Veit's Blog

Scheme Macros III: Defining let

9/20/2017

After having explored how to implement a module system and generic functions in Scheme macros, we'll look this time at how to reimplement let-style local bindings. As a little extra, we'll explore a way of defining them that I've never seen used before, partly because it's somewhat inefficient—but at least it gets rid of mutable state—no `set!` required.

As always, we'll first define an API, and then implement it bit by bit, learning about the intricacies of well-crafted macros along the way.

Some experience with Scheme macros—being able to read through them should suffice—is assumed. Experience with `let` is not required.

The API

`let` is an integral part of any Scheme. It is the standard way to create local bindings for most implementations. Some implement it through macros, some opt for a special form. Today we're going to see how to implement a simple form of `let` and its derivatives as a macro.

```
(let ((x 1)
      (y 2))
  (+ x y))
```

Fig. 1: A simple use case of let.

As you can see in Figure 1, `let` takes a list of pairs, where the first value is the variable name and the second value is what we bind the name to. Simple enough, but fairly powerful, because it gives us something a lot of languages have lacked for a long time: block scope. The values bound by `let` won't leak outside their body, which makes it a powerful building block for abstractions.

This is not the full power of `let`, though. `let` can also be used for looping, like so:

```
(let my-loop ((x 1))
  (if (> x 10)
    (write "We're done!")
    (my-loop (+ x 1))))
```

Fig. 2: A silly loop.

This is very similar to a named function, but it's ephemeral. The name `my-loop`, too, will not leak outside of the context introduced by `let`. We parametrize every call to that function-like thing with the new values for the local bindings, much like we would for a function.

Bindings inside the `let` block do not know of each other. That means that a construct like the following is not valid:

```
(let ((x 1)
      (y (+ x 1))) ; this will complain about the absence of x
  (+ x y))
```

Fig. 3: Blowing up `let`.

All is not lost, however. This is where `let*` comes into play. It is defined for exactly this purpose: being able to reference earlier bindings from later ones. The above construct suddenly becomes valid Scheme if we add that little asterisk.

There is another limitation to `let`, of course: bindings cannot be recursive. While you are able to define functions in `let`, making them call themselves is not allowed. Let’s consider another example:

```
(let ((my-func (lambda (x)
                 (if (x > 10) x (my-func (+ x 1))))))
  (my-func 3))
```

Fig. 4: `let` is letting us down. Geddit? Sorry.

And again, yet another construct comes to the rescue: `letrec`. It allows us to define recursive functions in its bindings. But, of course, we lose the ability of `let*`: we cannot reference later bindings anymore. That’s what `letrec*` is for…

At this point, you might ask yourselves a simple question: why? Why don’t we make `letrec*` the default, if it’s the most capable of all of these constructs? This is where programming language history comes into play: Scheme is very old. If you’ve read some of my earlier musings, you’ll know that Scheme appeared in 1975. That makes it only slightly younger than C (1972, according to Wikipedia), the oldest programming language most of us still use. Back then, efficiency mattered. We say it matters now, but when the best computer most programmers use has 9KB of memory and an add takes 5 microseconds[1], efficiency truly matters—I imagine people took a lot of coffee breaks back then. Now, when efficiency truly matters, you don’t want to waste any cycles for features you don’t need, and sometimes a fancy programming language feature does exactly that. So back in the day it was a great idea to make the least expensive feature the default, and if people needed it, they could upgrade to one of the fancier versions.[2]

For now, let’s try and implement the simplest version first: plain `let`.

let

At it’s core, `let` is quite simple: if we just want to introduce local bindings, we can rewrite `let` expressions to a lambda that takes a bunch of arguments and is immediately called with the values that were provided in the form. Let’s build a simple macro that does exactly this:

```
(define-syntax let
  (syntax-rules ()
    ((let ((var val) ...) body ...)
      ((lambda (var ...) body ...) val ...))))
```

Fig. 5: `let`, implemented.

That doesn't look so bad, does it? Let's walk through it together before we make our version feature-complete.

This definition of `let` makes heavy use of ellipses—the `...`—to destructure the form. We take all of the variable names—named `var` here—and put them in the argument list of the lambda. Then we take the body and set it as the body of the lambda. Finally, we take all of the values—named `val` here—and call our lambda with them. Let's look at the example from Figure 1, before and after transformation:

```
; before:
(let ((x 1)
      (y 2))
  (+ x y))

; after:
((lambda (x y) (+ x y)) 1 2)
```

Fig. 6: The example from Figure 1, before and after macro expansion.

