BEAUTIFUL RACKET / CHAPTER 2

Make your first programming language — in one hour

> *This is an excerpt from a book I'm currently writing about making programming languages with Racket. I welcome comments & suggestions. Of course, I'm publishing the book with Pollen.*
> *— Matthew Butterick*

Let's make `stacker`, a programming language that implements a stack-based calculator.

```
#lang stacker
push 4
push 8
+
push 3
*
```

```
36
```

The language relies on a stack of arguments, which starts out empty. The `push` command puts a new argument on the top of the stack. An operator — either `+` or `*` — takes the top two arguments from the stack, and replaces them with the result of applying the operator (meaning `first + second` or `first * second`). So this sample program means `3 * (8 + 4)`, which is `36`.

Even though `stacker` is simple, it will let us tour the whole language-making workflow in Racket. Every language project will follow the same basic steps.

Later, we'll see how `stacker` can become the foundation of a whole virtual-machine language.

PREREQUISITES

Install Racket and the `beautiful-racket` package as described in Setup. Though you can work through this tutorial with any code editor, I recommend — and will assume you're using — Racket's graphical editor, DrRacket.

Some general programming experience is assumed. No specific experience with Racket, Scheme, or Lisp is assumed. Don't worry if some of the terminology or concepts go over your head — all the important topics will be reiterated later.

WHAT IS A PROGRAMMING LANGUAGE?

On a technical level, a programming language just another program. It takes certain input, then evaluates that input to produce a result.

On a design level, a programming language is a specification for what kind of input is acceptable, how that input should be interpreted, and what kind of results should be generated. In that regard, what makes a programming language different from an ordinary program is that it offers the widest horizon of abstraction.

On a cultural level, a programming language is a way to express our evolving ideas and discoveries about programming generally — logic, performance, ergonomics, and so on. Though a programming language is a way of writing a program, it's also a way of *thinking* about a program.

Because if languages were just about writing programs, we could've stopped with C. (And some have.) But what would be the fun in that? Programs are interesting specifically because they're malleable. And the more we expect out of programs, the more vital it is to explore new ways of making programs.

Including new languages.

By the way, does a programming language need to be large & general-purpose, like C? Not at all. A **domain-specific language** (or **DSL**) is a "little language" — optimized to handle a specific task, and nothing else. Sometimes DSLs are designed to be used inside larger languages — e.g,. regular expressions, SQL. Sometimes they stand alone — e.g., CSS/HTML, PostScript, R, TeX. Many Unix-descended tools also qualify as DSLs — e.g., awk, bash, lex/yacc, make.

WHY WOULD I MAKE A PROGRAMMING LANGUAGE?

If you think programming should fill your brain and soul with feelings of power, creativity, curiosity, and fun — then you'll probably like making programming languages.

If not, you can move along. (No hard feelings.)

But in terms of practical benefits:

- **Enlarge the solution space**. You can do things at the language level that you can't do within a program. Being able to create a language raises the ceiling of possibilities.
- **Shave fewer yaks**. It doesn't matter which language you like best or use most — sometimes you encounter a problem that doesn't fit well with the language idiom. Whereas a new language can be tailored to fit exactly.
- **Certain programs naturally describe languages**. Languages are good for what we might call wide-funnel problems, where there's a large universe of possible inputs, but a relatively small scope of outputs. (How to tell: does your program need to parse input from arbitrarily complex files, and turn them into a simpler representation?)
- **Better glue**. A little language can fill holes in a larger toolchain. For instance, Python doesn't have a preprocessor like C does, but you could make one as a DSL.
- **Get more done**. Whether you work alone or in a team, a language can streamline common tasks at a higher level than a library or framework. (You don't even need to tell your manager.)
- **You'll know something others don't**. Moreover, knowing how to make a programming langugage will teach you ideas that you can fold into your usual programming work.

Of course, a programming language isn't the right solution for every problem. But when creating a language is easy and inexpensive — as the rest of this tutorial will show you — it becomes a realistic option for many more problems.

HOW ARE LANGUAGES IMPLEMENTED IN RACKET?

Racket is a descendant of Lisp and Scheme that provides a high-level interface for making new languages.

The basic idea is to take a program written in the new language and convert its surface syntax and semantics to a standard Racket program. Then Racket runs this program normally.

In that sense, every language in Racket is just an indirect way of writing other Racket programs (aka a source-to-source compiler). This approach makes it possible for a new language to immediately benefit from everything in Racket's toolchain: the libraries, the cross-platform compatibility, the DrRacket IDE, and so on.

Because every language is itself just a program, a language can do anything a program can. So you can make languages that behave like traditional programming languages. Or languages that behave in unconventional ways.

And why not? Making a programming language used to be a years-long commitment. Now you can make one in minutes. (To be fair, this first one will be a little slower. But you'll get the hang of it.)

