Environments in Racket, and the Mutator set!

We describe the structure of environments used to accomplish the creation and application of procedures in Racket, and their interaction with mutators like `set!`. An environment may be thought of as a table of symbols, giving a value (or binding) for each symbol.

The top-level environment and set!

When you first enter Racket, it has an initial top-level (or global) environment that contains the values for symbols like `+` and `car` that represent built-in Racket procedures. When we execute a `define`, the symbol is added to the top-level environment with the given value. For example,

```
> (define x (* 3 3))
```

adds the symbol `x` to the top-level environment, with the value 9. We’ll picture this as follows:

```
<table>
<thead>
<tr>
<th>top-level environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>+  &lt;proc: built-in&gt;</td>
</tr>
<tr>
<td>cons &lt;proc: built-in&gt;</td>
</tr>
<tr>
<td>x       9</td>
</tr>
</tbody>
</table>
```

Of course, you should imagine many other built-in symbols defined in the top-level environment.

In the top-level environment the mutator `set!` changes the value of a previously defined symbol to the value of the expression. For example,

```
> (set! x (* 2 x))
```

evaluates the expression `(* 2 x)`, getting the value 18, and changes the value of `x` in the top-level environment to this value. The top-level environment now may be pictured as follows.

```
<table>
<thead>
<tr>
<th>top-level environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>+  &lt;built-in procedure&gt;</td>
</tr>
<tr>
<td>cons &lt;built-in procedure&gt;</td>
</tr>
<tr>
<td>x       18</td>
</tr>
</tbody>
</table>
```

Note that the value associated with `x` has been changed to 18. From here on, we’ll show only the symbols we are focusing on in the top-level environment.

Creating procedures

We now consider what happens when we define and apply a simple procedure in the top-level environment. Suppose we define the following procedure.
> (define add4 (lambda (n) (+ n 4)))

Alternatively, we could write the definition as follows.

> (define (add4 n) (+ n 4))

This form of definition is “syntactic sugar” for the preceding form; in this section of the notes we will explicitly use \texttt{lambda} everywhere. This definition adds a new symbol \texttt{add4} to the top-level environment, whose value is the procedure of one argument \texttt{n} that returns the sum of \texttt{n} and 4. The procedure is created when the \texttt{lambda} expression is evaluated in the top-level environment. We picture this as follows.

\begin{verbatim}
  top-level environment
  ---------------------
      ...
      x 18
      add4 <proc (n) (+ n 4)>
\end{verbatim}

Note that the procedure’s formal arguments (\texttt{n}) and its body (the expression (\texttt{+ n 4})) are shown in this representation. Now suppose we call this procedure, that is, apply it to an argument as follows.

> (add4 7)

This is a procedure application, so the symbol \texttt{add4} is evaluated by looking it up in the top-level environment, which gives the procedure just described. The other expression, \texttt{7}, in the application is evaluated, giving value \texttt{7} for the actual argument. Now a new local environment is created with an entry for each symbol that is a formal argument of the procedure, and the value of the corresponding actual argument. This new environment can be pictured as:

\begin{verbatim}
  local environment #1
  ---------------------
      n 7
\end{verbatim}

This represents the correspondence between the formal and actual arguments for this call. There is one more important piece of information that we need for a local environment, and that is where next to look for any symbols that are not in the local environment. This we will term the “search pointer” for the environment.

The complete process of evaluating a symbol consists of looking it up in the current environment, and, if it is not found, following the search pointer to another environment to look it up in, and, if it is not found there, following *that* environment’s search pointer, and so on, until the symbol is found, or until we have run out of further environments to search. (In particular, the top-level environment has no search pointer.) The value of the symbol is its value in the environment where it is first found in this search process. If it is not found, it is an unbound variable, and an error message to that effect is generated.

Where does the search pointer for an environment come from? Every procedure has an additional piece of information, which we term its “birth environment”. A (user) procedure is created by the evaluation of a \texttt{lambda} expression in some environment. That environment is the birth environment of the procedure. When a procedure is applied, the local environment created for its formal and actual arguments has its search pointer pointing to the birth environment of the procedure.

