Fear of Macros

Greg Hendershott

June 12, 2018

A practical guide to Racket macros.

Copyright (c) 2012-2018 by Greg Hendershott. All rights reserved.

Last updated 2018-06-12T20:14:05

Feedback and corrections are welcome here.

Contents

1 Preface

I learned Racket after 25 years of mostly using C and C++.

Some psychic whiplash resulted.

"All the parentheses" was actually not a big deal. Instead, the first mind warp was functional programming. Before long I wrapped my brain around it, and went on to become comfortable and effective with many other aspects and features of Racket.

But two final frontiers remained: Macros and continuations.

I found that simple macros were easy and understandable, plus there were many good tutorials available. But the moment I stepped past routine pattern-matching, I kind of fell off a cliff into a terminology soup. I marinaded myself in material, hoping it would eventually sink in after enough re-readings. I even found myself using trial and error, rather than having a clear mental model what was going on. Gah.

I'm starting to write this at the point where the shapes are slowly emerging from the fog.

If you have any corrections, criticisms, complaints, or whatever, please let me know.

My primary motive is selfish. Explaining something forces me to learn it more thoroughly. Plus if I write something with mistakes, other people will be eager to point them out and correct me. Is that a social-engineering variation of meta-programming? Next question, please. :)

Finally I do hope it may help other people who have a similar background and/or learning style as me.

I want to show how Racket macro features have evolved as solutions to problems or annoyances. I learn more quickly and deeply when I discover the answer to a question I already have, or find the solution to a problem whose pain I already feel. Therefore I'll give you the questions and problems first, so that you can better appreciate and understand the answers and solutions.

2 Our plan of attack

The macro system you will mostly want to use for production-quality macros is called `syntax-parse`. And don't worry, we'll get to that soon.

But if we start there, you're likely to feel overwhelmed by concepts and terminology, and get very confused. I did.

1. Instead let's start with the basics: A syntax object and a function to change it—a "transformer". We'll work at that level for awhile to get comfortable and to de-mythologize this whole macro business.

2. Soon we'll realize that pattern-matching would make life easier. We'll learn about `syntax-case` and its shorthand cousin, `define-syntax-rule`. We'll discover we can get confused if we want to munge pattern variables before sticking them back in the template, and learn how to do that.

3. At this point we'll be able to write many useful macros. But, what if we want to write the ever-popular anaphoric if, with a "magic variable"? It turns out we've been protected from making certain kind of mistakes. When we want to do this kind of thing on purpose, we use a syntax parameter. [There are other, older ways to do this. We won't look at them. We also won't spend a lot of time advocating "hygiene"—we'll just stipulate that it's good.]

4. Finally, we'll realize that our macros could be smarter when they're used in error. Normal Racket functions optionally can have contracts and types. These catch usage mistakes and provide clear, useful error messages. It would be great if there were something similar for macro. There is. One of the more-recent Racket macro enhancements is `syntax-parse`.

3 Transform!

```
YOU ARE INSIDE A ROOM.
THERE ARE KEYS ON THE GROUND.
THERE IS A SHINY BRASS LAMP NEARBY.

IF YOU GO THE WRONG WAY, YOU WILL BECOME
HOPELESSLY LOST AND CONFUSED.

> pick up the keys

YOU HAVE A SYNTAX TRANSFORMER
```

3.1 What is a syntax transformer?

A syntax transformer is not one of the transformers.

Instead, it is simply a function. The function takes syntax and returns syntax. It transforms syntax.

Here's a transformer function that ignores its input syntax, and always outputs syntax for a string literal:

These examples assume `#lang racket`. If you want to try them using `#lang racket/base`, you'll need to `(require (for-syntax racket`

```
> (define-syntax foo
    (lambda (stx)
      (syntax "I am foo")))
```

Using it:

```
> (foo)
"I am foo"
```

When we use `define-syntax`, we're making a transformer *binding*. This tells the Racket compiler, "Whenever you encounter a chunk of syntax starting with `foo`, please give it to my transformer function, and replace it with the syntax I give back to you." So Racket will give anything that looks like `(foo ...)` to our function, and we can return new syntax to use instead. Much like a search-and-replace.

Maybe you know that the usual way to define a function in Racket:

```
(define (f x) ...)
```

is shorthand for:

```
(define f (lambda (x) ...))
```

That shorthand lets you avoid typing `lambda` and some parentheses.

Well there is a similar shorthand for `define-syntax`:

```
> (define-syntax (also-foo stx)
    (syntax "I am also foo"))
> (also-foo)
"I am also foo"
```

What we want to remember is that this is simply shorthand. We are still defining a transformer function, which takes syntax and returns syntax. Everything we do with macros, will be built on top of this basic idea. It's not magic.

Speaking of shorthand, there is also a shorthand for `syntax`, which is `#'`:

`#'` is short for `syntax` much like `'` is short for `quote`.

```
> (define-syntax (quoted-foo stx)
    #'"I am also foo, using #' instead of syntax")
> (quoted-foo)
"I am also foo, using #' instead of syntax"
```

We'll use the `#'` shorthand from now on.

Of course, we can emit syntax that is more interesting than a string literal. How about returning `(displayln "hi")`?

```
> (define-syntax (say-hi stx)
    #'(displayln "hi"))
> (say-hi)
hi
```

When Racket expands our program, it sees the occurrence of `(say-hi)`, and sees it has a transformer function for that. It calls our function with the old syntax, and we return the new syntax, which is used to evaluate and run our program.

3.2 What's the input?

Our examples so far have ignored the input syntax and output some fixed syntax. But typically we will want to transform the input syntax into something else.

Let's start by looking closely at what the input actually *is*:

```
> (define-syntax (show-me stx)
    (print stx)
    #'(void))
> (show-me '(+ 1 2))
#<syntax:10:0 (show-me (quote (+ 1 2)))>
```

The `(print stx)` shows what our transformer is given: a syntax object.

A syntax object consists of several things. The first part is the S-expression representing the code, such as `'(+ 1 2)`.

Racket syntax is also decorated with some interesting information such as the source file, line number, and column. Finally, it has information about lexical scoping (which you don't need to worry about now, but will turn out to be important later.)