THE STRUCTURE OF A LANGUAGE

Every language in Racket must provide two functions:

1. A **reader**, which converts the source file from a string of characters into Racket-style parenthesized forms. (We'll see how that works in a minute.)
2. An **expander**, which determines how the reader forms correspond to real Racket expressions, which are then evaluated to produce a result. (We'll see why it's called an *expander* later on.)

When you make a language, you'll always have to specify a reader and an expander. Sometimes you'll write them from scratch. Sometimes you'll import them from elsewhere.

But for `stacker`, we'll write both the reader and expander. In most projects, the expander is more involved, so it often makes sense to start there and work backward to the reader. But for this project, they're equally simple. Thus, we'll start with the reader, so we get a better sense of the program flow, start to finish.

BOOTSTRAPPING YOUR LANGUAGE

Every Racket source file begins with a `#lang` line. If you've explored Racket you might have seen variants like:

```
#lang racket
#lang racket/base
#lang scribble/doc
#lang datalog
```

We just learned that the first step in interpreting a language is to process the source with a reader. The `#lang` line's job is to tell Racket where to find a reader for the source file. The reader, in turn, will tell Racket where to find the expander.

Often, the reader and expander for a language are stored in different source files. But they don't have to be. For this project, we'll just use one file, called `stacker-lang.rkt`. (There's no magic in the filename. You can call it whatever you want.)

To use the abbreviated `#lang stacker` notation, we'll have to do some extra housekeeping that we'll cover at the end of the tutorial.

For now, we'll use the special bootstrap `#lang reader ...` notation, like so:

```
#lang reader "stacker-lang.rkt"
```

This is the same as the notation above, but allows us to use the path to the source file that has our reader.

RACKET BASICS

We need to write some Racket code. So let's learn some Racket.

- The basic building block of Racket is the **expression**. An expression can be a single value like `2` or `"blue"`, a variable like `edge`, a list of values like `(list 2 "blue" edge)`, or a function call like `(* 21 2)`.
- Function calls go between parentheses. Unlike most languages, a function name takes the first position inside the parentheses, followed by its arguments. Thus you write `(* inside 4)`, not `(inside * 4)`. These parenthesized forms are also known as *S-expressions*.
- Every expression is evaluated to produce a value. A variable evaluates to whatever value it holds (so after we say `(define inside 2)`, `inside` would evaluate to `2`). A function call evaluates to its return value (so `(+ 2 2)` would evaluate to `4`).

- Expressions can be nested to any depth. Thus, `(* inside 4)` could be written `(* inside (+ 2 2))` or `(* inside (+ (+ 1 1) (+ 1 1)))`.
- True and false are spelled `#t` and `#f`. In conditional forms like `if`, the only value that evaluates to false is `#f`. Everything else counts as true (even `0`).
- Linebreaks and indentation are used for readability. They don't change the meaning of the code.
- Square brackets are sometimes used for readability. They mean the same thing as parentheses.

  Beyond that, Racket has the usual numbers, strings, and functions that you know from other languages. We'll handle those as we encounter them.

SETTING UP OUR SOURCE FILES

Open DrRacket. Start a new file called `stacker-lang.rkt`. We'll write this file using a special tutorial language called `br`. (`br` comes from the `beautiful-racket` package we installed. It's Racket, but with some extra conveniences.) For testing purposes, let's add a second line, so your whole file looks like this:

stacker-lang.rkt

```
#lang br
42
```

Click Run in DrRacket (or learn the key shortcut, because you'll be using it a lot). In the interactions window, you'll see the result of running the file:

```
42
```

In the same directory, create a second file that we'll use to test our new language as we go:

stacker-test.rkt

```
#lang reader "stacker-lang.rkt"
```

Run this file too. You should get an awful-looking error like *`read-syntax: cannot find reader for '#lang reader "stacker-lang.rkt"`*. If so, that's the right result: we're telling `#lang` to get a reader out of `"stacker-lang.rkt"`, but we haven't made one yet.

So let's do that.

DESIGNING OUR READER

Recall the purpose of the reader:

- The **reader** converts the source file from a string of characters into Racket-style parenthesized forms.

Before plunging into the code, we should visualize what our reader needs to accomplish. Our `stacker` language will ultimately look like this:

```
#lang stacker
push 4
push 8
+
push 3
*
```

These, however, are not parenthesized forms. How can we get there? We notice that our language has one instruction on each line. Thus, the simplest idea — never a bad place to start — would be to just wrap parentheses around each line:

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

Not bad. These now look like Racket expressions. If we code up a function called `push`, these expressions would evaluate correctly.

