Submodules in Racket

You Want it *When*, Again?

Matthew Flatt

PLT and University of Utah
mflatt@cs.utah.edu

Abstract

In an extensible programming language, programmers write code that must run at different times—in particular, at compile time versus run time. The module system of the Racket programming language enables a programmer to reason about programs in the face of such extensibility, because the distinction between run-time and compile-time phases is built into the language model. *Submodules* extend Racket's module system to make the phase-separation facet of the language extensible. That is, submodules give programmers the capability to define new phases, such as "test time" or "documentation time," with the same reasoning and code-management benefits as the built-in distinction between run time and compile time.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Macros, modules, language tower

1. Introduction

Racket's module system is similar to the module systems of other languages. Modules are static, they reside in a global namespace with hierarchical names, and they statically declare their dependencies on other modules. These properties of the module system simplify compilation and linking tasks; for example, the `raco make` tool can traverse and compile the dependencies of a program as needed, and `raco exe` can combine all needed modules into a single executable.

A distinguishing feature of Racket's module system is the way that it interacts with macro-based "languages." Each module explicitly declares its language, so that different modules in the same program can have different syntaxes or different semantics for a given syntactic form. Such languages are implemented by macros via arbitrary Racket code that runs at compile time, and the module-and-macro system ensures that run-time and compile-time evaluation are kept separate (Flatt 2002). The separation of run time from compile time enables compilation, analysis, and reasoning about programs in general, as well as limiting evaluation at each phase to only the code that is needed for that phase.

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2373-4/13/10. . . $15.00.
http://dx.doi.org/10.1145/2517208.2517211

The original design for modules in Racket allows module declarations only at the top level. Our new extension to Racket's module system enables modules to contain nested module declarations, which are called *submodules*. In most module systems that support nesting, lexically nested modules merely define a local namespace, and they are instantiated along with their enclosing modules. Racket submodules, in contrast, have a lifetime that can be independent of the enclosing module. The code of a submodule need not even be loaded when the enclosing module is used, or vice versa—unless the submodule imports from the enclosing module, or vice versa, in which case the usual phase-sensitive module loading and instantiation rules apply.

Submodules solve a number of practical problems for Racket programmers. They provide a natural way to express a "main" routine that is used only when the module is run as a program and ignored when the module is used as a library. Submodules provide a place for testing internal functions without making the functions public and without causing the tests to run on all uses of the module. Submodules enable abstraction over sets of modules, which is not possible when modules and only modules exist at the top level. Submodules also provide a communication channel for static semantic information about a module, such as whether the module is implemented in Typed Racket (Tobin-Hochstadt and Felleisen 2008) and the types of its exports.

More generally, submodules give a programmer the ability to define new phases along the same lines as the built-in run-time and compile-time phases. We can think of tests, for example, as "test-time" code, as opposed to run-time code. Unlike the distinction between the compile and run phases, the test phase subsumes the run phase, but not vice versa. As another example, when documentation is written as part of the library that it documents, then the code to produce the documentation is document-time code, which is independent of run-time code. In much the same way that run-time code and compile-time code can coexist within a module and benefit from a shared lexical scope (so that the macro's expansion can conveniently refer to bindings in the same module), document-time code can coexist with run-time code and benefit from a shared scope (so that the documentation can conveniently refer to bindings in the module, where the references are rendered as links to the bindings' documentation).

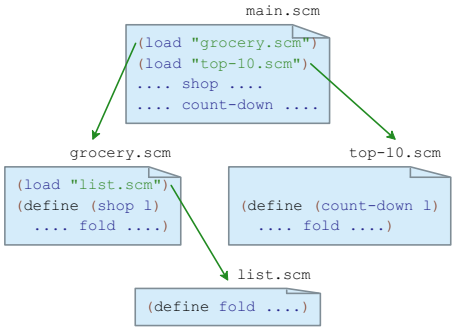
In this paper, we describe Racket's module system, its handling of different phases, and its support for submodules. We show how submodules are used for tasks such as testing and documentation. Finally, we provide a formal model that specifies the interaction of macro expansion, module declarations, compilation, and evaluation.

2. Modules and Phases

Before introducing submodules, we begin with a recap of Racket's design for modules and phases (Flatt 2002). In particular, we moti-

vate Racket's module system in terms of compilation, but we mean "compilation" in a broad sense—as a proxy for any task that requires understanding what a program means without actually running the program. In a macro-extensible language, any such compilation involves running code for macro expansion. At the same time, when running code can have a side effect, running the code only at the right time is particularly important.

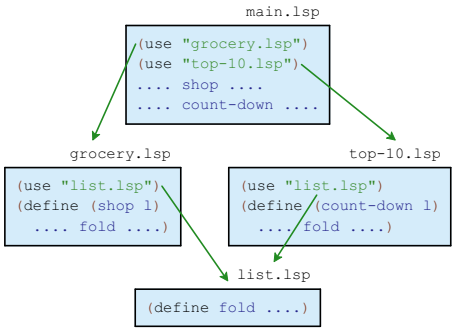
In a traditional Lisp or Scheme setting, a program is constructed by loading code into a `read-eval-print` loop (REPL). For example, if a program is implemented across four files, then it is linked together by having files `load` other files:[1]



In this example, note that the `"top-10.scm"` file uses `fold` without explicitly `load`ing `"list.scm"`. The program works, anyway, because the files are effectively flattened into a linear sequence as they are loaded, and `"list.scm"` gets loaded via `"grocery.scm"`, which is in turn loaded before `"top-10.scm"`.

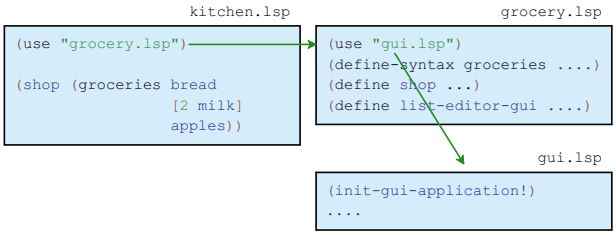
Relying on load-order side effects for program structuring is clearly a bad idea. Reversing the initial loads of `"top-10.scm"` and `"grocery.scm"` will break the program, even though a programmer may be tempted to think of the files as modules that can be imported in either order. Indeed, the lack of an explicit dependency for `"top-10.scm"` on `"list.scm"` is likely to be a mistake that will be discovered unfortunately late. As the number of implementors and "module" files grows, the problem becomes acute.

A step in the right direction is to introduce a concept of "packages" and constrain access to a variable to those in packages that are explicitly imported with, say, a `use` declaration:[2]



This hypothetical `use` declaration plays two roles: it makes the target package's bindings available in the current package, and it ensures that the target package is loaded before the current package. In other words, it helps avoid the mistakes possible with `load` by enforcing a connection between the lexical structure of the program and it dynamic evaluation; the `use` form leverages lexical scope.