There are a variety of functions available to access a syntax object. Let's define a piece of syntax:

```
> (define stx #'(if x (list "true") #f))
> stx
#<syntax:11:0 (if x (list "true") #f)>
```

Now let's use functions that access the syntax object. The source information functions are:

```
> (syntax-source stx)
'eval
> (syntax-line stx)
11
> (syntax-column stx)
0
```

`(syntax-source stx)` is returning `'eval`, only because of how I'm generating this documentation, using an evaluator to run code snippets in Scribble. Normally this would be something like "my-file.rkt".

More interesting is the syntax "stuff" itself. `syntax->datum` converts it completely into an S-expression:

```
> (syntax->datum stx)
'(if x (list "true") #f)
```

Whereas `syntax-e` only goes "one level down". It may return a list that has syntax objects:

```
> (syntax-e stx)
'(#<syntax:11:0 if> #<syntax:11:0 x> #<syntax:11:0 (list "true")>
#<syntax:11:0 #f>)
```

Each of those syntax objects could be converted by `syntax-e`, and so on recursively—which is what `syntax->datum` does.

In most cases, `syntax->list` gives the same result as `syntax-e`:

```
> (syntax->list stx)
'(#<syntax:11:0 if> #<syntax:11:0 x> #<syntax:11:0 (list "true")>
#<syntax:11:0 #f>)
```

(When would `syntax-e` and `syntax->list` differ? Let's not get side-tracked now.)

When we want to transform syntax, we'll generally take the pieces we were given, maybe rearrange their order, perhaps change some of the pieces, and often introduce brand-new pieces.

3.3 Actually transforming the input

Let's write a transformer function that reverses the syntax it was given:

```
> (define-syntax (reverse-me stx)
    (datum->syntax stx (reverse (cdr (syntax->datum stx)))))
> (reverse-me "backwards" "am" "i" values)
"i"
"am"
"backwards"
```

The `values` at the end of the example allows the result to evaluate nicely. Try `(reverse-me "backwards" "am" "i")` to see why it's handy.

Understand Yoda, can we. Great, but how does this work?

First we take the input syntax, and give it to `syntax->datum`. This converts the syntax into a plain old list:

```
> (syntax->datum #'(reverse-me "backwards" "am" "i" values))
'(reverse-me "backwards" "am" "i" values)
```

Using `cdr` slices off the first item of the list, `reverse-me`, leaving the remainder: `("backwards" "am" "i" values)`. Passing that to `reverse` changes it to `(values "i" "am" "backwards")`:

```
> (reverse (cdr '(reverse-me "backwards" "am" "i" values)))
'(values "i" "am" "backwards")
```

Finally we use `datum->syntax` to convert this back to `syntax`:

```
> (datum->syntax #f '(values "i" "am" "backwards"))
#<syntax (values "i" "am" "backwards")>
```

That's what our transformer function gives back to the Racket compiler, and *that* syntax is evaluated:

```
> (values "i" "am" "backwards")
"i"
"am"
"backwards"
```

The first argument of `syntax->datum` contains the lexical context information that we want to associate with the `syntax` outputted by the transformer. If the first argument is set to `#f` then no lexical context will be associated. Instead of "compile time vs. run time", you may hear it described as "syntax phase vs. runtime phase". Same difference.

3.4 Compile time vs. run time

```
(define-syntax (foo stx)
  (make-pipe) ;Ce n'est pas le temps d'exécution
  #'(void))
```

Normal Racket code runs at ... run time. Duh.

But a syntax transformer is called by Racket as part of the process of parsing, expanding, and compiling our program. In other words, our syntax transformer function is evaluated at compile time.

This aspect of macros lets you do things that simply aren't possible in normal code. One of the classic examples is something like the Racket form, `if`:

```
(if <condition> <true-expression> <false-expression>)
```

If we implemented `if` as a function, all of the arguments would be evaluated before being provided to the function.

```
> (define (our-if condition true-expr false-expr)
    (cond [condition true-expr]
          [else false-expr]))
> (our-if #t
          "true"
          "false")
"true"
```

That seems to work. However, how about this:

```
> (define (display-and-return x)
    (displayln x)
    x)
> (our-if #t
          (display-and-return "true")
          (display-and-return "false"))
true
false
"true"
```

One answer is that functional programming is good, and side-effects are bad. But avoiding side-effects isn't always practical.

Oops. Because the expressions have a side-effect, it's obvious that they are both evaluated. And that could be a problem—what if the side-effect includes deleting a file on disk? You wouldn't want `(if user-wants-file-deleted? (delete-file) (void))` to delete a file even when `user-wants-file-deleted?` is `#f`.

So this simply can't work as a plain function. However a syntax transformer can rearrange the syntax – rewrite the code – at compile time. The pieces of syntax are moved around, but they aren't actually evaluated until run time.

Here is one way to do this:

```
> (define-syntax (our-if-v2 stx)
    (define xs (syntax->list stx))
    (datum->syntax stx `(cond [,(cadr xs) ,(caddr xs)]
                              [else ,(cadddr xs)])))
> (our-if-v2 #t
             (display-and-return "true")
             (display-and-return "false"))
true
"true"
> (our-if-v2 #f
             (display-and-return "true")
             (display-and-return "false"))
false
"false"
```

That gave the right answer. But how? Let's pull out the transformer function itself, and see what it did. We start with an example of some input syntax:

```
> (define stx #'(our-if-v2 #t "true" "false"))
> (displayln stx)
#<syntax:32:0 (our-if-v2 #t "true" "false")>
```

1. We take the original syntax, and use `syntax->list` to change it into a `list` of syntax objects:

```
> (define xs (syntax->list stx))
> (displayln xs)
(#<syntax:32:0 our-if-v2> #<syntax:32:0 #t> #<syntax:32:0 "true">
#<syntax:32:0 "false">)
```

2. To change this into a Racket `cond` form, we need to take the three interesting pieces—the condition, true-expression, and false-expression—from the list using `cadr`, `caddr`, and `cadddr` and arrange them into a `cond` form:

```
`(cond [,(cadr xs) ,(caddr xs)]
       [else ,(cadddr xs)])
```

3. Finally, we change that into `syntax` using `datum->syntax`:

```
> (datum->syntax stx `(cond [,(cadr xs) ,(caddr xs)]
                            [else ,(cadddr xs)]))
#<syntax (cond (#t "true") (else "fals...>
```

So that works, but using `cadddr` etc. to destructure a list is painful and error-prone. Maybe you know Racket's `match`? Using that would let us do pattern-matching.