The problem comes with the operators, which become expressions that look like `(+)` and `(*)`. That parenthesizing means these functions would be evaluated without any arguments. That would be wrong — they're supposed to take their input arguments from the top of the stack. (You're welcome to run `(+)` or `(*)` in a separate DrRacket window to see what you get.)

We need to make it possible to manage the interaction of our stack and our operators. So let's imagine that we have an intermediate function — call it `dispatch` — that will inspect each instruction and decide what to do. Then we can wrap each instruction like so:

```
(dispatch push 4)
(dispatch push 8)
(dispatch +)
(dispatch push 3)
(dispatch *)
```

What have we accomplished? Most importantly, our operators won't be treated as function calls, but rather as arguments to another function. `(dispatch +)` means "call `dispatch` with `+` as the argument." In that case, `dispatch` can get the arguments off the stack and apply the operation. Likewise for `(dispatch push 4)` — `dispatch` can just treat it like it were `(push 4)`. Therefore, as long as we can write a `dispatch` function for our expander that works this way — trust me, we can — then this reader transformation will suffice.

Curious characters might wonder if we could make a reader that treated the two types of instructions differently:

```
(push 4)
(push 8)
(dispatch +)
(push 3)
(dispatch *)
```

Yes, we could. But for this tutorial, we're going to use the simplest possible reader, and wrap everything with `dispatch`. We'll reserve all the other work for the expander. In your own languages, you can choose how to allocate responsibilities between the reader and expander.

To summarize — our reader will convert code that looks like this:

```
push 4
push 8
+
push 3
*
```

Into this:

```
(dispatch push 4)
(dispatch push 8)
(dispatch +)
(dispatch push 3)
(dispatch *)
```

Now that we know what our reader needs to do, we can implement it.

PROGRAMMING OUR READER: OUTPUT

By convention, Racket always expects the main reader function to be called `read-syntax`. Racket will call this function with two arguments: the path to the source file, and a port for reading from the file. But for now, let's set these aside and focus on the output of the function.

`read-syntax` has one job: to return a *syntax object* describing a *module*. The module, in turn, will invoke the expander for our language, triggering the full expansion and evaluation of the input program.

Let's see how this works by looking at some code.

`stacker-lang.rkt`

```
#lang br
(define (read-syntax src-path src-port)
  #'(module lucy br
      42))
(provide read-syntax)
```

To create a function, we use `define`. `(define (read-syntax src-path src-port) ...)` defines a function called `read-syntax` that accepts two arguments. These are positional arguments, so we can name them whatever we like. To make `read-syntax` function available outside the file, we use `provide`. (In Racket, all definitions within a source file are private by default.)

In the body of our function we see our module syntax. (Racket has no `return` statement — a function simply returns the last expression in its body.) In Racket, a **module** is the basic organizational unit for code. Like everything else in Racket, a module is an expression. A module expression has this form:

```
(module name-of-module language-that-will-provide-the-expander
   expressions-to-evaluate ...)
```

So our module expression:

```
(module lucy br
    42)
```

Means "a module named `lucy`, using the expander from the `br` language, that evaluates the expression `42`."

Then we need to turn this into a syntax object. A **syntax object** is a way of storing a reference to the literal code of an expression, so it can be evaluated later. We can turn any expression into a syntax object by wrapping it in the `syntax` converter:

```
(syntax (module untitled br
          42))
```

Syntax objects are so common in Racket that there's a shorthand notation for making them — just append the `#'` prefix (pronounced *hash-quote*) to the expression. The idiomatic way of writing the above syntax object would be:

```
#'(module untitled br
    42)
```

"This syntax object will be evaluated later — but when?" When we invoke the language. Save `"stacker-lang.rkt"`. Run `"stacker-test.rkt"`. This time, instead of an error, you should get a different result:

```
42
```

What's happening here? When we run `"stacker-test.rkt"`, we're invoking our language, which means calling our reader function, `read-syntax`. The contents of the `"stacker-test.rkt"` source file are replaced with the result of this function, which is a syntax object describing a module. Once that syntax has been moved into the new location, it's evaluated, producing `42`.

Let's persuade ourselves that nothing spooky has happened. We can accomplish the same transformation by hand. We replace the `#lang` line in `"stacker-test.rkt"` with the module expression (this time without the `#'` syntax prefix, because we want the code to be evaluated), and run it:

stacker-test.rkt

```
(module untitled br
    42)
```

Once again we get `42`.

This proved that we can make a round-trip from a source file to a reader and back. Our reader works, except that it's not, you know, *reading* anything. So let's fix that.

PROGRAMMING OUR READER: INPUT

We'll upgrade our reader to do three new tasks:

1. Read in the lines of the source file.
2. Convert these lines to `(dispatch ...)` form.
3. Put these new forms into the module we're returning as a syntax object.