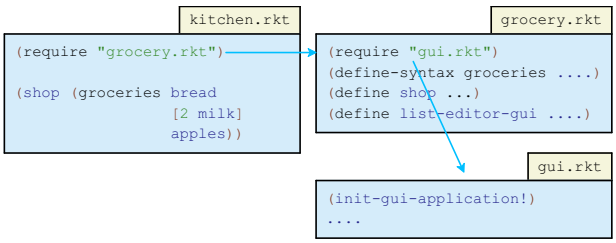
Unfortunately, in a macro-extensible language, simply declaring dependencies on bindings does not cover the full dependency story. Suppose that a `"kitchen.lsp"` package depends on a `"grocery.lsp"` package to supply both a function `shop` and a macro `groceries` (to simplify the construction of grocery lists), while `"grocery.lsp"` depends on `"gui.lsp"` to implement a graphical list manager:



On the one hand, if `use` means only that `"grocery.lsp"` must be loaded before `"kitchen.lsp"` is run, then the implementation of the `groceries` macro will not be available at compile time for `"kitchen.lsp"`, which is when its macros must be expanded. On the other hand, if `use` means that a package is loaded at both run and compile times, then compiling `"kitchen.lsp"` means that `"gui.lsp"` will be loaded to start a graphical interface even at compile time, but the graphical interface should start only at run time.

The traditional solution to this problem is to annotate `use` declarations with an `eval-when` declaration to say when the corresponding package should be loaded. Such annotations, however, are a return to `load`-style scripting of dependencies, which are difficult to get right when many packages are involved. Furthermore, compile-time code must be split into separate packages from run-time code, so that the packages can be loaded at different times.

Racket modules with `require` look essentially the same as our hypothetical packages with `use`:

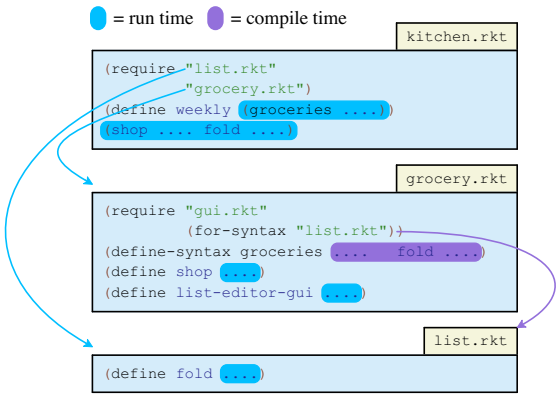


The meaning of `require`, however, is to *run the compile-time portions of the imported modules at compile time*, and to *run the run-time portion of the imported modules at run time*. Thus, the compile-time implementation of `groceries` is available to expand the body of `"kitchen.rkt"`, while `shop`, `list-editor-gui`, and `init-gui-application!` are deferred until run time.

The `require` form triggers the right code at the right time by leveraging lexical scope to determine which parts of a module are for compile time and which parts are for run time. To a first approximation, macro definitions in a module are part of the module's compile time, while function definitions and top-level expressions are part of the module's run time. The following variant of our example illustrates this difference, where blue regions highlight run-time code, and purple highlights compile-time code:

[1] We use a `".scm"` suffix in the figure because many pre-R^{6}RS Scheme implementations often worked as illustrated.

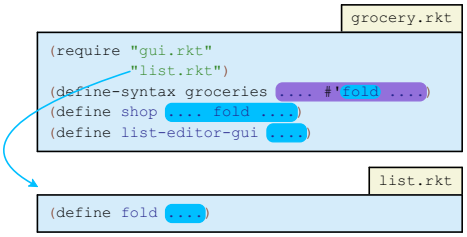
[2] We use a `".lsp"` suffix in the figure because the Common Lisp package system works roughly as illustrated.



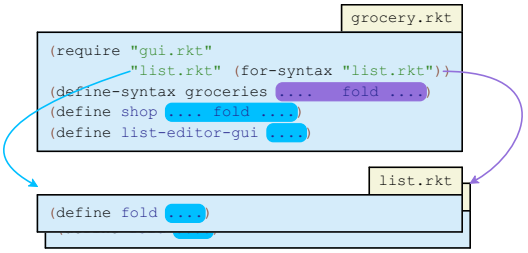
In `"kitchen.rkt"`, the run-time regions include references to `groceries` and `fold`, and therefore they are imported with a plain `require`. Whether `fold` is a macro, function, or variable, all that matters is that `fold` is referenced from a run-time position. In contrast, `"grocery.rkt"` uses `fold` in a compile-time position, so it must use `require` with `for-syntax` to import `fold` from `"list.rkt"`. If `"grocery.rkt"` used `require` without `for-syntax` to import `"list.rkt"`, then `fold` in `"grocery.rkt"` would be unbound for compile-time code, and its use as shown would trigger a syntax error.

The difference between `require` with `for-syntax` and `eval-when` with `use` is subtle but crucial: with `eval-when`, a programmer attempts to say when code should run to make identifiers available; with `require`, a programmer says in which phase an identifier should be bound, leaving the questions of loading and running up to the language. That is, with `require`, programmers reason about scope, instead of reasoning about side-effecting loads.

If the `groceries` macro in `"grocery.rkt"` expands to a run-time use of `fold` instead of using `fold` at compile time, then `"grocery.rkt"` should import `"list.rkt"` normally, instead of `for-syntax`. Put another way, a syntax-quoting `#'` in a macro embeds a run-time region within compile-time code, and a `#'`-quoted reference to `fold` in a macro is the same as a direct reference in a run-time position:



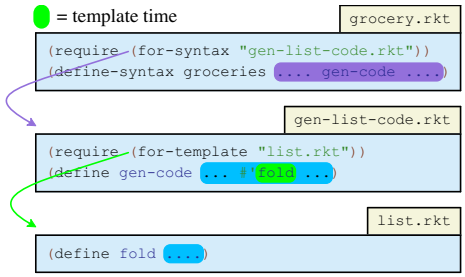
Some libraries, such as a list-processing library like `"list.rkt"`, may even be useful at both compile time and run time. In that case, the module can be `required` both normally and `for-syntax`:



When a module is used in multiple phases, then it is instantiated separately in each phase. Furthermore, to ensure that all-at-once compilation is consistent with separate compilation of modules, a module that is used at compile time is instantiated separately for each module to be compiled. Separate instantiations avoid cross-phase interference and help tame state enough to make it useful for communication among macros (Culpepper et al. 2007; Flatt 2002).

Using lexical scope to determine and manage evaluation phases fits naturally with hygienic macros (Kohlbecker et al. 1986), which also obey lexical scope. For example, if the `groceries` macro expands to a use of a private function that is defined within `"grocery.rkt"`, then hygiene ensures that the expanded reference is bound by the definition in `"grocery.rkt"` and not a definition in the client module where the `groceries` macro is used. Furthermore, the fact that `groceries` was bound for use in a certain position implies `"grocery.rkt"` was imported for that position's phase, which in turn implies that the private definition from `"grocery.rkt"` will be ready by the time that the position is evaluated.

