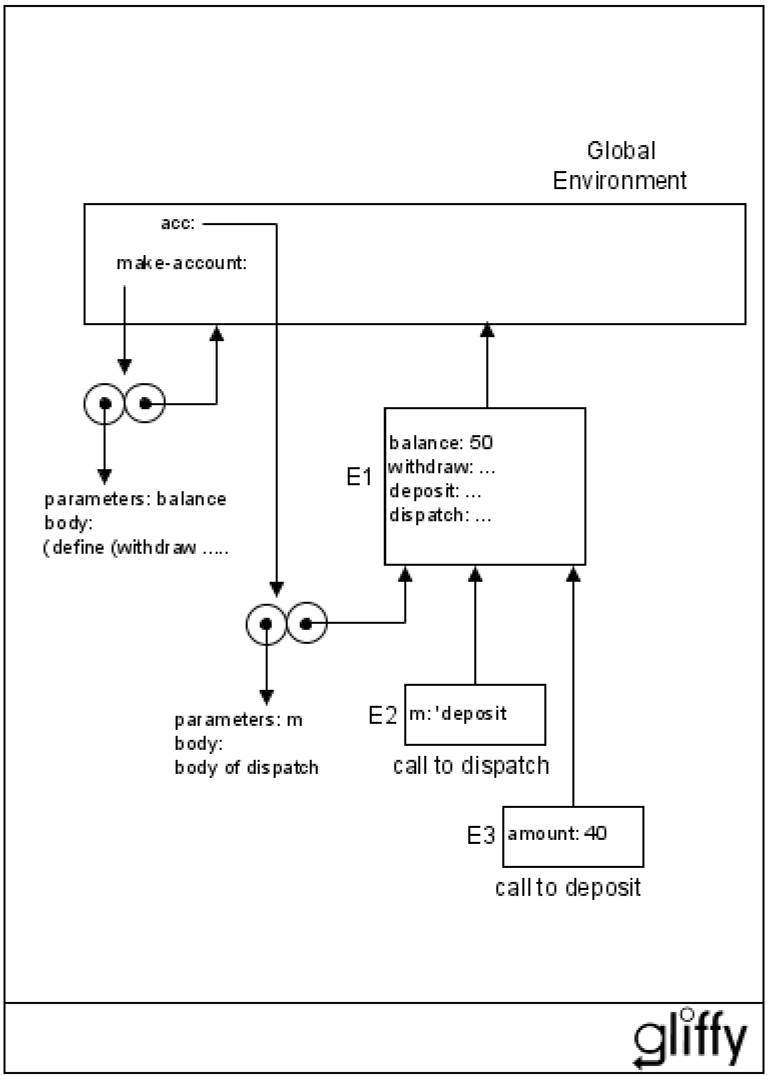
The environment is:



The call:

```
((acc 'deposit) 40)
```

Produces:

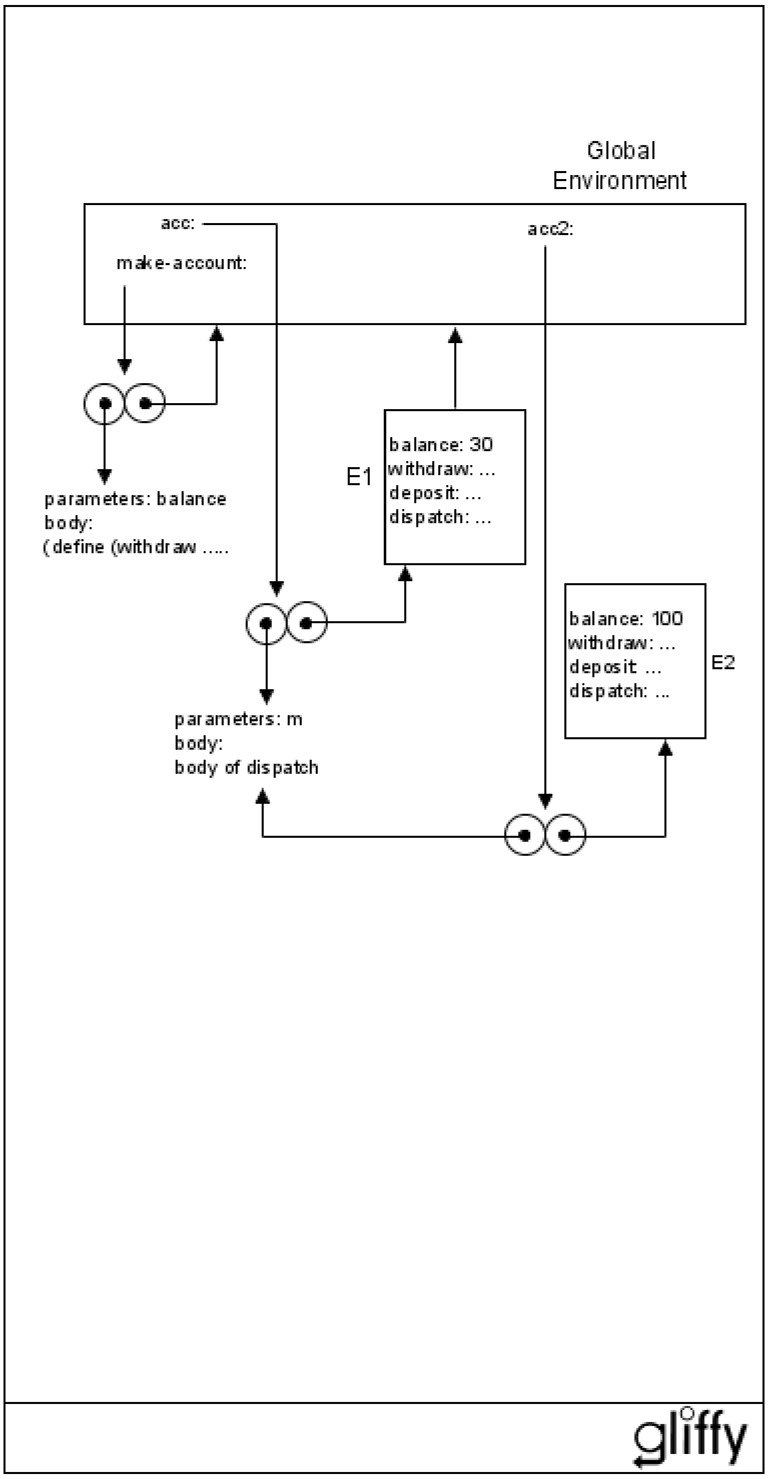


I removed the pointer from `withdraw` to the internal `withdraw` lambda in environment `E1` for clarity. The call `(acc 'deposit)` evaluates to a call to `dispatch` with the argument `deposit`, which is depicted in environment `E2`. Note that the environment's ancestor is `E1` and not the global env, since this `E1` is the env for `acc`.

I'll skip the call to `withdraw` because it's similar. The call:

```
(define acc2 (make-account 100))
```

Generates (the `balance` of `E1` is the result of the `deposit` and `withdraw`):



The accounts in `acc` and `acc2` are distinct because each one has an environment of its own. In drawing this diagram I assumed that the code is shared, although this is an implementation detail, as explained in footnote 15 in the book.

For comments, please send me an email.