Instead of:

```
> (define-syntax (our-if-v2 stx)
    (define xs (syntax->list stx))
    (datum->syntax stx `(cond [,(cadr xs) ,(caddr xs)]
                              [else ,(cadddr xs)])))
```

We can write:

```
> (define-syntax (our-if-using-match stx)
    (match (syntax->list stx)
      [(list name condition true-expr false-expr)
       (datum->syntax stx `(cond [,condition ,true-expr]
                                 [else ,false-expr]))]))
```

Notice that we don't care about the first item in the syntax list. We didn't take `(car xs)` in our-if-v2, and we didn't use `name` when we used pattern-matching. In general, a syntax transformer won't care about that, because it is the name of the transformer binding. In other words, a macro usually doesn't care about its own name.

Great. Now let's try using it:

```
> (our-if-using-match #t "true" "false")
match: undefined;
 cannot reference an identifier before its definition
  in module: 'program
  phase: 1
```

Oops. It's complaining that `match` isn't defined.

Our transformer function is working at compile time, not run time. And at compile time, only `racket/base` is required for you automatically—not the full `racket`.

Anything beyond `racket/base`, we have to require ourselves—and require it for compile time using the `for-syntax` form of `require`.

In this case, instead of using plain `(require racket/match)`, we want `(require (for-syntax racket/match))`—the `for-syntax` part meaning, "for compile time".

So let's try that:

```
> (require (for-syntax racket/match))
> (define-syntax (our-if-using-match-v2 stx)
    (match (syntax->list stx)
      [(list _ condition true-expr false-expr)
       (datum->syntax stx `(cond [,condition ,true-expr]
                                 [else ,false-expr]))]))
> (our-if-using-match-v2 #t "true" "false")
"true"
```

Joy.

3.5 `begin-for-syntax`

We used `for-syntax` to `require` the `racket/match` module because we needed to use `match` at compile time.

What if we wanted to define our own helper function to be used by a macro? One way to do that is put it in another module, and `require` it using `for-syntax`, just like we did with the `racket/match` module.

If instead we want to put the helper in the same module, we can't simply `define` it and use it—the definition would exist at run time, but we need it at compile time. The answer is to put the definition of the helper function(s) inside `begin-for-syntax`:

```
(begin-for-syntax
 (define (my-helper-function ....)
   ....))
(define-syntax (macro-using-my-helper-function stx)
  (my-helper-function ....)
  ....)
```

In the simple case, we can also use `define-for-syntax`, which composes `begin-for-syntax` and `define`:

```
(define-for-syntax (my-helper-function ....)
  ....)
(define-syntax (macro-using-my-helper-function stx)
  (my-helper-function ....)
  ....)
```

To review:

- Syntax transformers work at compile time, not run time. The good news is this means we can do things like rearrange the pieces of syntax without evaluating them. We can implement forms like `if` that simply couldn't work properly as run time functions.

- More good news is that there isn't some special, weird language for writing syntax transformers. We can write these transformer functions using the Racket language we already know and love.
- The semi-bad news is that the familiarity can make it easy to forget that we're not working at run time. Sometimes that's important to remember.
  - For example only `racket/base` is required for us automatically. If we need other modules, we have to require them, and do so *for compile time* using `for-syntax`.
  - Similarly, if we want to define helper functions in the same file/module as the macros that use them, we need to wrap the definitions inside a `begin-for-syntax` form. Doing so makes them available at compile time.

4 Pattern matching: syntax-case and syntax-rules

Most useful syntax transformers work by taking some input syntax, and rearranging the pieces into something else. As we saw, this is possible but tedious using list accessors such as `cadddr`. It's more convenient and less error-prone to use `match` to do pattern-matching.

Historically, `syntax-case` and `syntax-rules` pattern matching came first. `match` was added to Racket later.

It turns out that pattern-matching was one of the first improvements to be added to the Racket macro system. It's called `syntax-case`, and has a shorthand for simple situations called `define-syntax-rule`.

Recall our previous example:

```
(require (for-syntax racket/match))
(define-syntax (our-if-using-match-v2 stx)
  (match (syntax->list stx)
    [(list _ condition true-expr false-expr)
     (datum->syntax stx `(cond [,condition ,true-expr]
                               [else ,false-expr]))]))
```

Here's what it looks like using `syntax-case`:

```
> (define-syntax (our-if-using-syntax-case stx)
    (syntax-case stx ()
      [(_ condition true-expr false-expr)
       #'(cond [condition true-expr]
               [else false-expr])]))
> (our-if-using-syntax-case #t "true" "false")
"true"
```

Pretty similar, huh? The pattern matching part looks almost exactly the same. The way we specify the new syntax is simpler. We don't need to do quasi-quoting and unquoting. We don't need to use `datum->syntax`. Instead, we supply a "template", which uses variables from the pattern.

There is a shorthand for simple pattern-matching cases, which expands into `syntax-case`. It's called `define-syntax-rule`:

```
> (define-syntax-rule (our-if-using-syntax-rule condition true-
expr false-expr)
    (cond [condition true-expr]
          [else false-expr]))
> (our-if-using-syntax-rule #t "true" "false")
"true"
```

Here's the thing about `define-syntax-rule`. Because it's so simple, `define-syntax-rule` is often the first thing people are taught about macros. But it's almost deceptively

simple. It looks so much like defining a normal run time function—yet it's not. It's working at compile time, not run time. Worse, the moment you want to do more than `define-syntax-rule` can handle, you can fall off a cliff into what feels like complicated and confusing territory. Hopefully, because we started with a basic syntax transformer, and worked up from that, we won't have that problem. We can appreciate `define-syntax-rule` as a convenient shorthand, but not be scared of, or confused about, that for which it's shorthand.