Preserving lexical scope across module boundaries ultimately leads to the need for an import form that is like `require` with `for-syntax` but that works in the other direction. For example, imagine that the core implementation of the `groceries` macro is both sophisticated and general enough that it should be put in its own module, `"gen-list-code.rkt"` that is `required` with `for-syntax` for use by the `groceries` macro:



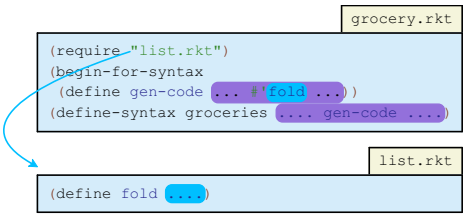
As `gen-code` constructs an expression for the expansion of `groceries`, it uses a syntax-quoted `fold`. The `gen-code` implementation exists at the run-time phase relative to its enclosing module, while the reference to `fold` is generated for sometime further in the future—when the result of `gen-code` is run as part of a generated program. Using `require` with `for-template` enables a reference to a binding that exists in that future (as opposed to `for-syntax`, which enables a reference to a binding that exists in the past, relative to run time), and so `"gen-list-code.rkt"` `requires` the `"list.rkt"` module with `for-template`.

Following the dependencies from `"grocery.rkt"` through `"list.rkt"`, the `for-syntax` and `for-template` phase shifts effectively cancel each other. A `for-syntax` import implies a phase shift of +1 (toward the past), and a `for-template` import implies a phase shift of -1 (toward the future), so that the combination gives +1 + -1 = 0.[3] Consequently, a phase-0 (i.e., run time) use of `"grocery.rkt"` triggers a phase-0 use of `"list.rkt"`, which means that a use of `groceries` and its expansion to `fold` are consistent. Reasoning about such phase shifts as a dynamic process is difficult, but reasoning locally about a reference to a future `fold` within `"gen-list-code.rkt"` is easy.

Using a macro inside of a compile-time context, such as the right-hand side of a `define-syntax` form, means that the macro runs at compile time relative to compile time—that is, at phase 2. In `require`, `for-syntax` and `for-template` specifications can be nested to import bindings into arbitrary phase levels. Furthermore, `begin-for-syntax` allows compile-time defini-

[3] Our phase numbers are negated compared to phase numbers in the staged-computation literature. The negation reflects our emphasis on macros instead of run-time code generation.

tions to be written within a module whose macros need the definitions. For example, if `gen-code` is needed only by macros within `"grocery.rkt"`, it might be better implemented within the `"grocery.rkt"` module instead of in a separate module:



This variant of `"grocery.rkt"` effectively inlines the earlier `"gen-list-code.rkt"` module, shifting the inlined code with `begin-for-syntax` instead of `require` with `for-syntax`.

In the same way that a module can import bindings at multiple phases, it can also *export* bindings at multiple phases. The most prominent example of multi-phase exports is the `racket` module, which is implicitly imported in the above module pictures. The `racket` module re-exports all of the bindings of a more primitive `racket/base` module, and it re-exports the bindings at both phase 0 and phase 1. That's why a module that imports `racket` can include a definition of the form

```
(begin-for-syntax (define gen-code ...))
```

The `racket` module exports `define` at phase 1, so the `define` above is bound for the phase where it is used. If the module's initial import is `racket/base`, instead of `racket`, then the macro definition above is a syntax error unless `define` is explicitly imported `for-syntax`.

Naturally, `begin-for-syntax` forms can be nested to implement a function that is needed for a macro that is used on the right-hand side of a macro implementation, and so on, provided that `begin-for-syntax` itself is imported into each phase where it is used. Despite many possibilities for phases and nesting, the lexical and phase-sensitive constraints on each variable ensure that evaluation times are properly kept in sync.

3. Testing Time And Documentation Time

We can deploy the principles of lexical scope and phase separation to distinguish phases other than just run time and compile time. For example, suppose that we implement a function that uses the Racket `current-seconds` function to implement a `current-hours` function, and suppose that the implementation uses a private `seconds->hours` function:

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))
```

The internal `seconds->hours` function should be tested:

```
(require rackunit)
(check-equal 0 (seconds->hours 0))
(check-equal 1 (seconds->hours 3600))
(check-equal 42 (seconds->hours 151200))
```

Where should we put these tests? If we put them in a separate module, then `seconds->hours` must be exported, but we wanted to keep that function private. If we add the tests to the end of `"hours.rkt"`, then the private function `seconds->hours` is available, but the tests will run every time that the `"hours.rkt"` module is used, and the dependency on `rackunit` means that every program that uses `"hours.rkt"` must carry along the testing framework.

The Racket `module+` form offers a solution that lets a programmer include tests with the code, which makes the most sense in terms of scoping, but puts the code in a separate *submodule* that effectively implements a testing phase (highlighted orange):

hours.rkt

```
(provide current-hours)

(define (current-hours)
  (seconds->hours (current-seconds)))

(define (seconds->hours s)
  (quotient s (* 60 60)))

(module+ test
  (require rackunit)
  (check-equal 0 (seconds->hours 0))
  (check-equal 1 (seconds->hours 3600))
  (check-equal 42 (seconds->hours 151200)))
```

When `"hours.rkt"` is imported into another module with `require`, then the `test` submodule is ignored; the code for the `test` module is not even loaded in that case, assuming that the file has been compiled ahead of time. If the `"hours.rkt"` module is run with the `raco test` command-line tool or run in DrRacket, then the `test` module is also run.[4]

The testing example uses a submodule relatively explicitly. As an example of a new phase that looks more like compile time, consider the problem of documenting library exports at the site of their implementations. Common Lisp supports documentation in the form of *docstrings*, which are string literals that start the body of a function:

```
(define (current-seconds)
  "reports the time in seconds since the Epoch"
  ....)
(define (current-days)
  "reports the time in days since the Epoch"
  ....)
```

Docstrings are often accessed at run time, but they are required to be literal strings so that they can be recognized syntactically. With the restriction to literal docstrings, a tool can build a documentation index without running a program. Constraining docstrings to literal strings, however, prevents abstraction:

```
(define (docs-for-current what)
  (format "reports the time in ~a since the Epoch"
          what))

(define (current-seconds)
  (docs-for-current "seconds")
  ....)

(define (current-days)
  (docs-for-current "days")
  ....)
```

The `docs-for-current` abstraction doesn't work if docstrings are restricted to literal strings. In Racket, we can address the problem by developing a macro-based extension of the language: a `begin-for-doc` form for documentation-time code along with a keyword syntax (in a macro replacement of `define`) to identify a part of a function definition as documentation:

[4] Sometimes you do want to run tests when a particular library is used. For example, this paper depends on the implementation of the model that is presented in section 5, and the paper explicitly imports the implementation's `test` submodule so that the model is tested whenever the paper is rendered (Klein et al. 2012).

