BEAUTIFUL RACKET / TUTORIALS

Make a language in one hour: stacker

THE EXPANDER

To recap—every language in Racket must provide two things:

① A reader, which converts source code from a string of characters into S-expressions.

② An expander, which determines how these S-expressions correspond to real Racket expressions, which are then evaluated to produce a result.

We made our reader. Now we'll make our expander.

First, why is it called an *expander*? Recall that our reader consisted of a `read-syntax` function that surrounded each line of the source file with `(handle ...)`, in essence "expanding" it. The expander will perform a similar process on the rest of the code, with the help of macros.

INTRODUCING MACROS

It turns out that a lot of things in Racket that look like normal functions are actually copying & rewriting code when the program is compiled (an event we'll call compile time) rather than being invoked when the program is evaluated (an event we'll call run time). For instance, when we invoke `and` within a program:

```
(and cond-a cond-b cond-c)
```

At compile time, `and` gets expanded—that is, rewritten—so it looks like this:

```
(if cond-a
    (if cond-b
        cond-c
        #f)
    #f)
```

This is the code that actually gets evaluated. Let's not be alarmed. These nested `if` conditionals mean the same thing as our original `and`. The expander has simply converted the `and` expression into the equivalent `if` expressions.

Within Racket, `and` belongs to a special and important class of functions called *macros*. Macros have a restricted interface: they take certain code as input (packaged inside a syntax object), and return other code as output (also as a syntax object). Thus, macros are also known more precisely as syntax transformers. But we can think of them as a sophisticated form of search-and-replace—they're like regular expressions designed to work on code fragments, rather than strings.

Strictly, macros are functions, but it's conventional in Racket to call them *macros*, and reserve the term *function* for a procedure that's evaluated at run time. So if we talk about "converting a function into a macro," that's the intended contrast.

Macros are also known, especially within the Racket documentation, as forms—a nice term, because it emphasizes that a macro doesn't evaluate the code it gets as input. Rather, it just implements a template for putting items into certain positions, like filling in the blanks of a form. We could, for instance, use the `and` form like this:

```
(and (error 'boom) (error 'bam) (error 'pow))
```

If `and` were a function, these arguments would be evaluated first, and the program would end with `error: boom` before the arguments were ever passed to `and`.

THANK YOU FOR YOUR COMMENT

But because `and` is a macro, it happily rewrites our code as usual:

```
(if (error 'boom)
    (if (error 'bam)
        (error 'pow)
        #f)
    #f)
```

When this new code runs, we'll still get an error, of course. But the `and` macro was able to complete its work anyhow. This brings us to the three golden rules of macros:

1. At compile time, a macro takes code as input, and converts it to new code that will be evaluated at run time. The input & output code is packaged inside a syntax object.
2. Because compile time happens before run time, all macros operate before any run-time functions.
3. Because compile time happens before run time, a macro can only treat its input code as a literal syntactic entity. It cannot evaluate arguments or expressions within that code, because that information is only available at run time.

Under this definition, our reader's `read-syntax` function wasn't a macro, because it didn't transform certain code into other code—rather, it took two input arguments (a path and an input port) and made code from that (packaged into a syntax object).

But as we'll see, the linchpin of our expander will be a macro.

BEFORE WE ENTER THE HALL OF THE MACRO KING

Racket's macro system is its crown jewel. Refined for nearly 20 years, it's a work of great science and beauty. It's also fundamental to how Racket models the implementation of languages. Therefore, anyone who wants to make languages with Racket has to be willing to learn a few things about macros.

That said, though the macro system is always rewarding, it's not always easy. For most programmers, these are new & unfamiliar concepts. Don't panic. These ideas aren't esoteric or complicated. They just introduce a new way of thinking about program code. Most of us haven't thought about code this way because we haven't worked with languages with an elegant macro system like Racket's.

WHAT DOES THE EXPANDER DO?

If the reader was responsible for the *form* of the code, we could say the expander is responsible for its *meaning*. The expander's job is to prepare the code so Racket can evaluate it.

Prepare it how? The code can be evaluated only if every name used in the code has a connection to an actual value or function. In Racket, a "name used in the code" is known as an identifier, and "a connection to an actual value or function" is known as a binding. So we'll say that the expander prepares the code for evaluation by ensuring that every identifier has a binding. Once an identifier has a binding, it becomes a variable.

We already came across the ideas of identifiers and bindings when we tested our `stacker` reader without an expander. Recall that we got the error `handle: unbound identifier in module`. Now we know what it means. The code from the reader referred to a `handle` identifier, e.g.:

```
(handle 4)
```

This expression means "call `handle` with the argument `4`." But the expander hadn't given `handle` a binding. Thus we got the "unbound identifier" error.