Most of the materials I found for learning macros, including the Racket *Guide*, do a very good job explaining how patterns and templates work. So I won't regurgitate that here.

Sometimes, we need to go a step beyond the pattern and template. Let's look at some examples, how we can get confused, and how to get it working.

4.1 Pattern variable vs. template—fight!

Let's say we want to define a function with a hyphenated name, a-b, but we supply the a and b parts separately. The Racket `struct` macro does something like this: `(struct foo (field1 field2))` automatically defines a number of functions whose names are variations on the name `foo`—such as `foo-field1`, `foo-field2`, `foo?`, and so on.

So let's pretend we're doing something like that. We want to transform the syntax `(hyphen-define a b (args) body)` to the syntax `(define (a-b args) body)`.

A wrong first attempt is:

```
> (define-syntax (hyphen-define/wrong1 stx)
    (syntax-case stx ()
      [(_ a b (args ...) body0 body ...)
       (let ([name (string->symbol (format "~a-~a" a b))])
         #'(define (name args ...)
             body0 body ...))]))
eval:47:0: a: pattern variable cannot be used outside of a
template
  in: a
```

Huh. We have no idea what this error message means. Well, let's try to work it out. The "template" the error message refers to is the `#'(define (name args ...) body0 body ...)` portion. The `let` isn't part of that template. It sounds like we can't use `a` (or `b`) in the `let` part.

In fact, `syntax-case` can have as many templates as you want. The obvious, required template is the final expression supplying the output syntax. But you can use `syntax` (a.k.a. `#'`) on a pattern variable. This makes another template, albeit a small, "fun size" template. Let's try that:

```
> (define-syntax (hyphen-define/wrong1.1 stx)
    (syntax-case stx ()
      [(_ a b (args ...) body0 body ...)
       (let ([name (string->symbol (format "~a-~a" #'a #'b))])
         #'(define (name args ...)
             body0 body ...))]))
```

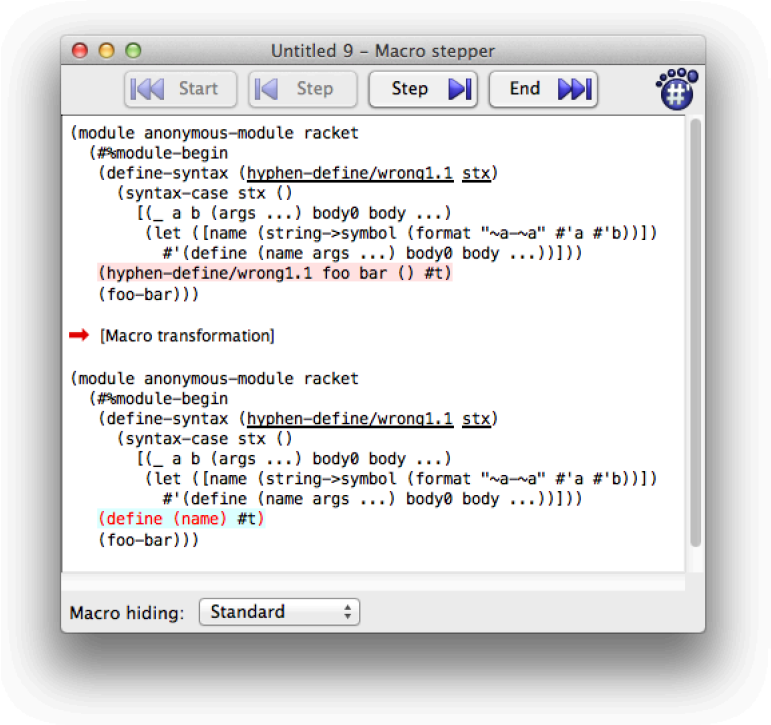
No more error—good! Let's try to use it:

```
> (hyphen-define/wrong1.1 foo bar () #t)
> (foo-bar)
foo-bar: undefined;
 cannot reference an identifier before its definition
  in module: 'program
```

Apparently our macro is defining a function with some name other than `foo-bar`. Huh.

This is where the Macro Stepper in DrRacket is invaluable.

Even if you prefer mostly to use Emacs, this is a situation where it's definitely worth temporarily using DrRacket for its Macro Stepper.



The Macro Stepper says that the use of our macro:

```
(hyphen-define/wrong1.1 foo bar () #t)
```

expanded to:

```
(define (name) #t)
```

Well that explains it. Instead, we wanted to expand to:

```
(define (foo-bar) #t)
```

Our template is using the symbol `name` but we wanted its value, such as `foo-bar` in this use of our macro.

Is there anything we already know that behaves like this—where using a variable in the template yields its value? Yes: Pattern variables. Our pattern doesn't include `name` because we don't expect it in the original syntax—indeed the whole point of this macro is to create it. So `name` can't be in the main pattern. Fine—let's make an *additional* pattern. We can do that using an additional, nested `syntax-case`:

```
> (define-syntax (hyphen-define/wrong1.2 stx)
    (syntax-case stx ()
      [(_ a b (args ...) body0 body ...)
       (syntax-case (datum->syntax #'a
                                   (string->symbol (format "~a-
~a" #'a #'b)))
                    ()
         [name #'(define (name args ...)
                   body0 body ...)])]))
```

Looks weird? Let's take a deep breath. Normally our transformer function is given syntax by Racket, and we pass that syntax to `syntax-case`. But we can also create some syntax of our own, on the fly, and pass *that* to `syntax-case`. That's all we're doing here. The whole `(datum->syntax ...)` expression is syntax that we're creating on the fly. We can give that to `syntax-case`, and match it using a pattern variable named `name`. Voila, we have a new pattern variable. We can use it in a template, and its value will go in the template.