To do this, let's swap out the code in `"stacker-lang.rkt"`:

stacker-lang.rkt

```
#lang br
(define (read-syntax src-path src-port)
  (define src-strs (port->lines src-port))
  (define (make-datum str) (format-datum '(dispatch ~a) str))
  (define src-exprs (map make-datum src-strs))
  (inject-syntax ([#'(src-expr ...) src-exprs])
    #'(module stacker-reader br
        src-expr ...)))
(provide read-syntax)
```

We'll step through each line to see what it does.

```
(define src-strs (port->lines src-port))
```

Recall that Racket passes two arguments to our function: the path to the source file, which we've named `src-path`, and an input port pointing at this file, which we've named `src-port`. A **port** is a generic interface for retrieving data from a file or other source. Using `port->lines`, we'll retrieve all the lines from the file as strings, and store them in `src-strs`. (In this project, we won't need `src-path` for anything.)

Next, we'll convert these strings into datums. A **datum** is a string that's been parsed as a Racket form, but not yet evaluated. For instance, these are strings:

```
"(list 1 2 3)" "#t" "hello" "(+ 1 2)"
```

And these are the corresponding datums (I know the plural of *datum* ought to

be *data*, but allow me some grammatical license, for clarity):

```
'(list 1 2 3)  #t  'hello  '(+ 1 2)
```

To create a datum from an existing parenthesized expression, we simply wrap it with the `quote` converter:

```
(quote (module untitled br
         42))
```

Though as with syntax objects, there is ane equivalent shorthand notation — in this case, we prefix the expression with `'` (also pronounced *quote*):

```
'(module untitled br
   42)
```

If a datum sounds like a syntax object, that's no coincidence — a syntax object is just a datum with some extra information attached. The `#'` prefix notation we used for a syntax object now connects logically: we can think of the `#` as representing the extra info that gets attached to the datum:

```
#'(module untitled br
    42)
```

Back to our code:

```
(define (make-datum str) (format-datum '(dispatch ~a) str))
(define src-datums (map make-datum src-strs))
```

We want the strings we got from our source file to behave as expressions, so we'll convert them to datums. Because we're converting a list of strings, we'll use `map` to apply a conversion function, `make-datum`, to every element in `src-strs`. This will create a new list of `src-exprs`. In turn, `make-datum` relies on `format-datum` to create a datum with the right form — `'(dispatch ~a)` (the value of `str` gets interpolated into the `~a` slot).

Finally, the twistiest bit of code — but don't panic:

```
(inject-syntax ([#'(src-expr ...) src-exprs])
  #'(module stacker-reader br
      src-expr ...))
```

Recall that the output of our function is a syntax object describing a module. We need to update this syntax object to incorporate the datums we just made.

To do this, we'll use `inject-syntax` to make new syntax available within the module syntax. In the first line, we match our datums to a **syntax pattern**, `#'(src-expr ...)`, which is just a way of labeling elements so they can be rearranged as pieces of syntax. Here, the syntax pattern unpacks the elements of `src-exprs` so we can refer to them individually.

This is easier to understand if you look at how the syntax object has changed. `src-expr` refers to the first item in the list. After that, the `...` notation means "do the same to every other item in the list." So if `src-exprs` has three items, say `(list '(dispatch 42) '(dispatch "Hello world") '(dispatch #t))`, then the resulting syntax object will look like this:

```
#'(module stacker-reader br
    (dispatch 42)
    (dispatch "Hello world")
    (dispatch #t))
```

Notice that the quote prefixes on our datums disappear. That's because a) `inject-syntax` automatically upgrades our datums to syntax objects and then b) the syntax-object prefix `#'` on the front of the module expression applies to everything in it.

TESTING OUR READER

At this point, it would be nice to see if our updated reader works the way we hope. Let's make a source file with these sample values and see if we do, in fact, get this code back. Save `"stacker-lang.rkt"`. Update `"stacker-test.rkt"` as follows:

stacker-lang.rkt

```
#lang reader "stacker-lang.rkt"
42
"Hello world"
#t
```

Run it. What happens? You should get an error that looks like *dispatch: unbound identifier in module*. In time, you will think of this as a good-news error — it means that our code is trying to call the `dispatch` function (which is what we wanted) but can't, because we haven't written it yet (which shouldn't surprise us).

In the meantime, how can we check the reader function? For debugging purposes, we can take a sneaky shortcut: add a `'` prefix to the `src-expr`, like so:

```
(inject-syntax ([#'(src-expr ...) src-exprs])
  #'(module stacker-reader br
      'src-expr ...))
```

What will happen now? Before we run our test file again, let's reason through the code. There is a trap afoot for the unwary. Here are the two possibilities.