```
(begin-for-doc
  (define (docs-for-current what)
    (format "reports the time in ~a since the Epoch"
            what)))

(define (current-seconds)
  #:doc (docs-for-current "seconds")
  ....)
```

With this approach, documentation need not be constrained to plain text. Figure 1 provides a more complete example that uses Scribble (Flatt et al. 2009) syntax for writing documentation, and where the `begin-for-doc`, `for-doc`, and `define` forms are implemented in `"doc-define.rkt"`. The `require...for-doc` import binds Scribble typesetting forms such as `code` for use in writing documentation, where `code` in turn uses the same scope information as used at run-time to hyperlink the documentation's mention of `current-seconds` to the documentation of `current-seconds`. Similarly, the contract expressions after `#:contract` are mainly run-time code, but they are also incorporated into the documentation, where `->` is hyperlinked to documentation on the contract-construction form and `exact-integer?` is hyperlinked to documentation of the number-testing predicate.

The example in figure 1 does not use submodules explicitly, but the `for-doc` imports, `begin-for-doc` forms, and documentation-time `#:doc` expressions are all macro-expanded into a `srcdoc` submodule (as we sketch later in section 4). The submodule is then available for use by documentation tools. The module that provides `current-hours`, etc., does not itself depend on the Scribble libraries or use the code for `current-docs` when the module is loaded.

4. Submodules

Our pictorial representation of modules so far omits the `#lang` line that starts an actual Racket module. For example, the module

represents the file `"clock.rkt"` whose content is

```
#lang racket
"tick"
```

The initial `#lang racket` line, in turn, is a shorthand for using the parenthesized `module` form:

```
(module clock racket
  "tick")
```

The `racket` here indicates a module to supply the initial bindings for the `clock` module body. The `clock` module does not use any of those bindings (except for the implicit literal-expression form), but it could use any `racket` function or syntactic form—and only those syntactic forms, until it uses `require` to import more. Running the `clock` module prints the result of the expression in its body, so it prints `"tick"`.

To a first approximation, a *submodule* is simply a nested `module` form.

```
(module clock racket
  "tick"
  (module tock racket
    "tock"))
```

Running the `clock` module still prints just `"tick"`. Similarly, evaluating `(require "clock.rkt")` in the Racket REPL prints `"tick"`. In those cases, the `tock` submodule is declared, but it is not instantiated and run. Evaluating `(require (submod "clock.rkt" tock))` in the Racket REPL prints `"tock"` and does not print `"tick"`. Submodule nesting implies a kind of nesting of module names, but it does not imply any run-time relationship between a submodule and its enclosing module.

```
#lang at-exp racket/base
(require "doc-define.rkt"
         racket/contract/base
         (for-doc racket/base
                  scribble/manual))

(define (seconds->hours secs)
  #:contract (-> exact-integer? exact-integer?)
  #:doc @{Takes @code{secs}, a number of seconds
          since the Epoch, and converts it to a
          number of days since the Epoch.

          For example, compose with with the
          @code{current-seconds} function to get
          @code{current-hours}.}
  (quotient secs (* 60 60)))

(begin-for-doc
 (define (current-docs what)
   @list{Returns @what since the epoch.}))

(define (current-seconds)
  #:contract (-> exact-integer?)
  #:doc @{@current-docs["seconds"]}
  (inexact->exact
   (floor (/ (current-inexact-milliseconds) 1000))))

(define (current-hours)
  #:contract (-> exact-integer?)
  #:doc @{@current-docs["hours"]}
  (seconds->hours (current-seconds)))
```

Figure 1: Example program with documentation time

A module can explicitly run one of its submodules using `require`, the same as it would trigger any other module. A module can reference one of its submodules using the relative form `(submod "." ....)`:

```
(module clock racket
  (module tock racket
    "tock")
  (require (submod "." tock))
  "tick")
```

Running this module prints both `"tock"` and `"tick"`, since the module explicitly `require`s its submodule and therefore creates an instantiation relationship.[5]

A submodule declared with `module` cannot import from its enclosing module. The `module*` form is the same as `module` for declaring submodules, but it allows the submodule to import from its enclosing module and not vice versa. (Module dependencies in Racket must be acyclic.) Thus, with the `clock` variant

```
(module clock racket
  "tick"
  (module* tock racket
    (require (submod ".."))
    "tock"))
```

evaluating `(require (submod "clock.rkt" tock))` in the Racket REPL prints `"tick"` followed by `"tock"`.

For the same reason that module nesting does not imply a connection in instantiation times, module nesting alone does not imply a lexical-binding connection. For example,

```
(module clock racket
```

[5] The `require` form is a dependency declaration, not a side-effecting statement, and its placement relative to the `"tick"` expression does not matter. Putting `"tick"` before the `tock` submodule declaration will still print `"tock"` first, since a `required` module is instantiated before the importing module.

```
(define sound "tick")
(module* tick racket
  sound))
```

is a syntax error, because the `tock` module starts with only the bindings of `racket`, just like any module that declares `racket` as its language. Furthermore,

```
(module clock racket
  (define sound "tick")
  (module* tick racket
    (require (submod ".."))
    sound))
```

is also a syntax error, because `clock` does not export its `sound` binding. A submodule can specify `#f` as its initial language to indicate that the enclosing module body provides the submodule's bindings:

```
(module clock racket
  (define sound "tick")
  (module* tick #f
    sound))
```

In this case, `(racket (submod "clock.rkt" tock))` prints `"tick"`, which is the value of `sound`.

The `module+` form, which we used in section 3, is a macro that expands to `module*` with `#f` as its language. The `module+` form also collects multiple declarations with the same module name and concatenates the bodies in a single submodule declaration, so `(module+ test ....)` can be used multiple times in a module to build up one `test` submodule.

Submodules can be nested under `begin-for-syntax`. For a submodule declared with `module` or with `module*` and a non-`#f` initial import, the nesting has no effect on the submodule, since the submodule starts with a fresh lexical context. Nesting `module*` with a `#f` initial import under `begin-for-syntax` has the effect of shifting the enclosing module's bindings *down* by one phase in the body of the submodule. That is, a submodule's body by definition starts at phase 0 relative to the submodule, so if the submodule is lexically at phase *ph* relative to an enclosing module, then the enclosing module is at phase *-ph* relative to the submodule.

Relative phase shifts are useful in the case of documentation submodules, where all of the bindings inside of a module are relevant for the submodule, but the submodule should not have a direct execution dependence on the enclosing module. For example, building documentation should not require running the documented library. Figure 2 sketches the expansion of the module with documentation in figure 1. In the expansion, contracts for each function declaration are moved to a `provide...contract-out` form, the function definitions contain only the function implementations, and all documentation is moved into a `srcdoc` submodule. The submodule is declared under `begin-for-syntax`, so that the enclosing module is at template time—i.e., an unspecified time in the future—relative to the implementation of its documentation; the `code` form can then reflect on binding information inherited from the enclosing module to properly hyperlink references to identifiers that are used in the function contracts and documentation prose.

5. Model

Our model of submodules in Racket shows how modules are compiled and instantiated, including support for submodules, macros, and macros that expand to submodule declarations. The model is implemented in PLT Redex (Felleisen et al. 2010), and while we cover only the most relevant details here, the full model is available in executable and typeset forms at

`http://www.cs.utah.edu/plt/submod3/`

```
#lang at-exp racket/base
(require "doc-define.rkt"
         racket/contract/base)

(provide
 (contract/out
  [seconds->hours (-> exact-integer? exact-integer?)]
  [current-seconds (-> exact-integer?)]
  [current-hours (-> exact-integer?)]))

(define (seconds->hours secs) (quotient secs (* 60 60)))
(define current-seconds ....)
(define current-hours ....)