So, what is the birth environment of the procedure value that is the value of \texttt{add4}? The \texttt{lambda} expression to create it was evaluated in the top-level environment, so that is the birth environment for the procedure. Incorporating this additional information, we have the following picture.
Now the body of the procedure, the expression (+ n 4), is evaluated in local environment #1. This expression is an application, so we evaluate the symbol + (which is not found in the current environment, but we follow the search pointer and find it in the top-level environment), and the expression n, which is found in the current environment to have a value of 7. The primitive procedure + is applied to the arguments 7 and 4, and returns a value of 11, which is the final value of the application.

> (add4 7)
11
>

Once the procedure returns its value, the local environment #1 is no longer accessible, and may be “garbage collected.” That is, its storage may be freed for re-use, so we will erase it from our subsequent pictures.

**Objects: procedures with state**

We now use these basic mechanisms to implement procedures with state (or memory), which form the basis of an object-oriented approach to programming. We implement a very simple kind of object, a counter. The state of a counter is an integer, and we implement operations to zero, increment, and return the value of a counter.

Here is a procedure that functions as a counter object. (It uses Racket’s “case” syntax for readability.)

```
(define counter
  (let ((count 0))
    (lambda (cmd)
      (case cmd
        [(value) count]
        [(increment!) (set! count (+ 1 count)) count]
        [(zero!) (set! count 0) count]))))
```

Here is an example of using this counter object.

```
> (counter 'value)
0
> (counter 'increment!)
1
> (counter 'increment!)
```
> (counter 'value)
> (counter 'zero!)
> (counter 'value)

How does this work? The define that defines the variable counter adds the variable to the top-level environment, and evaluates the expression following it to determine the value to bind it to. The first thing encountered is the expression (let ((count 0)) ...). This creates a new local environment in which the variable count is bound to the value 0. The search pointer for that local environment points to the environment in which the let expression is being evaluated, in this case, the top-level environment. Then the body of the let, (lambda (cmd) ...), is evaluated in this new local environment, which we show as local environment #1 below.

```
  top-level environment, search: none
  -----------------------------
       ...
       counter <value not determined yet>

  local environment #1, search: top-level environment
  -----------------------------------------
       count 0
```

Evaluating the expression (lambda (cmd) ...) in local environment #1 creates a new procedure, with formal argument list (cmd), body consisting of the expression (case cmd ...), and birth environment pointer pointing to local environment #1. This new procedure becomes the value of counter in the top-level environment. The diagram now looks as follows.

```
  top-level environment, search: none
  -----------------------------
       ...
       counter <proc (cmd) (case ...)>
       birth pointer: local environment #1>

  local environment #1, search: top-level environment
  -----------------------------------------
       count 0
```

The birth pointer for the procedure keeps local environment #1 from being garbage collected. Suppose we then call the counter procedure as follows.

> (counter 'increment!)
1
>
The procedure application \((\text{counter} \ '\text{increment!})\) creates a new local environment (local environment \#2 below) using the formal argument(s) and the actual argument(s) of the procedure and makes its search pointer point to the birth environment of the procedure. It is in this new environment that the body of the procedure (the \((\text{case cmd} \ldots)\) expression) is evaluated to determine the value of the procedure application. At the point that the body is about to be evaluated, the environment structure is as follows.

```
<table>
<thead>
<tr>
<th>Environment</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>top-level</td>
<td>none</td>
</tr>
<tr>
<td>counter</td>
<td>proc (cmd) (case ...)</td>
</tr>
<tr>
<td>birth pointer</td>
<td>local environment #1</td>
</tr>
</tbody>
</table>
```

At the point that the body is about to be evaluated, the environment structure is as follows.

```
<table>
<thead>
<tr>
<th>Environment</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>local environment #1</td>
<td>top-level environment</td>
</tr>
<tr>
<td>count</td>
<td>0</td>
</tr>
</tbody>
</table>

local environment #2, search: local environment #1
-----------------------------