Option A: "Well, adding the `'` prefix turns an expression into a datum, which is the literal code `src-expr`. And if there are three lines of input, the `...` means that datum will get repeated three times in the syntax object, resulting in this—"

```
#'(module stacker-reader br
    'src-expr
    'src-expr
    'src-expr)
```

Option B: "No, everything works the same way as before, except that every line now has a `'` in front of it, like so—"

```
#'(module stacker-reader br
    '(dispatch 42)
    '(dispatch "Hello world")
    '(dispatch #t))
```

As we might hope, the answer is B. But why? Don't forget that the point of using a syntax object is to *delay* evaluation of the code inside. While we're using `inject-syntax`, the only parts of our syntax object that can change are the pieces referenced by our syntax pattern — namely, the variable `src-expr` and the `...` shorthand. The rest, including the `'` prefixing, won't be evaluated until after the function exits.

It may help to remember that the `'` prefix is equivalent to wrapping in `quote` like so:

```
(inject-syntax ([#'(src-expr ...) src-exprs])
  #'(module stacker-reader br
      (quote src-expr) ...))
```

This makes it a little more plain that the `quote` will be repeated for every expression in `(src-expr ...)`.

But *then* what will happen? Run `stacker-test.rkt` and see for yourself:

```
'(dispatch 42)
'(dispatch "Hello world")
'(dispatch #t)
```

By using the `'` prefix within the syntax object, we convert our code back into datums, which print as values in DrRacket. Now we can ask: did the reader do what we hoped? Looks like it did.

To finish our test, let's change our source file to use the input from the very first example:

`stacker-test.rkt`

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

Run it again and we get:

```
'(dispatch push 4)
'(dispatch push 8)
'(dispatch +)
'(dispatch push 3)
'(dispatch *)
```

Let's compare that to the reader behavior we were looking for:

```
(dispatch push 4)
(dispatch push 8)
(dispatch +)
(dispatch push 3)
(dispatch *)
```

Except for the quotes (which are just there temporarily for debugging), that's exactly right. We can move on — after we've made one last change to prepare the reader to cooperate with the expander (which we'll building in the next section). To do this, we change the module expression so that instead of

invoking `br` as the expander, it invokes `"stacker-lang.rkt"`. So the finished reader looks like this:

stacker-lang.rkt

```
#lang br
(define (read-syntax src-path src-port)
  (define src-strs (port->lines src-port))
  (define (make-datum str) (format-datum '(dispatch ~a) str))
  (define src-exprs (map make-datum src-strs))
  (inject-syntax ([#'(src-expr ...) src-exprs])
    #'(module stacker-mod "stacker-lang.rkt"
        src-expr ...)))
(provide read-syntax)
```

THE EXPANDER

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

1. The **reader** converts the source file from a string of characters into Racket-style parenthesized forms.
2. The **expander** determines how the reader forms 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 took source code and, instead of returning a value, produced a syntax object that added `(dispatch ...)` to each line of the source code, in essence "expanding" it.

It turns out that a lot of things in Racket that look like run-time functions are instead copying & rearranging code at compile time. These are a special class of functions called **macros**. Macros have a restricted interface: they take one syntax object as input, and return another syntax object as output. Thus, macros are also known as **syntax transformers**.

(Though macros are functions, it's conventional in Racket to only refer to them as "macros," and reserve the term "function" for something that's called at run time. So if I say something like "convert a function into a macro," that's the intended contrast.)

Under this definition, our `read-syntax` function wasn't quite a macro. Though it did return a syntax object, it accepted two input arguments: a path and an input port.

But as we'll see, our expander is going to start with a macro, which will in turn bring in the other functions we need to run our source file.

DESIGNING OUR EXPANDER

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

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 evaluate the code prepared by the reader. 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**. In short, the expander has to ensure that every identifier in the code has a binding. (A **variable** in Racket = an identifier + a binding.)

We already came across this terminology when we ran `"stacker-test.rkt"` without an expander (run it again if you like). We got the error `dispatch: unbound identifier in module`. Now we see what it means. The code from the reader refers to a `dispatch` identifier, e.g.:

```
(dispatch push 4)
```

This expression means "call the function `dispatch` with the arguments `push` and `4`." But the expander hasn't yet given `dispatch` a binding. Hence the error when the program runs: `dispatch` is undefined.

This error isn't endemic to our language. You'll trigger the same error in Racket whenever you use a name that hasn't yet been defined, for instance:

```
#lang racket
foo
```

```
#lang scribble/doc
@bar[]
```

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

```
(dispatch push 4)
(dispatch push 8)
(dispatch +)
(dispatch push 3)
(dispatch *)
```

From this sample, we can list the identifers that will need bindings:

1. `dispatch`, which is a function.
2. `push`, which is a stack command.
3. `+` and `*`, which are stack operators.