(begin-for-syntax
 (module srcdoc #f
  (require scribble/base
           scribble/manual)

  (define (current-docs what)
    @list{Returns @what since the epoch.})

  (define doc
   (list
    (format-function-doc
     @code{seconds->hour}
     (list @code{src})
     @code{(-> exact-integer? exact-integer?)}
     @{Takes @code{secs}, a number of seconds
       since the Epoch, ....})
    (format-function-doc
     @code{current-seconds}
     @code{()}
     @code{(-> exact-integer?)}
     @{@current-docs["seconds"]})
    (format-function-doc
     @code{current-hours}
     @code{()}
     @code{(-> exact-integer?)}
     @{@current-docs["hours"]})))))
```

Figure 2: Sketch of macro expansion for figure 1

mod ::= **MODULE**(*mname*, *dep* ..., *defn* ...)

dep ::= *mname*@*ph*
defn ::= ⟨*dname*@*ph*, *kind*, *ast*⟩

kind ::= **VALUE** | **MACRO**
ph ::= *integer*

mname ::= a module name
dname ::= a defined name

ast ::= a compiled expression

M ::= a mapping from *mname* to *mod*
Σ ::= a store

Figure 3: Compiled-module representation

5.1 Running Modules

The representation of a compiled module, *mod*, is shown in figure 3. A *mod* contains the module's name, a set of modules that the module depends on, and a sequence of definitions for the module's body. Each dependency and definition is associated with a particular phase, *ph*. A normal `require` of a module creates a dependency with phase 0, while a `require` with `for-syntax` creates a dependency with phase 1, and a `require` with `for-template` creates a dependency with phase -1.

Each definition is either for a run-time value or a macro, as indicated by *kind*, and its associated phase. A value definition is evaluated at its associated phase and can be referenced in its associated phase. A macro definition at phase *ph* can be referenced from a phase *ph* context, although the macro itself runs at phase *ph*+1.

run : *mname ph M Σ → Σ*

Run *mname* (as declared in *M*) at phase ph_Δ and record in Σ, but only if $mname@ph_\Delta$ is not already recorded as ran in Σ.

$\mathsf{run}[\![mname, ph_\Delta, M, \Sigma]\!] = \Sigma$
subject to $\Sigma(mname@ph_\Delta) = \textbf{READY}$
$\mathsf{run}[\![mname, ph_\Delta, M, \Sigma]\!] = \mathsf{run\text{-}body}[\![defn\ ..., mname, ph_\Delta, \Sigma_{deps}]\!]$
subject to $M(mname) = \textbf{MODULE}(mname, deps, defn\ ...)$,
$\Sigma[\![mname@ph_\Delta \leftarrow \textbf{READY}]\!] = \Sigma_{init}$,
$\mathsf{run}^\star[\![deps, ph_\Delta, M, \Sigma_{init}]\!] = \Sigma_{deps}$

run* : *dep ... ph M Σ → Σ*

Run each *dep* (which is a *mname*@*ph*) with phase shift *ph*.

$\mathsf{run}^\star[\![\varepsilon, ph_\Delta, M, \Sigma]\!] = \Sigma$
$\mathsf{run}^\star[\![mname@ph\ dep\ ..., ph_\Delta, M, \Sigma]\!] = \mathsf{run}^\star[\![dep\ ..., ph_\Delta, M, \Sigma_{new}]\!]$
subject to $\mathsf{run}[\![mname, ph_\Delta{+}ph, M, \Sigma]\!] = \Sigma_{new}$

run-body : *defn ... mname ph Σ → Σ*

Evaluate each run-time *defn* with phase shift ph_Δ, as long as ph_Δ plus the enclosing module's instantiation phase lands at phase 0. Record definitions in Σ.

$\mathsf{run\text{-}body}[\![\varepsilon, mname, ph_\Delta, \Sigma]\!] = \Sigma$
$\mathsf{run\text{-}body}[\![\langle dname@ph_{def}, \textbf{VALUE}, ast\rangle\ defn\ ..., mname, ph_\Delta, \Sigma]\!]$
$= \mathsf{run\text{-}body}[\![defn\ ..., mname, ph_\Delta, \Sigma_{new}]\!]$
subject to $ph_\Delta{+}ph_{def} = 0$, $\mathsf{eval}[\![\mathsf{shift}[\![ast, ph_\Delta]\!], \Sigma]\!] = val$,
$\Sigma[\![\langle 0, mname, dname\rangle \leftarrow \langle \textbf{VALUE}, val\rangle]\!] = \Sigma_{new}$
$\mathsf{run\text{-}body}[\![defn_{skip}\ defn\ ..., mname, ph_\Delta, \Sigma]\!]$
$= \mathsf{run\text{-}body}[\![defn\ ..., mname, ph_\Delta, \Sigma]\!]$

Figure 4: The run metafunction

A module is instantiated and run with the run metafunction, as shown in figure 4. A store Σ is updated with module instantiations and returned by run. The run metafunction evaluates a module body, but only after folding run across all dependencies of the module. For each dependency, if a module was imported `for-syntax`, then its definitions will be shifted higher by one phase, and if a module was imported `for-template`, then its definitions will be shifted lower by one phase.

A *mod* does not record the original module's submodules, if any. Instead, the module repository *M* contains all compiled module definitions, including those that were submodules. For simplicity, the model assumes that all modules and submodules in a program have globally distinct names. Any module–submodule or not–that is not a dependency of the initially run module will itself not run.

5.2 Compiling Modules

The *M* module repository used by run must be generated from a sequence of module declarations, except for the predefined module `base`. A module declaration starts as an S-expression, *s-exp*, which is either a name[6] or a parenthesized sequence of S-expressions, as shown in figure 5. The compiler must take an S-expression representation of a module, such as