```
<table>
<thead>
<tr>
<th>Environment</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>local environment #2</td>
<td>local environment #1</td>
</tr>
<tr>
<td>cmd</td>
<td>'increment</td>
</tr>
</tbody>
</table>
```

To evaluate the \((\text{case cmd} \ldots)\) expression in local environment #2, the interpreter needs to look up the value of \(\text{cmd}\), which it finds in local environment #2, and then evaluate the expression \((\text{set! count} (+ 1 \text{count}))\). To do this, it evaluates \(+\) by looking it up (finding it in the top-level environment after following two search pointers), \(1\) (to itself, a constant), \(\text{count}\) by looking it up (finding it in local environment #1 after following one search pointer), and then setting the value of \(\text{count}\) in local environment #1 to the result, namely 1. It then looks up \(\text{count}\) again in local environment #1, and finds the value 1, which is returned as the value of the procedure application. After the application completes, local environment #2 is no longer accessible from the top-level environment, and may be garbage collected. The resulting environment structure is now the following.

```
<table>
<thead>
<tr>
<th>Environment</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>top-level</td>
<td>none</td>
</tr>
<tr>
<td>counter</td>
<td>proc (cmd) (case ...)</td>
</tr>
<tr>
<td>birth pointer</td>
<td>local environment #1</td>
</tr>
</tbody>
</table>

local environment #1, search: top-level environment
-----------------------------

```
<table>
<thead>
<tr>
<th>Environment</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>local environment #1</td>
<td>count 1</td>
</tr>
</tbody>
</table>
```

The next time \((\text{counter} \ '\text{increment!})\) is evaluated, it will cause the value of \(\text{count}\) in local environment #1 to be incremented from 1 to 2.

Here is a different procedure that can create several distinct counter objects. We analyze it in terms of environments below.
(define make-counter
  (lambda (count)
    (lambda (command)
      (case command
        ((zero) (set! count 0) count)
        ((increment) (set! count (+ 1 count)) count)
        ((value) count)
        (else 'error)))))

Here is an example of using make-counter to create and use a counter object.

> (define counter1 (make-counter 0))
  > (counter1 'value)
  0
  > (counter1 'increment)
  1
  > (counter1 'increment)
  2
  > (counter1 'value)
  2
  > (counter1 'zero)
  0
  > (counter1 'value)
  0
  > (counter1 'hi)
  'error

Notice that counter1 is created with an initial value of 0. We invoke its operations by calling it as a procedure on one of the symbols zero, increment, or value. It responds by performing the indicated operation to the stored value and returning the updated value. The fact that the procedure has internal state (or memory) is indicated by the fact that it is not a function of its arguments. In particular, the call (counter1 'increment) returns two different values. Any other argument to the procedure counter1 returns the symbol 'error.

We can define another counter object, which will have a separate state. For example we might define counter2 with an initial value of 16 and operate on both counter1 and counter2.

> (define counter2 (make-counter 16))
  > (counter2 'value)
  16
  > (counter1 'value)
  0
  > (counter2 'increment)
  17
  > (counter1 'increment)
  1
Analysis in terms of environments

We apply our description of environments to model what is going on with this example. Assume we start with the initial top-level environment and define `make-counter` as above. Then the top-level environment is as follows.

```
top-level environment, search: none
-----------------------------------
... make-counter <proc (count) (lambda (command).. )
       birth: top-level environment>
```

The procedure that is the value of `make-counter` has a formal argument list containing the symbol `count`, and a body that begins with the inner `lambda` expression. Note that the inner `lambda` expression has not been evaluated yet. The birth environment of the `make-counter` procedure is the top-level environment.