We also need numbers. But we get those for free — in Racket, numbers can't be identifiers. They automatically evaluate to their numeric value.

Our expander design is complete. If our expander provides bindings for these identifiers — and one special macro, that we'll discuss in the next section — then the `stacker` language will work.

PROGRAMMING OUR EXPANDER: OUTPUT

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

Racket has a small number of macros with special status. You can recognize them by the `#%` prefix. `#%module-begin` is special because when a module is evaluated, all the code inside that module gets sent to `#%module-begin`, which can do anything it pleases with it, before returning a syntax object, after which the program expansion & evaluation will continue.

Because every Racket language provides its own `#%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 heavy lifting.

Of course, because all the `#%module-begins` have the same name, some care needs to be taken.

But let's see the careless approach first. Put this code in `"stacker-`

lang.rkt":

```
(define #'(#%module-begin reader-line ...)
  #'(#%module-begin
     (define studio (* 6 9))
     (displayln studio)))
(provide #%module-begin)
```

As before, we'll use `define` and `provide` to set up our function and make it available to other files. We'll talk about `#'(#%module-begin reader-line ...)` in the next section.

For now, let's focus on the output of the macro. As with `read-syntax`, it's another syntax object (meaning, it begins with `#'`). This preliminary `#%module-begin` isn't going to do anything with the inbound code. It's just going to create a variable called `studio` and print its value. Then it will pass this syntax to the `#%module-begin` from the `br` language (which is always available within `"stacker-lang.rkt"` because the file is written in `#lang br`).

Make sure `"stacker-test.rkt"` still looks like this, and run it:

stacker-test.rkt

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

What happened? Nothing yet? OK, one Mississippi, two Mississippi ... how about now? Nothing at all? Why is this taking so long?

Unfortunately the run will never finish, because our expander has an awful bug in it. Within DrRacket, select *Force the program to quit* or type `ctrl+K`. (Consider memorizing this key combination, because it's not the last time you'll need to kill a program in your language-debugging career.)

Astute observers probably already noticed the bug: when we defined our macro with the name `#%module-begin`, the `#%module-begin` inside the macro, instead of referring to the `#%module-begin` in `#lang br`, now referred to itself. (A problem of variable naming more generally known as *shadowing*.) So our `#%module-begin` called itself recursively ... forever.

We can cure the problem by giving our macro an arbitrary name when we `define` it, and then using `rename-out` to change its name as it passes through `provide`, like so:

```
(define #'(stacker-module-begin reader-line ...)
  #'(#%module-begin
     (define studio (* 6 9))
     (displayln studio)))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

After you replace this code, run `"stacker-test.rkt"` again, and you'll get the expected result:

```
54
```

"Hold on. Why does renaming the macro work? Doesn't the output syntax still refer to `#%module-begin`, which is the final name of the macro?" No, but only because Racket macros are **hygienic**. Macros copy code to other places, packaged in syntax objects. That code contains identifiers. So a question arises: how should we determine the binding — aka, the meaning — of a copied identifier? Should we look at the destination for the copy (where the macro is being invoked)? Or at the source for the copy (where the macro is defined)?

Hygienic macros follow the second rule — a macro copies code, but the identifiers used in the macro keep their bindings from the place where the macro was defined. (Hence the "hygiene" metaphor — the macro keeps its identifiers "cleanly" separated from those at the destination.)

To see hygiene in action, go into DrRacket and hover your cursor over the `#%module-begin` inside `stacker-module-begin`. Now that we've changed the name of the macro, you'll see a message pop up: `binding #%module-begin imported from br`. What hygiene guarantees is that whenever this macro is used, the `#%module-begin` identifier will *always* refer to the `#%module-begin` from `#lang br`.

For now, we won't delve into the details of how Racket keeps all this straight. And yes, it is possible to override hygiene when needed. What's important to notice is that a macro is not merely a find-and-replace mechanism — it floats within a bubble of bindings.

Moreover, hygiene is good policy: it guarantees that wherever our macro is

used, the copied code will always mean the same thing. Here's an example of why that might be a wise idea. Curious characters are invited to paste this program into DrRacket and hover over the two references to `*` to understand why the `*-macro` does not, in fact, start global thermonuclear war (unfortunately, `(* 6 9)` does):

```
#lang br
(module the-macro br
  (provide *-macro)
  (define #'(*-macro x y) #'(* x y)))

(require 'the-macro) ; `require` imports bindings from a module
(define (* x y) 'global-thermonuclear-war-initiated)
(*-macro 6 9)
(* 6 9)
```

We've now proved 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.

PROGRAMMING OUR EXPANDER: INPUT

Recall that a macro takes one syntax object as input and returns another. Consistent with that logic, we can use the `#'` syntax prefix within `define` to easily convert a function into a macro. Just add the `#'` prefix to both the input and output expressions, like so.