```
(module m base (define x (quote ok)))
```

and turn it into an executable form, *mod*.

The entry point for compilation is cmodule, which takes an S-expression for a single module along with a repository of previously compiled modules and returns an updated repository:

[6] We use the term *name* instead of *symbol* for syntactic names. We use *symbol* for the run-time reflection of such names, which always appear in our model with the **QUOTE** constructor. See Flatt et al. (2012, section 3.2.1) for further discussion.

s-exp ::= *name* | (*s-exp* ...)
name ::= a token such as `x`, `clock`, or `lambda`

mname ::= *name*
dname ::= *name*

stx ::= *id* | (*stx* ...)
id ::= **STX**(*name*, *bindings*)

binding ::= ⟨*ph*, *mname*, *dname*⟩
bindings ::= *binding* ...

body ::= *stx*@*ph*

import : *mname ph M → bindings*

Collect a module's exports into a set of *bindings*.

wrap : *s-exp bindings → stx*

Attach bindings to a raw S-expression.

strip : *stx → s-exp*

Strip bindings from a syntax object to get an S-expression.

resolve : *id ph → binding*

Find relevant binding at a given phase, *ph*.

Figure 5: Syntax objects and bindings

cmodule : *s-exp M → M*

$\mathsf{cmodule}[\![\texttt{(module}\ mname\ mname_{init}\ s\text{-}exp\ ...\texttt{)}, M]\!]$
$= \mathsf{cbody}[\![body\ ..., mname, mname_{init}@0, \varepsilon, M, \Sigma_{init}]\!]$
subject to $\mathsf{visit}[\![mname_{init}, 0, M, \emptyset]\!] = \Sigma_{init}$,
$\mathsf{import}[\![mname_{init}, 0, M]\!] = bindings$,
$\mathsf{wrap}[\![s\text{-}exp, bindings]\!]@0\ ... = body\ ...$

Compilation of a module begins by *visiting* the module for the new module's initial import. The visit metafunction is just like run, but instead of running only phase 0 value definitions, it runs all definitions for the given module in phase 1 or higher, as well as all macro definitions at phase 0. The visit metafunction is called with an empty store, which reflects that every module compilation starts with fresh state.

Meanwhile, the import metafunction extracts a set of bindings from the initial import, as shown in figure 5; the model omits `provide`, and instead assumes that all definitions of a module are exported. The collected bindings are then applied to the body of the module to be compiled via wrap, which converts the module's body from S-expressions to syntax objects, *stx*. In general, syntax objects enable lexically scoped macros (Dybvig et al. 1993; Flatt et al. 2012), but we simplify here by considering only model-level scope.

Finally, each syntax object for a module's body is paired with the phase at which it appears, producing a *body* element of the form *stx*@*ph*. Initially, all body forms are at phase 0, but `begin-for-syntax` may later move forms to a higher phase. This phase shifting is implemented in the cbody metafunction, which completes compilation of the module.

deps ::= *dep* ...
defns ::= *defn* ...

cbody : *body ... mname deps defns M Σ → M*

In addition to the *body* sequence, cbody receives the name of the module being compiled (which is eventually used to create a *mod* and add it to *M*), a sequence of dependencies to extended by `require` forms among the *body*s, a sequence of compiled *defn*s to

form ::= `(begin-for-syntax` *form* ...`)`
| `(define` *dname expr*`)`
| `(define-syntax` *dname expr*`)`
| `(require` *req*`)`
| `(module` *mname mname form* ...`)`
| `(module*` *mname mname form* ...`)`
| `(module*` *mname* `()` *form* ...`)`
| `(`*dname arb* ...`)`
req ::= *mname*
| `(for-syntax` *req*`)`
| `(for-template` *req*`)`
expr ::= an expression
arb ::= whatever the macro allows

Figure 6: Module syntax recognized by cbody

be extended by `define` and `define-syntax` forms among the *body*s, the set of previously compiled modules M to be extended by submodule declarations among the *body*s, and the current store Σ.

The simplest case of cbody is when no more *body*s are left, in which case the accumulated dependencies and definitions are combined into a *mod* and added to the result M:

$$\begin{aligned}&\text{cbody}[\![\varepsilon, mname, deps, defns, M, \Sigma]\!]\\ &= M\big[mname \leftarrow \textsc{Module}(mname, deps, defns)\big]\end{aligned}$$

Otherwise, cbody dispatches on the shape of the first *body*, matching one of the *form* cases of figure 6. The grammar of *form* shows `begin-for-syntax` as its first case. More precisely, from the perspective of cbody, a *body* to match that case must have a syntax object containing an identifier whose binding is `begin-for-syntax` from the pre-defined `base` module. That is, the forms listed in the grammar for *form* are available only when `base` is imported or via macro expansion from a module that imports `base`. The last *form* production is a macro invocation, where macro expansion can produce any of the other forms (or another macro invocation).

The cbody rule for `begin-for-syntax` is thus

$$\begin{aligned}&\text{cbody}[\![(id_{bfs}\ stx\ ...)\ @ph\ body_{rest}\ ..., mname, deps, defns, M, \Sigma]\!]\\ &= \text{cbody}[\![stx@ph{+}1\ ...\ body_{rest}\ ..., mname, deps, defns, M, \Sigma]\!]\\ &\text{subject to resolve}[\![id_{bfs}, ph]\!] = \langle ph, \texttt{base}, \texttt{begin-for-syntax}\rangle\end{aligned}$$

The syntax objects within `begin-for-syntax` get shifted up by one phase and added back to the list of bodies for the module.

We omit the rules for `define`, `define-syntax`, and macro invocation. For the purpose of explaining submodules, an interesting aspect of the omitted rules is that definitions are evaluated only when they have a phase greater than 0, so that they can be used in macro implementations, while definitions at phase 0 are merely compiled.

The `require` case of cbody uses a parse-req metafunction (not shown) to turn nested `for-syntax` and `for-template` specifications into a phase shift:

$$\begin{aligned}&\text{cbody}[\![(id_{req}\ stx_{in})\ @ph\ body_{rest}\ ..., mname, dep\ ..., defns, M, \Sigma]\!]\\ &= \text{cbody}[\![body_{new}\ ..., mname, dep\ ...\ dep_{new}, defns, M, \Sigma_{new}]\!]\\ &\text{subject to resolve}[\![id_{req}, ph]\!] = \langle ph, \texttt{base}, \texttt{require}\rangle,\\ &\qquad \text{parse-req}[\![stx_{in}, ph]\!] = \langle mname_{in}, ph_{\Delta}\rangle,\\ &\qquad \text{visit}[\![mname_{in}, ph_{\Delta}, M, \Sigma]\!] = \Sigma_{new},\\ &\qquad \text{import}[\![mname_{in}, ph_{\Delta}, M]\!] = bindings_{new},\\ &\qquad \text{add}[\![body_{rest}, bindings_{new}]\!]\ ... = body_{new}\ ...,\\ &\qquad mname_{in}@ph_{\Delta} = dep_{new}\end{aligned}$$

Having extracted a module name $mname_{in}$ and relative phase ph_{Δ}, the `require` rule of cbody is essentially the same as cmodule: the imported module is visited, its bindings are added to the rest of the module, and the module is recorded as a dependency.

$$\begin{aligned}&\text{cbody}[\![(id_{mod}\ id_{sub}\ id_{init}\ stx\ ...)\ @ph\ body_{rest}\ ..., mname, deps, defns, M, \Sigma]\!]\\ &= \text{cbody}[\![body_{rest}\ ..., mname, deps, defns, M_{new}, \Sigma]\!]\\ &\text{subject to resolve}[\![id_{mod}, ph]\!] = \langle ph, \texttt{base}, \texttt{module}\rangle,\\ &\qquad \text{strip}[\![id_{sub}]\!] = mname_{sub}, \text{strip}[\![id_{init}]\!] = mname_{init},\\ &\qquad \text{cmodule}[\![(\texttt{module}\ mname_{sub}\ mname_{init}\ \text{strip}[\![stx]\!]\ ...), M]\!] = M_{new}\end{aligned}$$

$$\begin{aligned}&\text{cbody}[\![(id_{mod}\ id_{sub}\ id_{init}\ stx\ ...)\ @ph\ body_{rest}\ ..., mname, deps, defns, M, \Sigma]\!]\\ &= \text{cmodule}[\![(\texttt{module}\ mname_{sub}\ mname_{init}\ \text{strip}[\![stx]\!]\ ...), M_{new}]\!]\\ &\text{subject to resolve}[\![id_{mod}, ph]\!] = \langle ph, \texttt{base}, \texttt{module*}\rangle,\\ &\qquad \text{strip}[\![id_{sub}]\!] = mname_{sub}, \text{strip}[\![id_{init}]\!] = mname_{init},\\ &\qquad \text{cbody}[\![body_{rest}\ ..., mname, deps, defns, M, \Sigma]\!] = M_{new}\end{aligned}$$

$$\begin{aligned}&\text{cbody}[\![(id_{mod}\ id_{sub}\ ()\ stx\ ...)\ @ph\ body_{rest}\ ..., mname, deps, defns, M, \Sigma]\!]\\ &= \text{cbody}[\![\text{shift}[\![stx, 0{-}ph]\!]@0\ ..., mname_{sub}, mname@0{-}ph, \varepsilon, M_{new}, \Sigma_{init}]\!]\\ &\text{subject to resolve}[\![id_{mod}, ph]\!] = \langle ph, \texttt{base}, \texttt{module*}\rangle,\\ &\qquad \text{strip}[\![id_{sub}]\!] = mname_{sub},\\ &\qquad \text{cbody}[\![body_{rest}\ ..., mname, deps, defns, M, \Sigma]\!] = M_{new},\\ &\qquad \text{visit}[\![mname, 0{-}ph, M_{new}, \varnothing]\!] = \Sigma_{init}\end{aligned}$$