Now suppose we create the first counter object as follows.

```
> (define counter1 (make-counter 0))
```

This defines `counter1` in the top-level environment to be the value of the expression `(make-counter 0)`. This is an application, so a new local environment is created for the argument `count` with value 0, and search pointer pointing to the birth environment of `make-counter`, that is, to the top-level environment. The local environment thus looks as follows.

```
local environment #2, search: top-level environment
-----------------------------------------------
count 0
```

It is in this environment that the body of the `make-counter` is evaluated. The body is the following expression.

```
(lambda (command)
  (case command
    ((zero) (set! count 0) count)
    ((increment) (set! count (+ 1 count)) count)
    ((value) count)
    (else 'error))
```

This is a `lambda` expression, the evaluation of which creates a new procedure. Note that the birth environment of this new procedure value is local environment #2. The created procedure becomes the value bound to `counter1` in the top-level environment. Thus after the `define` is evaluated we have the following structure of environments.
Note that the value of `counter1` is a procedure with one argument, `command`, and body consisting of the expression beginning with `case`, as follows.

\[
\text{(case command}
\begin{align*}
&((\text{zero}) (\text{set! count 0}) \text{ count}) \\
&((\text{increment}) (\text{set! count (+ 1 count)}) \text{ count}) \\
&((\text{value}) \text{ count}) \\
&(\text{else 'error})
\end{align*}
\]

Its birth environment is local environment #2. Because local environment #2 is still accessible from the top-level environment, it cannot be garbage-collected. Suppose we then call the procedure `counter1` to increment the counter as follows.

\[
\text{(counter1 'increment)}
\]

1

This is an application, which looks up `counter1` in the top-level environment and creates a new local environment for its formal and actual arguments. The new local environment has its search pointer pointing to the birth environment of the procedure, namely, local environment #2. Once the new local environment is set up, the complete picture of the environments is as follows.

The body of the procedure (the `case` statement) is evaluated in local environment #3. The `case` statement matches on the symbol `increment`, and the expression `(set! count (+ 1 count))` is evaluated. This looks
up count in local environment #3 and does not find it, following the search pointer to local environment #2, where it is found, with value 0. The set! changes its value to 1 in local environment #2, and then the final expression, count is evaluated, returning a value of 1. This is the value returned for the application of counter1. After the procedure returns, local environment #3 is no longer accessible and may be garbage-collected. At this point, the environment picture is as follows.

```
top-level environment, search: none
-----------------------------------
make-counter <proc (count) (lambda (command).. )
  birth: top-level environment>
counter1 <proc (command) (case ..)
  birth: local environment #2>

local environment #2, search: top-level environment
-----------------------------------
count 1
```

Note that the value of count in local environment #2 has been changed by the set!.

Briefly we now describe what happens when we create another counter object as follows.

```
> (define counter2 (make-counter 16))
```

The application of make-counter creates a new local environment #4 for the arguments, namely, count with value 16. This is the birth environment of the procedure that becomes the value of counter2 in the top-level environment. When the evaluation completes, the environment picture is as follows.

```
top-level environment, search: none
-----------------------------------
make-counter <proc (count) (lambda (command).. )
  birth: top-level environment>
counter1 <proc (command) (case ..)
  birth: local environment #2>
counter2 <proc (command) (case ..)
  birth: local environment #4>

local environment #2, search: top-level environment
-----------------------------------
count 1
local environment #4, search: top-level environment
-----------------------------------
count 16
```

Now suppose we apply counter2 as follows.
(counter2 'increment)

At the point when the body of the `counter2` procedure is about to be evaluated, the environments look as follows.

```plaintext

- top-level environment, search: none
- ... make-counter <proc (count) (lambda (command).. )
  birth: top-level environment>
- counter1 <proc (command) (case ..)
  birth: local environment #2>
- counter2 <proc (command) (case ..)
  birth: local environment #4>

- local environment #2, search: top-level environment
  count  1

- local environment #4, search: top-level environment
  count  16

- local environment #5, search: local environment #4
  command  increment
```

It is in local environment #5 that the body of the `counter2` procedure is evaluated. In this environment, a lookup of `count` finds the symbol defined in local environment #4, pointed to by the search pointer of local environment #5. Thus, it is in local environment #4 that the value will be updated. It is clear that any number of counter objects could be created and used in this way, each having a separate state variable.

This implements lexical scoping of variables in Racket. Non-local references are resolved by considering the expressions that enclose the current one lexically, working “outward.” This is contrasted with dynamic scoping, in which the chain of procedure calls leading to this one is used to resolve non-local references. This particular analysis has been very detailed, but, just as with gates and circuits, you will mostly just use the more abstract patterns without thinking about the details.