```
#lang br
(define (func x y z) (list x y z))
(define #'(mac x y z) #'(list x y z))
(list (func 4 5 6) (mac 4 5 6)) ; '((4 5 6) (4 5 6))
```

One wrinkle is that the input expression for a macro is actually a *syntax pattern*, like we saw in our `read-syntax` function:

```
(inject-syntax ([#'(src-expr ...) src-exprs])
  #'(module stacker-reader br
      src-expr ...))
```

That means we can use the `...` syntax ellipsis when needed:

```
#lang br
(define (func x y z) (list x y z))
(define #'(mac arg ...) #'(list arg ...))
```

```
(list (func 4 5 6) (mac 4 5 6)) ; '((4 5 6) (4 5 6))
```

There's often more than one valid way to write a macro. For instance, these three macros produce the same result (shouldn't be hard to infer why):

```
#lang br
(define (func x y z) (list x y z))
(define #'(mac x y z) #'(list x y z))
(define #'(mac-2 x y z) #'(func x y z))
(define #'mac-3 #'func)
(list (func 4 5 6) (mac 4 5 6) (mac-2 4 5 6) (mac-3 4 5 6))
```

For our `#%module-begin` macro, we'll invoke `define` with the syntax pattern `#'(#%module-begin reader-line ...)`. In so doing, our macro will match the first line of the incoming code to `reader-line`, and the remaining lines to the `...` shorthand. For now, we'll just pass the `reader-lines` through to the next `#%module-begin` in the chain, leaving us with the simplest possible macro:

```
(define #'(stacker-module-begin reader-line ...)
  #'(#%module-begin
     reader-line ...))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

TESTING OUR EXPANDER I/O

Just as we did in our reader, we can add the quote prefix `'` inside our macro to print out the literal code:

```
(define #'(stacker-module-begin reader-line ...)
  #'(#%module-begin
     'reader-line ...))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

When we run our `"stacker-test.rkt"`:

stacker-test.rkt

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

We should get the same result as we got when testing our reader:

```
'(dispatch push 4)
'(dispatch push 8)
'(dispatch +)
'(dispatch push 3)
'(dispatch *)
```

You might also get an error: *Interactions disabled: "stacker-lang.rkt" does not support a REPL*. Don't panic — we'll fix that shortly.

For now, let's make sure our `"stacker-lang.rkt"` is restored to its normal state. We're up to a whopping 12 lines of code:

stacker-lang.rkt

```
#lang br
(define (read-syntax src-path src-port)
  (define src-strs (port->lines src-port))
  (define (make-datum str) (format-datum '(dispatch ~a) str))
  (define src-exprs (map make-datum src-strs))
  (inject-syntax ([#'(src-expr ...) src-exprs])
    #'(module stacker-mod "stacker-lang.rkt"
        src-expr ...)))
(provide read-syntax)

(define #'(stacker-module-begin reader-line ...)
  #'(#%module-begin
     reader-line ...))
(provide (rename-out [stacker-module-begin #%module-begin]))
```

Run `"stacker-test.rkt"` again and we should see a different error: *dispatch: unbound identifier in module*. We'll fix that next.

PROGRAMMING OUR EXPANDER: BINDINGS

To complete our expander, we need to provide bindings for the identifiers we enumerated in our expander design:

1. `dispatch`, which is a function.
2. `push`, which is a stack command.
3. `+` and `*`, which are stack operators.

We'll start with `push`, which takes one numerical argument and pushes onto the top of a stack. Add the following code to `"stacker-lang.rkt"`:

```
(define stack empty)
(define (push num) (set! stack (cons num stack)))
```

Our `stack` starts out as the `empty` list. `push` uses the `cons` function to put `num` onto the front of the list, and `set!`s the stack to the new version. `cons` is one of the oldest Lisp functions, getting its name from its role *constructing* a new item in memory. Every time we `cons` an item onto the list, we create a new list that's one bigger. `set!` is self-explanatory, except maybe the `!` suffix — that's a Racket convention to denote functions that update the value of a variable.

Now let's add our long-awaited `dispatch` function:

```
(define (dispatch arg-1 [arg-2 #f])
  (cond
    [(number? arg-2) (push arg-2)]
    [else
     (define op arg-1)
     (define op-result (op (first stack) (second stack)))
     (set! stack (cons op-result (drop stack 2)))]))
```

There are a few new Racket functions here, so we'll walk through each line.

```
(define (dispatch arg-1 [arg-2 #f])
```

We `define` our `dispatch` function to take two arguments. How do we know we need two arguments? We just look at the code that's coming into our expander.