Figure 7: Submodule cases of cbody

5.2.1 Compiling Submodules

The remaining cases of cbody handle submodules, as shown in figure 7. The `module` rule and first `module*` rule are similar: the submodule is compiled using cmodule, and the difference is only whether the submodule is compiled before or after the rest of the body of the current module. A `module` form is compiled before the rest of the body, so that it can be used by a later `require` in the current module. A `module*` form is compiled after the rest of the current module's body, so that the submodule can `require` the current module.

The last rule in figure 7 handles `(module*` *mname* `()` *form* ...`)`, which represents a submodule that inherits all bindings of its enclosing module. (We use `()` instead of `#f` in the model to avoid the need for boolean literals in S-expressions and syntax objects.) Inheriting all bindings of the enclosing module is different from importing the enclosing module, because it makes any imports of the enclosing module visible in the submodule, as opposed to only the definitions of the enclosing module. Therefore, instead of compiling the submodule via a context-stripping cmodule, the submodule's compilation uses cbody directly. At the same time, the bindings inherited by the submodule must be shifted by a negative amount that corresponds to the submodule's phase nesting within its enclosing module. Although the submodule inherits bindings of the enclosing module, it does not inherit the store; the submodule's compilation via cbody starts with a fresh store that is initialized by visiting the enclosing module at the appropriate phase offset.

As noted for the first cbody rule, the result of cbody is ultimately a set of compiled modules to act as the result of cmodule. Folding cmodule over a set of S-expressions that represent a program accumulates a set of compiled modules, which then can be passed to run with a main-module name to run the program. The full model on the web site includes several example programs as tests.

6. Implementation and Discussion

We added submodules to Racket in version 5.3 (August 2012), which is 10 years after originally adding modules to Racket (then PLT Scheme). To support nested scopes, the initial implementation of modules was soon paired with a `package` macro that imitates the nestable `module` form of Chez Scheme (Waddell and Dybvig 1999), but such nested scopes never found much use in the Racket code base. Submodules, in contrast, have found immediate and

widespread use, solving many different problems that we did not originally recognize as related: how to have code that is run when the module is "main" (in a more principled way than a Python-style dynamic test), how to include test code with a library's implementation, how to manage documentation in a library's implementation, how to provide extra exports from a module that are available only when specifically requested (by importing the submodule in addition to its enclosing module), how to package a read-time parser for a new language alongside its compile-time and run-time implementation (where read time is represented by a submodule), and how to declare dynamic file dependencies for use by a packaging tool (where the packaging tool can run a submodule to get information about the needed files). Naturally, we take the fact that submodules are conceptually simple but solve many problems as evidence that the submodule design is on the right track.

The key idea in our design is to allow a nested namespace to have its own dynamic extent relative to its enclosing environment. In a sense, submodules are a "meta" form of closures: in the same way that `(define (f x) (lambda (y) x))` returns a function that has access to the argument `x` beyond the dynamic extent of a call to `f`, submodules provide a way for a nested module to refer to the bindings of an enclosing module without necessarily implying a connection on the module extents (depending on how the bindings are referenced). More generally, lexical scope converts a potentially complex temporal question—how to ensure that a binding is available when it is needed—into a spatial problem that is easier for humans to reason about; that benefit applies just as much to modules, phases, and submodules as to function closures.

Racket's submodule design would look simpler if `module` and the two `module*` variants (with and without an initial import) could be collapsed into a single syntactic form. It may be possible to collapse `module` and `module*` and have the compiler infer (based on later `require`s) whether a submodule must be compiled before or after the rest of the enclosing module's body. The difference between `#f` or a module name in the initial-import position of `module*` might also be managed by cleaner syntax. We leave these problems for future work, and as a practical matter, choosing `module` or `module*` is easy enough for a Racket programmer.

The Racket implementation of modules includes a primitive `for-label` form in addition to `for-syntax` and `for-template`. A `for-label` import corresponds to binding at phase $-\infty$: arbitrarily far in the future. The `code` form for documentation looks for bindings at this label phase for generating hyperlinks, and in the expansion sketch of figure 2, the generated module's body has been shifted by $-\infty$ (but that fact is invisible in the sketch). The label phase is a kind of optimization hint to the module system, where a `for-label` dependency implies that no execution of the module at any finite phase will require the execution of the dependency. The label phase seems useful, but we are not yet sure whether it is fundamentally necessary.

As suggested by the model, top-level `module` forms are the unit of compilation in Racket. When a program is run from source in Racket, then a module and all of its submodules must be compiled together. In the case of a library module that contains its own documentation, this compilation process involves much more code than is needed to just run the library, negating an intended benefit of submodules. Racket modules are normally compiled to bytecode in advance, and the bytecode for a module starts with a directory of all submodules separate from the main module code, so that the module or any individual submodule can be loaded independently. Thus, a key benefit of submodules in practice relies on bytecode compilation.

7. Related Work

By enabling programming at different layers, submodules play a role similar to Java annotations and C# attributes. JavaDoc, in particular, is a use of Java annotations in the same way that Racket uses submodules for in-source documentation. Java annotations allow the decoration of code with data, and they are preserved through run time, so that annotations can be inspected in source, compiled code or reflectively. Still, Java annotations are limited to data, so that any abstraction or programmatic interpretation of the data depends on yet another external tool and language, or else the code part (such as test to run for a `@Test` annotation) is difficult to separate from the main program. C# attributes can have associated methods, but attribute code is still mingled with run-time code. Submodules, in contrast, generalize annotations to make them "live," so that the language of annotations can include expressions, function, and even syntactic extension, without necessarily tangling the submodule code with the base code. At the same time, submodules allow these live annotations to connect with the lexical scope of the associated code, which is useful in cases such as testing and documentation.

Although the model of submodules in this paper uses a simplistic notion of scope for syntax objects, in practice and in spirit submodules build on a long line of work on hygienic macros (Dybvig et al. 1993; Flatt et al. 2012; Kohlbecker et al. 1986). The R^{6}RS standard for Scheme (Sperber 2007) also includes modules with phases, as directly influenced by our previous work, but R^{6}RS does not include submodules or `begin-for-syntax`, and it does not allow macro expansion to introduce new imports within a module. R^{6}RS also does not require implementations to enforce a phase distinction. Some implementations support implicit phasing (Ghuloum and Dybvig 2007), where `for-syntax` and `for-template` annotation on imports are inferred automatically based on the use of imported identifiers. In Racket, we stick with explicit phasing because it supports different bindings for the same name in different phases (which is useful, for example, when documenting the `define` form of some language in a document that is implemented with Racket's normal `define`) and because explicit phasing allows macro expansion to depend in a reasonable way on side effects (as a last resort, but a useful one). We also suspect that checking programmer expectations against actual references is more useful in our context than implicit phasing's inference.