Within the expander, we have three basic techniques for adding bindings to code:

1. We can *define macros* that convert certain code into other code at compile time. (We've now seen two: the `and` macro and the `define` macro.)
2. We can *define functions* that are invoked at run time.
3. We can *import bindings* from existing Racket modules, which can include both macros and functions.

We'll use all three techniques in our `stacker` expander.

DESIGNING OUR EXPANDER

THANK YOU FOR YOUR COMMENT

As we did with our reader, let's pencil out what our expander needs to do.

In our case, the code we're getting from the reader will look like this:

```
(handle)
(handle 4)
(handle 8)
(handle +)
(handle 3)
(handle *)
```

From this sample, we can list the tasks for our expander:

① From the prototype code above, we see we need to provide bindings for three identifiers: `handle`, which determines what to do with each argument; `+`, a stack operator; and `*`, another stack operator.

We also need numbers. But we get those for free—in Racket, numbers can't be identifiers. They automatically evaluate to their numeric value. Likewise the parentheses—those are just delimiters, and Racket already knows what to do with them.

② Consistent with the design of `stacker`, we also have to implement a stack, with an interface for storing, reading, and doing operations on arguments, that can be used by `handle`.

③ Finally, our expander needs to provide the special `#%module-begin` macro to get everything started. We'll discuss that in the next section.

Once we have these elements in place, the `stacker` language will work.

PROGRAMMING OUR EXPANDER: OUTPUT

Recall the state of our code when our expander starts. Our reader has just finished its work, meaning `read-syntax` has returned code (inside a syntax object) that describes a module. For example, if the `stacker` reader started with source code like this:

```
4
8
+
3
*
```

It would return the following module as a syntax object:

```
(module stacker-mod "stacker.rkt"
  (handle)
  (handle 4)
  (handle 8)
  (handle +)
  (handle 3)
  (handle *))
```

This becomes the starting input for the expander.

We saw that Racket starts the reader for a language by invoking a function called `read-syntax`. Similarly, Racket starts the expander for a language by invoking a macro called, by convention, `#%module-begin`. Therefore, every expander needs to provide a `#%module-begin` macro.

Racket converts the module above into an invocation of the `#%module-begin` macro by passing it the code inside the module:

```
(#%module-begin
  (handle)
  (handle 4)
  (handle 8)
  (handle +)
  (handle 3)
  (handle *))
```

Like any macro, `#%module-begin` will take this code as input. And like any macro, `#%module-begin` will rewrite it as necessary and return new code, packaged as a syntax object. This new code will replace the code above, and expansion & evaluation will continue from there.

Because every Racket language provides a `#%module-begin`, the most common way to implement the `#%module-begin` in a new language is:

① Handle any language-specific processing of the code.

② Pass the result to another `#%module-begin` for the rest of the heavy lifting. (Caution: all the `#%module-begin` macros have the same name. Therefore, some care is needed to keep them straight.)

THANK YOU FOR YOUR COMMENT

Open `"stacker.rkt"` and update it like so:

stacker.rkt

```
#lang br/quicklang

(define (read-syntax path port)
  (define src-lines (port->lines port))
  (define src-datums (format-datums '(handle ~a) src-lines))
  (define module-datum `(module stacker-mod "stacker.rkt"
                          ,@src-datums))
  (datum->syntax #f module-datum))
(provide read-syntax)

(define-macro (stacker-module-begin HANDLE-EXPR ...)
  #'(#%module-begin
     HANDLE-EXPR ...))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

At the top, we see good old `read-syntax`, just as we left it.

Below that, our definition of `#%module-begin`. Let's step through each part.

```
(define-macro (stacker-module-begin HANDLE-EXPR ...)
```

When we made our reader, we saw that an ordinary function is defined with a list of input arguments. But because a macro gets a chunk of code, we typically define a macro with a syntax pattern instead. A syntax pattern is like a regular expression: it breaks down the input into pieces so they can be manipulated & rearranged.

Our ordinary `define` doesn't support syntax patterns in the first line, so we use `define-macro`, which does. This first line says we're defining a macro called `stacker-module-begin`. The `HANDLE-EXPR ...` is the syntax pattern, which will match each line of the code passed to the macro. (We'll defer the details till later, though the curious can detour through syntax patterns.)

Then we have the return value of our macro:

```
#'(#%module-begin
   HANDLE-EXPR ...)
```

Syntax objects in Racket are so common that we have a few ways to make them. Recall that in `read-syntax` we used `(datum->syntax #f '(module-datum))` to make a syntax object from `'(module-datum)`.

This time, we're using some new notation—the prefix `#'`—to make code into a syntax object. It's similar to the `'` prefix we've already used that makes code into a datum. But the `#'` prefix not only creates the datum, but also captures its lexical context, and attaches that to the new syntax object. Lexical context is fancy jargon for "a list of available variables." In practice, what it means is that this syntax object made with `#'` will be able to access all the variables that are defined at this point in the code.

One of these variables is the `HANDLE-EXPR ...` syntax pattern. Those lines of code will just be merged into the new syntax object.

Another variable is `#%module-begin`. Recall that our plan is to take our input code and pass it to the next `#%module-begin` in the chain. When we go into DrRacket and put the cursor over the `#%module-begin`, we'll see a purple arrow and a popup message that says `imported from "br/quicklang"`. We don't need to explicitly import this `#%module-begin`—it's automatically available because we're using `br/quicklang`, and every language provides its own `#%module-begin`. (For that matter, we could import one from a different language, but for `stacker`, there's no need.)

Finally we have to make `stacker-module-begin` available outside `stacker.rkt`:

```
(provide (rename-out [stacker-module-begin #%module-begin]))
```

We use `provide`, but add the `rename-out` qualifier so that `stacker-module-begin` is available outside this source file with the correct `#%module-begin` name.

Why didn't we just name our macro `#%module-begin` at the start? Suppose we had written our macro like this:

```
(define-macro (#%module-begin HANDLE-EXPR ...)
  #'(#%module-begin
     HANDLE-EXPR ...))
(provide #%module-begin)
```

THANK YOU FOR YOUR COMMENT

The problem here is that we'd be creating a name conflict between two `#%module-begin` macros: the new one we're defining, and the one from `br/quicklang` that we're hoping to rely on. This `#%module-begin` will just call itself, creating an infinite loop. Thus, we have to give our new macro a non-conflicting name, and change it in the `provide` expression.

TESTING OUR EXPANDER

Let's see if our simple `#%module-begin` works. Here's what `"stacker.rkt"` should look like:

stacker.rkt

```
#lang br/quicklang

(define (read-syntax path port)
  (define src-lines (port->lines port))
  (define src-datums (format-datums '(handle ~a) src-lines))
  (define module-datum `(module stacker-mod "stacker.rkt"
                          ,@src-datums))
  (datum->syntax #f module-datum))
(provide read-syntax)

(define-macro (stacker-module-begin HANDLE-EXPR ...)
  #'(#%module-begin
     HANDLE-EXPR ...))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

Now run `"stacker-test.rkt"` (which is the same as ever):

```
#lang reader "stacker.rkt"
4
8
+
3
*
```

We get an error: `handle: unbound identifier in module`. That's the same error we got when we tried to run our program without an expander. But we haven't gotten around to defining `handle` yet, so we shouldn't be surprised that the error persists.

Can we at least check that our macro is working? Sure. Let's reuse the trick we learned when we needed to test the reader the first time: adding a `'` prefix to the output so it gets converted back into literal code. This time, we'll add the `'` prefix to the `HANDLE-EXPR` inside our macro definition like so:

```
(define-macro (stacker-module-begin HANDLE-EXPR ...)
  #'(#%module-begin
     'HANDLE-EXPR ...))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

Now when we run `"stacker-test.rkt"`, we get:

```
'(handle)
'(handle 4)
'(handle 8)
'(handle +)
'(handle 3)
'(handle *)
```

What we're seeing is our source code, converted into `(handle ···)` expressions by `read-syntax` in our reader, and then passed through the new `#%module-begin` macro in our expander, and then returned to the output window of DrRacket. That's good news—we've verified that we can make a round-trip from our source file, through our reader, then through our expander, and back. We're another step closer to having a working language.

Let's remove the `'` prefix we just added to the front of `HANDLE-EXPR`, and move on.

PROGRAMMING OUR EXPANDER: BINDINGS

When we designed our expander, we set out three tasks:

① Provide the special `#%module-begin` macro.

② Implement a stack, with an interface for storing, reading, and doing operations on arguments, that can be used by `handle`.

③ Provide bindings for three identifiers: `handle`, which determines what to do with each argument; `+`, a stack operator; and `*`, another stack operator.

We made our `#%module-begin`. So now we need to implement a stack and the bindings for our identifiers.

THANK YOU FOR YOUR COMMENT

First, the stack, which we'll implement using Racket list operations. Add the following code to "stacker.rkt":

```
(define stack empty)

(define (pop-stack!)
  (define item (first stack))
  (set! stack (rest stack))
  item)