We might have one more—just one, I promise!—small problem left. Let's try to use our new version:

```
> (hyphen-define/wrong1.2 foo bar () #t)
> (foo-bar)
foo-bar: undefined;
 cannot reference an identifier before its definition
  in module: 'program
```

Hmm. `foo-bar` is *still* not defined. Back to the Macro Stepper. It says now we're expanding to:

```
(define (|#<syntax:11:24foo>-#<syntax:11:28 bar>|) #t)
```

Oh right: `#'a` and `#'b` are syntax objects. Therefore

```
(string->symbol (format "~a-~a" #'a #'b))
```

is the printed form of both syntax objects, joined by a hyphen:

```
|#<syntax:11:24foo>-#<syntax:11:28 bar>|
```

Instead we want the datum in the syntax objects, such as the symbols `foo` and `bar`. Which we get using `syntax->datum`:

```
> (define-syntax (hyphen-define/ok1 stx)
    (syntax-case stx ()
      [(_ a b (args ...) body0 body ...)
       (syntax-case (datum->syntax #'a
                                   (string->symbol (format "~a-
~a"
                                                           (syntax-
>datum #'a)
                                                           (syntax-
>datum #'b))))
                    ()
         [name #'(define (name args ...)
                   body0 body ...)]) ]))
> (hyphen-define/ok1 foo bar () #t)
> (foo-bar)
#t
```

And now it works!

Next, some shortcuts.

4.1.1 `with-syntax`

Instead of an additional, nested `syntax-case`, we could use `with-syntax`. This rearranges the `syntax-case` to look more like a `let` statement—first the name, then the value. Also it's more convenient if we need to define more than one pattern variable.

Another name for `with-syntax` could be, "with new pattern variable".

```
> (define-syntax (hyphen-define/ok2 stx)
    (syntax-case stx ()
      [(_ a b (args ...) body0 body ...)
```

```
        (with-syntax ([name (datum->syntax #'a
                                           (string-
>symbol (format "~a-~a"
                                                                   (syntax-
>datum #'a)
                                                                   (syntax-
>datum #'b))))])
          #'(define (name args ...)
              body0 body ...))]))
> (hyphen-define/ok2 foo bar () #t)
> (foo-bar)
#t
```

Again, `with-syntax` is simply `syntax-case` rearranged:

```
(syntax-case <syntax> () [<pattern> <body>])
(with-syntax ([<pattern> <syntax>]) <body>)
```

Whether you use an additional `syntax-case` or use `with-syntax`, either way you are simply defining additional pattern variables. Don't let the terminology and structure make it seem mysterious.

4.1.2 `with-syntax*`

We know that `let` doesn't let us use a binding in a subsequent one:

```
> (let ([a 0]
        [b a])
    b)
a: undefined;
 cannot reference an identifier before its definition
  in module: 'program
```

Instead we can nest `let`s:

```
> (let ([a 0])
    (let ([b a])
      b))
0
```

Or use a shorthand for nesting, `let*`:

```
> (let* ([a 0]
```

```
          [b a])
    b)
0
```

Similarly, instead of writing nested `with-syntax`s, we can use `with-syntax*`:

```
> (require (for-syntax racket/syntax))
> (define-syntax (foo stx)
    (syntax-case stx ()
      [(_ a)
        (with-syntax* ([b #'a]
                       [c #'b])
          #'c)]))
```

One gotcha is that `with-syntax*` isn't provided by `racket/base`. We must `(require (for-syntax racket/syntax))`. Otherwise we may get a rather bewildering error message:

```
...: ellipses not allowed as an expression in: ....
```

4.1.3 format-id

There is a utility function in `racket/syntax` called `format-id` that lets us format identifier names more succinctly than what we did above:

```
> (require (for-syntax racket/syntax))
> (define-syntax (hyphen-define/ok3 stx)
    (syntax-case stx ()
      [(_ a b (args ...) body0 body ...)
       (with-syntax ([name (format-id #'a "~a-~a" #'a #'b)])
         #'(define (name args ...)
             body0 body ...))]))
> (hyphen-define/ok3 bar baz () #t)
> (bar-baz)
#t
```

Using `format-id` is convenient as it handles the tedium of converting from syntax to symbol datum to string ... and all the way back.

The first argument of `format-id`, `lctx`, is the lexical context of the identifier that will be created. You almost never want to supply `stx` — the overall chunk of syntax that the macro transforms. Instead you want to supply some more-specific bit of syntax, such as an identifier that the user has provided to the macro. In this example, we're using `#'a`. The

resulting identifier will have the same scope as that which the user provided. This is more likely to behave as the user expects, especially when our macro is composed with other macros.

4.1.4 Another example

Finally, here's a variation that accepts an arbitrary number of name parts to be joined with hyphens:

```
> (require (for-syntax racket/string racket/syntax))
> (define-syntax (hyphen-define* stx)
    (syntax-case stx ()
      [(_ (names ...) (args ...) body0 body ...)
       (let ([name-stxs (syntax->list #'(names ...))])
         (with-syntax ([name (datum->syntax (car name-stxs)
                                            (string->symbol
                                             (string-
join (for/list ([name-stx name-stxs])
                                                                (symbol-
>string
                                                                 (syntax-
e name-stx)))
                                                               "-"))))])
           #`(define (name args ...)
               body0 body ...))]))
> (hyphen-define* (foo bar baz) (v) (* 2 v))
> (foo-bar-baz 50)
100
```

Just as when we used `format-id`, when using `datum->syntax` we're being careful with the first, `lctx` argument. We want the identifier we create to use the lexical context of an identifier provided to the macro by the user. In this case, the user's identifiers are in the `(names ...)` template variable. We change this from one `syntax` into a `list` of `syntax`es. The first element we use for the lexical context. Then of course we'll use all the elements to form the hyphenated identifier.

To review:

- You can't use a pattern variable outside of a template. But you can use `syntax` or `#'` on a pattern variable to make an ad hoc, "fun size" template.
- If you want to munge pattern variables for use in the template, `with-syntax` is your friend, because it lets you create new pattern variables.
- Usually you'll need to use `syntax->datum` to get the interesting value inside.

- `format-id` is convenient for formatting identifier names.

4.2 Making our own `struct`

Let's apply what we just learned to a more-realistic example. We'll pretend that Racket doesn't already have a `struct` capability. Fortunately, we can write a macro to provide our own system for defining and using structures. To keep things simple, our structure will be immutable (read-only) and it won't support inheritance.