It's that simple! In fact, it's so deceptively simple that it took me a while to believe that this actually was all `let` required to work. I'm still astounded to this day when I look at it.

To digest our marvel, let's look at how to implement the labels from Figure 2. We will add another syntax rule to our definition of `let`, which means we end up with the following definition:

```
(define-syntax let
  (syntax-rules ()
    ; the rule from Figure 5 goes here
    ; ...
    ((let label ((var val) ...) body ...)
      ((lambda ()
        (define label (lambda (var ...) body ...))
        (label val ...))))))
```

Fig. 7: A simplistic definition of labels.

This is also fairly simple! It's likely not the version you'll find in the books, because PLT purists have found a more theoretically sound implementation over the years, but it's good enough for us: it just wraps the expansion in another lambda, so that we have a local closure in which we can call our label. That's a lot of lingo, so let's look at an example of macro expansion:

```
; before:
(let my-loop ((x 1))
  (if (> x 10)
    (write "We're done!")
    (my-loop (+ x 1))))

; after:
((lambda ()
  (define my-loop (lambda (x)
                    (if (> x 10)
                      (write "We're done!")
                      (my-loop (+ x 1)))))
  (my-loop 1)))
```

Fig. 8: The example from Figure 2, before and after expansion.

And that's it! We're done with `let`.[3] And because that didn't require a lot of code, we'll now look at `let`s more handy brother, `let*`.

let*

`let*` isn't that different from `let`. The only thing we need to achieve is having the bindings happen sequentially, instead of concurrently, as they do in `let`. That sounds like a job for nested lambdas!

```
(define-syntax let*
  (syntax-rules ()
    ((let* () body ...) ; base case
      ((lambda () body ...)))
    ((let* ((var val) rest ...) body ...) ; binding case
      ((lambda (var) (let* (rest ...) body ...)) val))))
```

Fig. 9: Implementing sequential bindings with nested lambdas.

Let's walk though the code again—recursive macros can be a bit hard to keep straight. Our base case will be invoked when there are no more bindings to produce; it will create a lambda with the body that was provided to `let*` and immediately call it. This ensures that we don't leak any contents of the body if `let*` is called without bindings. If we used `begin` instead, we might produce unwanted side effects, for instance through `define` expressions.

The other case—I called it the binding case above—takes the head of the bindings, destructures it, and forms an immediately invoked one-argument lambda: our first binding. Inside the lambda, it will call itself recursively with the rest of the binding lists and the function body, until it will finally match the base case.

This should also make obvious why `let*`—at least implemented in this fashion—is more expensive than `let`. Because we build a new closure for every variable we define, we will end up wasting a whole bunch of precious cycles and memory. But, because we assume that those are cheap in our case, we put conceptual clarity first. We could also transform the bindings into a flat closure with a bunch of defines, but this wouldn't satisfy the contract of `let*` to disallow recursive function bindings. You could implement another flavor of `let` pretty cheaply using this pattern, though.

Because the output of this might not be immediately apparent, I will provide you with the expansion input and output of a simple `let*` expression:

```
; before macro expansion:
(let* ((x 1)
       (y (+ x 1)))
  (+ x y))

; after macro expansion
((lambda (x)
  ((lambda (y)
    ((lambda () (+ x y))))
   (+ x y)))
 1)
```

Fig. 10: A simple `let*` expression, before and after expansion (these are the moments where I'd wish for parenthesis-based highlighting on my blog).

This wasn't so bad, was it? If you're not fed up yet, try implementing labels for this form of `let*`. It shouldn't be too different from the version we used for our implementation of `let`.

Let's move on to `letrec*`.

letrec*

This one is the most interesting of all of the macros we're going to implement here. Our version will again be a bit unusual, but straightforward. Consider the implementation of `let*` from Figure 9 again; it's not too far from what we want: all that's left is transforming the bindings from anonymous bindings to named ones. How could we do that? Let's try `define`:

```
(define-syntax letrec*
  (syntax-rules ()
    ((letrec* () body ...) ; base case
      ((lambda () body ...)))
    ((letrec* ((var val) rest ...) body ...) ; binding case
      ((lambda ()
        (define var val)
        (letrec* (rest ...) body ...))))))
```

Fig. 11: `letrec*`, demystified.