In TemplateHaskell (Jones and Sheard 2002), macro implementations are restricted to pure functions, which perhaps lessens the need to track phases; there is no question, for example, of initializing a GUI subsystem as part of a macro expansion, and in the absence of side-effects as side channels, the code that is accessed at compile time can be more easily limited to that needed by the macro's implementation. Since functions used by macros can be implemented themselves with macros, TemplateHaskell effectively supports arbitrary phase levels, the phases are implicit, and bindings are the same across all phases. TemplateHaskell does not support phases other than the implicit phases of run and compile times, and in-source documentation with Haddock (Marlow 2002) is analogous to JavaDoc. The Converge programming language (Tratt 2005) infers phases for macro expansion in a similar way to TemplateHaskell.

The SugarJ (Erdweg et al. 2011) design for library-based language extensibility relies on a separation between parsing code and transformation code, while allowing the two parts to coexist in a library. In SugarJ, the languages of parsing and transformation are separate and distinct from the base language. Nevertheless, there is room for a scope connection that is currently absent in SugarJ, which would connect the output of the parser to specific transfor-

mations or binding references in transformations to particular base-language bindings.

Lightweight Module Staging (Rompf 2012; Rompf and Odersky 2010), or LMS, is an approach to code generation where the type system is used to separate code at different levels. While the details of LMS are different from Racket's macros—particularly the way that run-time values can be implicitly coerced to representations of values—LMS is effective for fundamentally the same reason as modules and phases: both rely on scope and binding to separate phases, and both allow phases to lexically overlap without destroying a programmer's ability to reason about the code.

More generally, multi-stage programming languages such as MetaML (Taha and Sheard 2000) include a notion of phases that is similar to the notion in Racket. Racket's tracking of phases differs in that a syntax object can accumulate bindings in many phases, and the determination of the relevant binding can be made late, when the identifier is used in an expression position. Staged languages more typically identify the phase of each expression and identifier statically. MetaML and other statically typed multi-stage languages not only ensure that an identifier is used at a suitable phase, but they ensure that the identifier is used in a suitable type context.

Nested modules in languages such as ML (OCaml, SML/NJ) and Chez Scheme (Waddell and Dybvig 1999) are namespace-management tools, where nested modules are always instantiated with the enclosing module. Racket submodules, in contrast, are separately loadable and separately instantiable in the same way that top-level modules are separately loadable and instantiable.

Java classes are related to submodules in that the declaration of two classes within a single compilation unit does not imply that both classes must be used together at run time. Instead, classes are loaded and initialized on demand in a Java implementation, which allows a Java implementation to avoid loading code that is unused, and it ensures dynamically that any code that is needed is loaded before it is used. With a macro-extensible Java in the style of Maya (Baker and Hsieh 2002), code that is used only at compile time will naturally not be loaded at run time. This approach has the same benefits and drawbacks as implicit scoping in Scheme (e.g., the difficulty of reasoning about initialization side effects).

In this paper, we have used in-source documentation as one motivation for submodules. We reported on in-source documentation for Racket in previous work (Flatt et al. 2009), but that report glosses over certain problems. Documentation was extracted via `include-extracted` by re-expanding the source module and pulling designated syntax objects out of the expansion. Besides being ugly and reporting syntax errors late, the implementation had quadratic complexity for extracting N sets of documentation from a single module. Our revised implementation of in-source documentation uses a submodule to avoid these problems.

8. Conclusion

Lexical scope is a powerful organizing principle. In the context of Racket, we believe that lexically scoped macros are the key to our ability to define a rich ecosystem of language variants and tools, from teaching dialects of Racket, to statically typed dialects of Racket, to documentation languages. Phasing is a natural extension of lexical scope that adds a new dimension to each identifier, allowing the identifier to have different meanings in different phases while relying on the surrounding context of an identifier to also determine the phase in which the identifier is used. Submodules continue that extension; they add new expressive power to Racket in line with the principles of lexical scope, they solve a variety of practical problems for Racket programmers, and they increase the ability of macros to implement new language constructs.

Acknowledgements I would like to thank Matthias Felleisen, Robby Findler, Ryan Culpepper, the Racket community, and anonymous reviewers for feedback and suggestions on the design of submodules and this presentation.

Bibliography

Jason Baker and Wilson C. Hsieh. Maya: Multiple-Dispatch Syntax Extension in Java. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 270–281, 2002.

Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced Macrology and the Implementation of Typed Scheme. In *Proc. Wksp. Scheme and Functional Programming*, 2007.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), pp. 295–326, 1993.

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-Based Syntactic Language Extensibility. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 391–406, 2011.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2010.

Matthew Flatt. Composable and Compilable Macros: You Want it *When?* In *Proc. ACM Intl. Conf. Functional Programming*, pp. 72–83, 2002.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 109–120, 2009.

Matthew Flatt, Ryan Culpepper, Robert Bruce Findler, and David Darais. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *J. Functional Programming* 22(2), pp. 181–216, 2012.

Abdulaziz Ghuloum and ; R. Kent Dybvig. Implicit Phasing for R6RS Libraries. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 303–314, 2007.

Simon Peyton Jones and Tim Sheard. Template metaprogramming for Haskell. In *Proc. ACM Wksp. Haskell*, pp. 1–16, 2002.

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proc. ACM Sym. Principles of Programming Languages*, 2012.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proc. Lisp and Functional Programming*, pp. 151–181, 1986.

Simon Marlow. Haddock, a Haskell Documentation Tool. In *Proc. ACM Wksp. Haskell*, pp. 78–89, 2002.

Tiark Rompf. Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming. PhD dissertation, École Polytechnique Fédérale de Lausanne, 2012.

Tiark Rompf and Martin Odersky. Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. Generative Programming and Component Engineering*, pp. 127–136, 2010.

Michael Sperber (Ed.). The Revised ^{6} Report on the Algorithmic Language Scheme. 2007.

Walid Taha and Tim Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science* 248(1-2), pp. 211–242, 2000.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 395–406, 2008.

Laurence Tratt. Compile-time Meta-programming in a Dynamically Typed OO Language. In *Proc. Dynamic Languages Symposium*, pp. 49–63, 2005.

Oscar Waddell and R. Kent Dybvig. Extending the Scope of Syntactic Abstraction. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 203–213, 1999.