MC28 Lab 3 – The Environment Inspector

Max Hailperin   Karl Knight

Due: October 16, 1995

In this lab you will use a feature of the Scheme system called the *environment inspector* to explore and manipulate environment structures. This will give you some experience with the "real world" analogue of the environment diagrams we've been drawing.

Note: you should do this lab on the NeXTs, since the environment inspector we are describing is the one in SchematiK (actually MIT Scheme), not SchematiX.

The environment inspector is called `where`. The environment inspector accepts a subset of the single-character debugger commands, namely `?`, `A`, `C`, `E`, `O`, `P`, `Q`, `S`, `V`, and `W` (see the debugging handout for a description of each of these commands). In addition to not supporting any of the other commands of `debug`, the other difference with `where` is that you can pass it an argument specifying which environment to inspect. This can either be a procedure, in which case it's associated environment is used, or an actual environment object, which we'll learn more about in a bit.

To give some flavor for what you might do, here is a simple example session:

```
(define (make-adder x)
  (lambda (y)
    (where (the-environment))
    (+ x y)))
;Value: make-adder

(define add3 (make-adder 3))
;Value: add3
```

Notice that we inserted a `(where (the-environment))` at the beginning of the adder procedure which `make-adder` makes, in order that we will have an opportunity at this point to explore the environment structure. The special form `(the-environment)` returns the environment in which it is evaluated (which is the same as the environment where `(+ x y)` will be evaluated), so that is the environment `where` will let us explore when we try using `add3`:

```
(add3 2)

Environment created by a lambda special form
Depth (relative to initial environment): 0
 has bindings:
```

```
y = 2
;You are now in the environment inspector.  Type q to quit, ? for commands.
```

Now we will look at the parent frame with `P`:

```
p

Environment created by the procedure: make-adder
Depth (relative to initial environment): 1
 has bindings:

x = 3
```

We can evaluate any expression in this environment with the `V` command. Here we will use it to evaluate `(the-environment)` to see the actual environment object:

```
v
;Evaluate expression:
(the-environment)
;Value: #[environment 19]
```

The only interesting thing in the printed representation of the environment is the so-called "hash code," here 19. This uniquely identifies the environment frame, so we for example will recognize it as being shared if it reappears as the parent of a different frame. (Note that although this number is unique, the actual value you get may well be other than 19.) We can also use the hash code to specify the environment as an argument to `where`, as we will see later. Now we move up once more to the parent, which we expect to be the global environment.

```
p

Environment named: (user)
Depth (relative to initial environment): 2
```

This frame is the top-level "user" environment in which you usually interact with Scheme. It is the environment frame in which the name names `make-adder` and `add3` are bound, but the environment inspector does not show these, nor any other names you may have defined.

On the other hand, this frame does not include any of the built-in primitives. This is because our Scheme interpreter actually splits what we have been drawing as a single global environment into two levels, this one for the user's global definitions and another one level higher for the built-ins:

```
p

Environment named: ()
Depth (relative to initial environment): 3
```

In this even-more-global environment frame, the environment inspector once again doesn't show any of the bindings.

Now we can exit the environment inspector and let the actual addition happen:

```
q
;Value: 5
```

We can inspect the environment associated with `add3` and see that it is indeed the same as the parent frame we saw above:

```
(where add3)

Environment created by the procedure: make-adder
Depth (relative to initial environment): 0
 has bindings:

x = 3
;You are now in the environment inspector.  Type q to quit, ? for commands.

v
;Evaluate expression:
(the-environment)
Value: #[environment 19]

q
;No value
```

Not only did we see the same binding of `x` to `3`, but we also confirmed that it was identically the same environment by comparing hash codes.

We could also explicitly ask to look at the environment with hash code 19 as follows, even if we didn't know it was `add3`'s environment:

```
(where #@19)

Environment created by the procedure: make-adder
Depth (relative to initial environment): 0
 has bindings:

x = 3
;You are now in the environment inspector.  Type q to quit, ? for commands.
```

Note that the syntax `#@19` is a general way of saying "the object with hash code 19" whether or not it is an environment. This can also be useful for such things as anonymous procedures. (If you evaluate `add3`, you will see something like `#[compound-procedure 2]`, which indicates that the procedure called `add3` has code 2). We can also evaluate a `set!` expression in an environment inspector in order to modify an environment:

```
v
```

```
;Evaluate expression:
(set! x 10)
;Value: 3

q
;No value
```

Now we will see the effect this change has. This time, we'll use the `A` command to see all the environment frames in the current environment at once, rather than stepping through them with `P`:

```
(add3 2)

Environment created by a lambda special form
Depth (relative to initial environment): 0
 has bindings:

y = 2
;You are now in the environment inspector.  Type q to quit, ? for commands.

a
----------------------------------------
Environment created by a lambda special form
Depth (relative to initial environment): 0
 has bindings:

y = 2

----------------------------------------
Environment created by the procedure: make-adder
Depth (relative to initial environment): 1
 has bindings:

x = 10

----------------------------------------
Environment named: (user)
Depth (relative to initial environment): 2

----------------------------------------
Environment named: ()
Depth (relative to initial environment): 3

q
;Value: 12
```

Note that we see the changed binding of `x`, and that it affects the final answer.

1. Use the techniques illustrated above to explore the environment structures involved in the following example (you may want to add one or more strategic `(where (the-environment))`'s to the definition):

```
(define make-scaled-squarer
  (let ((square (lambda (x) (* x x))))
    (lambda (scale)
      (lambda (x)
        (* scale (square x))))))

(define doubled-square (make-scaled-squarer 2))

(define tripled-square (make-scaled-squarer 3))

(doubled-square 5)
;Value: 50

(tripled-square 5)
;Value: 75
```

   How much of the two environments in which `(* scale (square x))` is evaluated is shared? Draw a diagram based on your debugger and/or environment-inspector output, showing whatever sharing you discover. For any environment frames or procedures for which you uncover the hash code, indicate the hash code on the diagram. Turn in the diagram together with your annotated transcript.

2. The following two pieces of code produce the same result as one another but by different mechanisms. Use the debugging tools to show the difference. Explain.

```
(let ((x 3))
  (let ((y 4))
    (set! x 5)
    (+ x y)))
;Value: 9

(let ((x 3))
  (let ((y 4))
    (define x 5)
    (+ x y)))
;Value: 9
```

3. Use the tools discussed in this lab to help you do exercise 3.10 (p. 196) from the book.