As promised this is very similar to `let*` as implemented in Figure 9. We just pushed the binding of `val` to `var` down into the body of the lambda. I'm not completely sold, though. Remember how I said we could implement another flavor of `let` pretty cheaply through flattened closures? This is the time to whip out this little trick. It makes the implementation a bit less pretty to look at, but it's interesting enough for us to try it here:

```
(define-syntax letrec*-helper
  (syntax-rules ()
    ((letrec*-helper () body ...)
      (begin body ...))
    ((letrec*-helper ((var val) rest ...) body ...)
      (begin
        (define var val)
        (letrec*-helper (rest ...) body ...)))))

(define-syntax letrec*
  (syntax-rules ()
    ((letrec* bindings body ...)
      ((lambda ()
        (letrec*-helper bindings body ...))))))
```

Fig. 12: `letrec*`, efficient yet hideous.

That's about an order of magnitude worse! Great, let's walk through it! First we define a helper macro—in my experience that's never a good sign—that does exactly what our first implementation of `letrec*` from Figure 11 does, but we swap the `lambda` expression for `begin`.[4] All that `letrec*` does, then, is wrap the code that the helper generates in a lambda that is immediately called, and we're done.

This macro is even more opaque than the ones before, so I'm sure you all look forward to a manual expansion!

```
; before
(letrec* ((x (lambda (n)
              (if (> n 3) n (x (+ n 1)))))
          (y 1))
  (x y))

; after expanding version 1
((lambda ()
  (define x (lambda (n)
              (if (> n 3) n (x (+ n 1)))))
  ((lambda ()
    (define y 1)
    ((lambda () (x y)))))))

; after expanding version 2
((lambda ()
  (begin
    (define x (lambda (n)
                (if (> n 3) n (x (+ n 1)))))
    (begin
      (define y 1)
      (begin (x y))))))
```

Fig. 13: A manual expansion of version 1 and 2 of `letrec*`.

Figure 13 should shed some light on how version 2 is more efficient than version 1, even if it's not more terse.

Recap

At this point you might ask yourself why we didn't talk about `letrec`. Well, `letrec` is the ugliest of all of the expressions in the `let` family. It requires the implementor to enable recursion, but not sequential bindings. I haven't found it to be tremendously helpful and mostly a pain to implement. Racket even ditched the name `letrec*` and made `letrec` behave like the former instead.

As always, I have a couple of ideas what you could work on to deepen your knowledge of implementing `let` in self-study:

You could implement labels for `let*` and `letrec*`. They should be a very good addition to start out with and pretty rewarding and useful to boot.

You could play around with an implementation based on the Y combinator—that is, the lambda calculus combinator, not the startup accelerator. There is a hint in the footnotes if you need it.[5]

Fin

We just implemented local bindings! I hope it was as much fun for you reading this as it was for me writing it. I don't write much Scheme these days—ever since abandoing zepto, really—, and it's a breath of fresh air to get back into it. Opening my own REPL and hacking away is one of the most rewarding things I do in my spare time, so if you have any suggestions for future posts in this series, ping me and I might write about it!

See you soon!

Footnotes

1. No, really. At least according to the numbers on Wikipedia.

2. My reading of the original lambda papers tells me that `let` was called `labels` back then. I'm not positive the "fancier" versions even existed.

3. Purists will object that there are other—and better—ways to implement labels. I wholeheartedly agree—particularly the versions that use the infamous Y combinator are interesting. They're a little harder to grok, though, and not particularly suited for introductory tutorials like this one. Here is Rosetta Code for implementing anonymous recursion.

4. This version will not end up generating prettier code, but it will be more efficient assuming `begin` is cheaper than calling a function, which it should always be.

5. The Y combinator can be used to assign a name to the function passed to

`let`. You will want to bind the value to a specific call to the combinator. Here be spoilers (hover to reveal): You will want to call the combinator with a lambda that takes an argument named like the variable and the binding as body, like so: `(Y (lambda (var) val))`. Put that in a `let` and bind it to `var`.

Want to go back to the list of posts?