```
(dispatch push 4)
(dispatch push 8)
(dispatch +)
(dispatch push 3)
(dispatch *)
```

We see there are two calling patterns for `dispatch`. Either we'll get two arguments, where the first argument is `push` and the second argument is a number. Or we'll get one argument, which will be a stack operator (`+` or `*`). The second argument is **optional**, so we put it in square brackets and assign a default value of `#f`.

```
(define (dispatch arg-1 [arg-2 #f])
```

```
  (cond
    [(number? arg-2) (push arg-2)]
    [else ...]))
```

`cond` is Racket's conditional expression. Each set of square brackets introduces a conditional test, followed by the code to evaluate if the test is true. An `else` branch is invoked when none of the tests are true.

Our first branch tests whether we got a `number?` for `arg-2`. If so, we know it's a `push` instruction, so we `(push arg-2)`.

```
[else
 (define op arg-1)
 (define op-result (op (first stack) (second stack)))
 (set! stack (cons op-result (drop stack 2)))]
```

If we don't have a `push` instruction, we must have a stack operator in the first argument. So we `(define op arg-1)`. We need to send our operator the `first` and `second` elements from the stack, so we get `(op (first stack) (second stack))`. Then we put the resulting value back onto the stack using `set!`, as we did in `push`. Notice, however, that we `drop` those two elements from the stack before we put the result onto the stack — the idea is that they get consumed by the operator.

That's it for bindings.

"Wait — what about `+` and `*`?" We don't need to do anything, because they already have bindings. Recall that macros, including `stacker-module-begin`, are hygienic. This means that identifiers inside any macro-created code take their meaning from where the macro was defined. Because `stacker-module-begin` has access to all the bindings provided by `#lang br`, these bindings travel with the code the macro creates. `+` and `*` are already defined by `#lang br` in the usual way, which suffices. (Neither `push` nor `dispatch` are defined by `#lang br`, so we had to make those.)

That said, we remain free to `define` new bindings for `+` and `*` within our language, even in ways that are counterinituitive:

```
(define + -)
(define * /)
```

Were we to do so, these new bindings would supersede the ones from `#lang`

`br` within `"stacker-lang.rkt"`, including within the code created by a macro like `stacker-module-begin`.

"Don't we have to `provide` all these language bindings, as we did for `stacker-module-begin`?" Actually, no. `provide` makes a binding available outside of a source file. But remember that a macro brings its own bindings along for the ride (= that's hygiene). So when our `stacker-module-begin` macro creates references to `dispatch`, `push`, `+`, and `*` elsewhere, they automatically refer to the definitions that are local to the macro.

If that still seems spooky or weird, think about it this way. The American embassy in Paris occupies a very nice building on the Place de la Concorde. Certainly, the embassy is physically within the boundaries of France. But when you step inside the embassy, what country are you in? You're no longer in France. You're in the United States, and US law applies. In that way, a macro behaves like a sovereign state. Though the code it creates travels to other locations, the identifiers in that code retain the meaning they have in the macro homeland.

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

stacker-lang.rkt

```
#lang br
(define (read-syntax src-path src-port)
  (define src-strs (port->lines src-port))
  (define (make-datum str) (format-datum '(dispatch ~a) str))
  (define src-exprs (map make-datum src-strs))
  (inject-syntax ([#'(src-expr ...) src-exprs])
                 #'(module stacker-mod "stacker-lang.rkt"
                     src-expr ...)))
(provide read-syntax)

(define #'(stacker-module-begin reader-line ...)
  #'(#%module-begin
     reader-line ...))
(provide (rename-out [stacker-module-begin #%module-begin]))

(define stack empty)

(define (push num) (set! stack (cons num stack)))

(define (dispatch arg-1 [arg-2 #f])
  (cond
    [(number? arg-2) (push arg-2)]
    [else
     (define op arg-1)
     (define op-result (op (first stack) (second stack)))
```

```
      (set! stack (cons op-result (drop stack 2)))]))
```

TESTING OUR EXPANDER WITH BINDINGS

Are we there yet? Just about. Let's run our `"stacker-test.rkt"` file, which still looks like this:

stacker-test.rkt

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

When we do, we get an error: *`Interactions disabled: "stacker-lang.rkt" does not support a REPL`*. The **REPL** means *read–evaluate–print loop*, and refers to the interactive command line that appears adjacent to our code in DrRacket. To clear this error, we need to add one housekeeping line to our `"stacker-lang.rkt"`, the explanation of which can wait till later:

```
(provide #%top-interaction)
```

Run `"stacker-test.rkt"` again. The good news — no errors. The bad news — no program result, either.

Fortunately this is also 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 the stack after all the `reader-line`s have been evaluated:

```
(define #'(stacker-module-begin reader-line ...)
#'(#%module-begin
   reader-line ...
   (display (first stack))))
```

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

```
36
```

Congratulations — you just made a programming language.