Given a structure declaration like:

```
(our-struct name (field1 field2 ...))
```

We need to define some procedures:

- A constructor procedure whose name is the struct name. We'll represent structures as a `vector`. The structure name will be element zero. The fields will be elements one onward.
- A predicate, whose name is the struct name with `?` appended.
- For each field, an accessor procedure to get its value. These will be named struct-field (the name of the struct, a hyphen, and the field name).

```
> (require (for-syntax racket/syntax))
> (define-syntax (our-struct stx)
    (syntax-case stx ()
      [(_ id (fields ...))
       (with-syntax ([pred-id (format-id #'id "~a?" #'id)])
         #`(begin
             ; Define a constructor.
             (define (id fields ...)
               (apply vector (cons 'id  (list fields ...))))
             ; Define a predicate.
             (define (pred-id v)
               (and (vector? v)
                    (eq? (vector-ref v 0) 'id)))
             ; Define an accessor for each field.
             #,@(for/list ([x (syntax->list #'(fields ...))]
                           [n (in-naturals 1)])
                  (with-syntax ([acc-id (format-id #'id "~a-
~a" #'id x)]
                                [ix n])
                    #`(define (acc-id v)
```

```
                        (unless (pred-id v)
                          (error 'acc-id "~a is not a ~a
struct" v 'id))
                        (vector-ref v ix))))))]))
; Test it out
> (require rackunit)
> (our-struct foo (a b))
> (define s (foo 1 2))
> (check-true (foo? s))
> (check-false (foo? 1))
> (check-equal? (foo-a s) 1)
> (check-equal? (foo-b s) 2)
> (check-exn exn:fail?
             (lambda () (foo-a "furble")))
; The tests passed.
; Next, what if someone tries to declare:
> (our-struct "blah" ("blah" "blah"))
format-id: contract violation
  expected: (or/c string? symbol? identifier? keyword? char?
number?)
  given: #<syntax:83:0 "blah">
```

The error message is not very helpful. It's coming from `format-id`, which is a private implementation detail of our macro.

You may know that a `syntax-case` clause can take an optional "guard" or "fender" expression. Instead of

```
[pattern template]
```

It can be:

```
[pattern guard template]
```

Let's add a guard expression to our clause:

```
> (require (for-syntax racket/syntax))
> (define-syntax (our-struct stx)
    (syntax-case stx ()
      [(_ id (fields ...))
       ; Guard or "fender" expression:
       (for-each (lambda (x)
                   (unless (identifier? x)
                     (raise-syntax-error #f "not an identi-
fier" stx x)))
```

```
                (cons #'id (syntax->list #'(fields ...))))
        (with-syntax ([pred-id (format-id #'id "~a?" #'id)])
          #`(begin
              ; Define a constructor.
              (define (id fields ...)
                (apply vector (cons 'id  (list fields ...))))
              ; Define a predicate.
              (define (pred-id v)
                (and (vector? v)
                     (eq? (vector-ref v 0) 'id)))
              ; Define an accessor for each field.
              #,@(for/list ([x (syntax->list #'(fields ...))]
                            [n (in-naturals 1)])
                   (with-syntax ([acc-id (format-id #'id "~a-
~a" #'id x)]
                                 [ix n])
                     #`(define (acc-id v)
                         (unless (pred-id v)
                           (error 'acc-id "~a is not a ~a
struct" v 'id))
                         (vector-ref v ix))))))
; Now the same misuse gives a better error message:
> (our-struct "blah" ("blah" "blah"))
eval:86:0: our-struct: not an identifier
  at: "blah"
  in: (our-struct "blah" ("blah" "blah"))
```

Later, we'll see how `syntax-parse` makes it even easier to check usage and provide helpful messages about mistakes.

4.3 Using dot notation for nested hash lookups

The previous two examples used a macro to define functions whose names were made by joining identifiers provided to the macro. This example does the opposite: The identifier given to the macro is split into pieces.

If you write programs for web services you deal with JSON, which is represented in Racket by a `jsexpr?`. JSON often has dictionaries that contain other dictionaries. In a `jsexpr?` these are represented by nested `hasheq` tables:

```
; Nested `hasheq's typical of a jsexpr:
> (define js (hasheq 'a (hasheq 'b (hasheq 'c "value"))))
```

In JavaScript you can use dot notation:

```
foo = js.a.b.c;
```

In Racket it's not so convenient:

```
(hash-ref (hash-ref (hash-ref js 'a) 'b) 'c)
```

We can write a helper function to make this a bit cleaner:

```
; This helper function:
> (define/contract (hash-refs h ks [def #f])
    ((hash? (listof any/c)) (any/c) . ->* . any)
    (with-handlers ([exn:fail? (const (cond [(procedure? def) (def)]
                                            [else def]))])
      (for/fold ([h h])
        ([k (in-list ks)])
        (hash-ref h k))))
; Lets us say:
> (hash-refs js '(a b c))
"value"
```

That's better. Can we go even further and use a dot notation somewhat like JavaScript?

```
; This macro:
> (require (for-syntax racket/syntax))
> (define-syntax (hash.refs stx)
    (syntax-case stx ()
      ; If the optional 'default' is missing, use #f.
      [(_ chain)
       #'(hash.refs chain #f)]
      [(_ chain default)
       (let* ([chain-str (symbol->string (syntax->datum #'chain))]
              [ids (for/list ([str (in-list (regexp-
split #rx"\\." chain-str))])
                     (format-id #'chain "~a" str))])
         (with-syntax ([hash-table (car ids)]
                       [keys       (cdr ids)])
           #'(hash-refs hash-table 'keys default)))]))
; Gives us "sugar" to say this:
> (hash.refs js.a.b.c)
"value"
; Try finding a key that doesn't exist:
> (hash.refs js.blah)
#f
; Try finding a key that doesn't exist, specifying the default:
> (hash.refs js.blah 'did-not-exist)
'did-not-exist
```

It works!