(define (push-stack! item)
  (set! stack (cons item stack)))
```

Our stack is a list that starts out empty.

The pop-stack! function removes the top element from the stack and returns it. We do this in three steps. We assign the top element of the stack to a temporary item variable using (first stack). Then we set! our stack equal to the elements after the first using (rest stack). (By convention, Racket functions that update the value of a variable are named with a ! at the end.)

The push-stack! function adds an element to the stack. We do this with cons. cons is one of the oldest Lisp functions, getting its name from its role *constructing* a new item in memory. (For more about cons, see Pairs.) Every time we cons an item onto the list, we create a new list that's one bigger. After we add the item, we set! our stack equal to this new, larger stack.

Now let's add our long-awaited handle function. Let's remember how handle is invoked in a stacker program:

```
(handle)
(handle 4)
(handle 8)
(handle +)
(handle 3)
(handle *)
```

We observe that it can have zero or one argument. If it has an argument, it's either a number or a stack operator (either + or *). So let's add the following handle function:

```
(define (handle [arg #f])
  (cond
    [(number? arg) (push-stack! arg)]
    [(or (equal? + arg) (equal? * arg))
     (define op-result (arg (pop-stack!) (pop-stack!)))
     (push-stack! op-result)]))
(provide handle)
```

Then we'll step through each line:

```
(define (handle [arg #f])
```

We define our handle function to take one argument, called arg. By declaring arg in brackets, we make it an optional argument. So if handle is called with an argument, arg takes that value; if not, arg is set to #f.

```
(cond
    [(number? arg) (push-stack! arg)]
```

This cond introduces a conditional expression with two branches. The condition on the left side of a branch is tested. If the condition is true, the code on the right side of the branch is evaluated. If not, we evaluate the next branch. (Details in Booleans and conditionals.) The first branch tests if arg is a number?. If so, we put it onto the stack with push-stack!.

```
[(or (equal? + arg) (equal? * arg))
     (define op-result (arg (pop-stack!) (pop-stack!)))
     (push-stack! op-result)]
```

In the second branch, we test if arg is equal? to one of our stack operators. If so, we retrieve the first two elements from the stack with two calls to pop-stack!. We apply arg to these values by putting it in the function position of a new expression with the stack values as arguments. We store the result in op-result. We then push op-result onto the stack.

What if handle is called with no arguments, and arg is #f? We were always planning to ignore that case. So we don't need to add a branch to our cond expression to deal with it. We'll just let it fall through. Likewise, any other input that doesn't meet one of our conditions will be ignored.

```
(provide handle)
```

THANK YOU FOR YOUR COMMENT

Finally, we make `handle` available outside our source file with the usual `provide`.

We also need to provide bindings for `+` and `*`. This is easy, because our `br/quicklang` language already defines these functions. We just need to borrow these bindings for `stacker`. We can do that by adding one more `provide` to our source file:

```
(provide + *)
```

Our `"stacker.rkt"` should now look like this:

stacker.rkt

```
#lang br/quicklang

(define (read-syntax path port)
  (define args (port->lines port))
  (define handle-datums (format-datums '(handle ~a) args))
  (define module-datum `(module stacker-mod "stacker.rkt"
                          ,@handle-datums))
  (datum->syntax #f module-datum))
(provide read-syntax)

(define-macro (stacker-module-begin HANDLE-EXPR ...)
  #'(#%module-begin
     HANDLE-EXPR ...
     (display (first stack))))
(provide (rename-out [stacker-module-begin #%module-begin]))

(define stack empty)

(define (pop-stack!)
  (define item (first stack))
  (set! stack (rest stack))
  item)

(define (push-stack! item)
  (set! stack (cons item stack)))

(define (handle [arg #f])
  (cond
    [(number? arg) (push-stack! arg)]
    [(or (equal? * arg) (equal? + arg))
     (define op-result (arg (pop-stack!) (pop-stack!)))
     (push-stack! op-result)]))
(provide handle)

(provide + *)
```

TESTING OUR EXPANDER WITH BINDINGS

We're getting close. Let's run our `"stacker-test.rkt"` file, which still looks like this:

stacker-test.rkt

```
#lang reader "stacker.rkt"
4
8
+
3
*
```

The good news—no errors.
The bad news—no program result, either.

This is a simple fix. The result of our program is whatever value is sitting on top of the stack at the end. So we add one line to our `stacker-module-begin`, to `display` the `first` element of our `stack` after all the `HANDLE-EXPR` lines have been evaluated:

```
(define-macro (stacker-module-begin HANDLE-EXPR ...)
  #'(#%module-begin
     HANDLE-EXPR ...
     (display (first stack))))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

Run `"stacker-test.rkt"` one more time, and we'll get:

```
36
```

That's it—we've made our first programming language.