We've started to appreciate that our macros should give helpful messages when used in error. Let's try to do that here.

```
> (require (for-syntax racket/syntax))
> (define-syntax (hash.refs stx)
    (syntax-case stx ()
      ; Check for no args at all
      [(_)
       (raise-syntax-error #f "Expected hash.key0[.key1 ...] [de-
fault]" stx #'chain)]
      ; If the optional `default' is missing, use #f.
      [(_ chain)
       #'(hash.refs chain #f)]
      [(_ chain default)
       (unless (identifier? #'chain)
         (raise-syntax-error #f "Expected hash.key0[.key1 ...]
[default]" stx #'chain))
       (let* ([chain-str (symbol->string (syntax->datum #'chain))]
              [ids (for/list ([str (in-list (regexp-
split #rx"\\." chain-str))])
                     (format-id #'chain "~a" str))])
         ; Check that we have at least hash.key
         (unless (and (>= (length ids) 2)
                      (not (eq? (syntax-e (cadr ids)) '||)))
           (raise-syntax-error #f "Expected
hash.key" stx #'chain))
         (with-syntax ([hash-table (car ids)]
                       [keys       (cdr ids)])
           #'(hash-refs hash-table 'keys default)))]))
; See if we catch each of the misuses
> (hash.refs)
eval:96:0: hash.refs: Expected hash.key0[.key1 ...]
[default]
  at: chain
  in: (hash.refs)
> (hash.refs 0)
eval:98:0: hash.refs: Expected hash.key0[.key1 ...]
[default]
  at: 0
  in: (hash.refs 0 #f)
> (hash.refs js)
eval:99:0: hash.refs: Expected hash.key
  at: js
  in: (hash.refs js #f)
```

```
> (hash.refs js.)
eval:100:0: hash.refs: Expected hash.key
  at: js.
  in: (hash.refs js. #f)
```

Not too bad. Of course, the version with error-checking is quite a bit longer. Error-checking code generally tends to obscure the logic, and does here. Fortunately we'll soon see how `syntax-parse` can help mitigate that, in much the same way as contracts in normal Racket or types in Typed Racket.

Maybe we're not convinced that writing `(hash.refs js.a.b.c)` is really clearer than `(hash-refs js '(a b c))`. Maybe we won't actually use this approach. But the Racket macro system makes it a possible choice.

5 Syntax parameters

"Anaphoric if" or "aif" is a popular macro example. Instead of writing:

```
(let ([tmp (big-long-calculation)])
  (if tmp
      (foo tmp)
      #f))
```

You could write:

```
(aif (big-long-calculation)
     (foo it)
     #f)
```

In other words, when the condition is true, an `it` identifier is automatically created and set to the value of the condition. This should be easy:

```
> (define-syntax-rule (aif condition true-expr false-expr)
    (let ([it condition])
      (if it
          true-expr
          false-expr)))
> (aif #t (displayln it) (void))
it: undefined;
 cannot reference an identifier before its definition
  in module: 'program
```

Wait, what? `it` is undefined?

It turns out that all along we have been protected from making a certain kind of mistake in our macros. The mistake is if our new syntax introduces a variable that accidentally conflicts with one in the code surrounding our macro.

The Racket *Reference* section, Transformer Bindings, has a good explanation and example. Basically, syntax has "marks" to preserve lexical scope. This makes your macro behave like a normal function, for lexical scoping.

If a normal function defines a variable named `x`, it won't conflict with a variable named `x` in an outer scope:

```
> (let ([x "outer"])
    (let ([x "inner"])
      (printf "The inner `x' is ~s\n" x))
    (printf "The outer `x' is ~s\n" x))
```

```
The inner `x' is "inner"
The outer `x' is "outer"
```

When our macros also respect lexical scoping, it's easier to write reliable macros that behave predictably.

So that's wonderful default behavior. But sometimes we want to introduce a magic variable on purpose—such as `it` for `aif`.

There's a bad way to do this and a good way.

The bad way is to use `datum->syntax`, which is tricky to use correctly.

See Keeping it Clean with Syntax Parameters (PDF).

The good way is with a syntax parameter, using `define-syntax-parameter` and `syntax-parameterize`. You're probably familiar with regular parameters in Racket:

```
> (define current-foo (make-parameter "some default value"))
> (current-foo)
"some default value"
> (parameterize ([current-foo "I have a new value, for now"])
    (current-foo))
"I have a new value, for now"
> (current-foo)
"some default value"
```

That's a normal parameter. The syntax variation works similarly. The idea is that we'll define `it` to mean an error by default. Only inside of our `aif` will it have a meaningful value:

```
> (require racket/stxparam)
> (define-syntax-parameter it
    (lambda (stx)
      (raise-syntax-error (syntax-e stx) "can only be used inside
aif")))
> (define-syntax-rule (aif condition true-expr false-expr)
    (let ([tmp condition])
      (if tmp
          (syntax-parameterize ([it (make-rename-
transformer #'tmp)])
            true-expr)
          false-expr)))
> (aif 10 (displayln it) (void))
10
> (aif #f (displayln it) (void))
```

Inside the `syntax-parameterize`, `it` acts as an alias for `tmp`. The alias behavior is created by `make-rename-transformer`.

If we try to use `it` outside of an `aif` form, and `it` isn't otherwise defined, we get an error like we want:

```
> (displayln it)
it: can only be used inside aif
```

But we can still define `it` as a normal variable in local definition contexts like:

```
> (let ([it 10])
    it)
10
```

or:

```
> (define (foo)
    (define it 10)
    it)
> (foo)
10
```

For a deeper look, see Keeping it Clean with Syntax Parameters.

6 What's the point of `splicing-let`?

I stared at `racket/splicing` for the longest time. What does it do? Why would I use it? Why is it in the Macros section of the reference?

Step one, ~~cut a hole in the box~~ de-mythologize it. For example, using `splicing-let` like this:

```
> (require racket/splicing)
> (splicing-let ([x 0])
    (define (get-x)
      x))
; get-x is visible out here:
> (get-x)
0
; but x is not:
> x
x: undefined;
 cannot reference an identifier before its definition
  in module: 'program
```

is equivalent to:

```
> (define get-y
    (let ([y 0])
      (lambda ()
        y)))
; get-y is visible out here:
> (get-y)
0
; but y is not:
> y
y: undefined;
 cannot reference an identifier before its definition
  in module: 'program
```

This is the classic Lisp/Scheme/Racket idiom sometimes called "let over lambda". A closure hides `y`, which can only be accessed via `get-y`.

A koan about closures and objects.

So why would we care about the splicing forms? They can be more concise, especially when there are multiple body forms:

```
> (require racket/splicing)
> (splicing-let ([x 0])
    (define (inc)
```

```
  (set! x (+ x 1)))
(define (dec)
  (set! x (- x 1)))
(define (get)
  x))
```

The splicing variation is more convenient than the usual way:

```
> (define-values (inc dec get)
    (let ([x 0])
      (values (lambda ()  ; inc
                (set! x (+ 1 x)))
              (lambda ()  ; dec
                (set! x (- 1 x)))
              (lambda ()  ; get
                x))))
```

When there are many body forms—and we're generating them in a macro—the splicing variations can be much easier.

7 Robust macros: syntax-parse

Functions can be used in error. So can macros.

7.1 Error-handling strategies for functions

With plain old functions, we have several choices how to handle misuse.

1. Don't check at all.

```
> (define (misuse s)
    (string-append s " snazzy suffix"))
; User of the function:
> (misuse 0)
string-append: contract violation
  expected: string?
  given: 0
  argument position: 1st
  other arguments...:
   " snazzy suffix"
; I guess I goofed, but - what is this "string-append" of which
you
; speak??
```

The problem is that the resulting error message will be confusing. Our user thinks they're calling `misuse`, but is getting an error message from `string-append`. In this simple example they could probably guess what's happening, but in most cases they won't.

2. Write some error handling code.

```
> (define (misuse s)
    (unless (string? s)
      (error 'misuse "expected a string, but got ~a" s))
    (string-append s " snazzy suffix"))
; User of the function:
> (misuse 0)
misuse: expected a string, but got 0
; I goofed, and understand why! It's a shame the writer of the
; function had to work so hard to tell me.
```

Unfortunately the error code tends to overwhelm and/or obscure our function definition. Also, the error message is good but not great. Improving it would require even more error code.

3. Use a contract.

```
> (define/contract (misuse s)
    (string? . -> . string?)
    (string-append s " snazzy suffix"))
; User of the function:
> (misuse 0)
misuse: contract violation
  expected: string?
  given: 0
  in: the 1st argument of
      (-> string? string?)
  contract from: (function misuse)
  blaming: program
   (assuming the contract is correct)
  at: eval:131.0
; I goofed, and understand why! I'm happier, and I hear the writer
of
; the function is happier, too.
```

This is the best of both worlds.

The contract is a simple and concise. Even better, it's declarative. We say what we want to happen, not how.

On the other hand the user of our function gets a very detailed error message. Plus, the message is in a standard, familiar format.

4. Use Typed Racket.

```
#lang typed/racket


> (: misuse (String -> String))
> (define (misuse s)
    (string-append s " snazzy suffix"))
> (misuse 0)
eval:3:0: Type Checker: type mismatch
  expected: String
  given: Zero
  in: 0
```

Even better, Typed Racket can catch usage mistakes up-front at compile time.

7.2 Error-handling strategies for macros

For macros, we have similar choices.

1. Ignore the possibility of misuse. This choice is even worse for macros. The default error messages are even less likely to make sense, much less help our user know what to do.

2. Write error-handling code. We saw how much this complicated our macros in our example of §4.3 “Using dot notation for nested hash lookups”. And while we’re still learning how to write macros, we especially don’t want more cognitive load and obfuscation.

3. Use `syntax-parse`. For macros, this is the equivalent of using contracts or types for functions. We can declare that input pattern elements must be certain kinds of things, such as an identifier. Instead of "types", the kinds are referred to as "syntax classes". There are predefined syntax classes, plus we can define our own.

7.3 Using `syntax-parse`

November 1, 2012: So here’s the deal. After writing everything up to this point, I sat down to re-read the documentation for `syntax-parse`. It was...very understandable. I didn’t feel confused.

Why? The documentation has a nice Introduction with many simple examples, followed by an Examples section illustrating many real-world scenarios.

Update: Furthermore, Ben Greenman has created a package whose docs provide an excellent set of even more Syntax Parse Examples.

Furthermore, everything I’d learned up to this point prepared me to appreciate what `syntax-parse` does, and why. The details of how to use it seem pretty straightforward, so far.

This might well be a temporary state of me "not knowing what I don’t know". As I dig in and use it more, maybe I’ll discover something confusing or tricky. If/when I do, I’ll come back here and update this.

But for now I’ll focus on improving the previous parts.

8 References and Acknowledgments

Eli Barzilay's blog post, Writing 'syntax-case' Macros, helped me understand many key details and concepts, and inspired me to use a "bottom-up" approach.

Eli wrote another blog post, Dirty Looking Hygiene, which explains `syntax-parameterize`. I relied heavily on that, mostly just updating it since his post was written before PLT Scheme was renamed to Racket.

Matthew Flatt's Composable and Compilable Macros: You Want it When? (PDF) explains how Racket handles compile time vs. run time.

Chapter 8 of *The Scheme Programming Language* by Kent Dybvig explains `syntax-rules` and `syntax-case`.

Fortifying Macros (PDF) is the paper by Ryan Culpepper and Matthias Felleisen introducing `syntax-parse`.

Shriram Krishnamurthi looked at a very early draft and encouraged me to keep going. Sam Tobin-Hochstadt and Robby Findler also encouraged me. Matthew Flatt showed me how to make a Scribble `interaction` print `syntax` as `"syntax"` rather than as `"#'"`. Jay McCarthy helped me catch some mistakes and confusions. Jon Rafkind provided suggestions. Kieron Hardy reported a font issue and some typos.

Finally, I noticed something strange. After writing much of this, when I returned to some parts of the Racket documentation, I noticed it had improved since I last read it. Of course, it was the same; I'd changed. It's interesting how much of what we already know is projected between the lines. My point is, the Racket documentation is very good. The *Guide* provides helpful examples and tutorials. The *Reference* is very clear and precise.