Composable and Compilable Macros

You Want it *When?*

Matthew Flatt
University of Utah

Abstract

Many macro systems, especially for Lisp and Scheme, allow macro transformers to perform general computation. Moreover, the language for implementing compile-time macro transformers is usually the same as the language for implementing run-time functions. As a side effect of this sharing, implementations tend to allow the mingling of compile-time values and run-time values, as well as values from separate compilations. Such mingling breaks programming tools that must parse code without executing it. Macro implementors avoid harmful mingling by obeying certain macro-definition protocols and by inserting phase-distinguishing annotations into the code. However, the annotations are fragile, the protocols are not enforced, and programmers can only reason about the result in terms of the compiler's implementation. MzScheme—the language of the PLT Scheme tool suite—addresses the problem through a macro system that separates compilation without sacrificing the expressiveness of macros.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages—*language constructs and features, Scheme*; D.3.4 [**Software**]: Processors—*parsing, preprocessors*; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Languages, Design

Keywords

Macros, modules, language tower

1 Introduction

Macro systems provide a convenient interface for extending a compiler to support new language constructs. In the most expressive macro systems, macro transformers are not constrained to mere pattern-matching transformations, but may perform arbitrary computation during expansion [12, 17, 3, 24, 26, 1]. In addition, macros may manipulate abstract syntax enriched with lexical information instead of manipulating raw source text [15, 2, 4, 8], which means that macro-defined constructs can be assigned a meaning independent of details of the macro's expansion (e.g., whether the macro introduces a local variable named *temp* or happens to call the `car` function). Finally, in the Lisp and Scheme tradition where macros are themselves defined in a macro-extensible language, extensions can be stacked in a "language tower." Each extension of the language can be used in implementing the next extension.

Trouble with Expressive Macro Systems. In a typical Scheme system, however, language towers cause trouble [19]. Advances in macro technology have simplified the creation of individual blocks for a tower, but they have not delivered a reliable mortar for assembling the blocks. For example, suppose `"P.scm"` is implemented in an extension of Scheme *E*, where *E* is implemented by `"E.scm"` directly in Scheme. A typical load sequence for *P* is

```
(load "E.scm")
(load "P.scm")
```

The above statements might be placed in a file `"loadP.scm"`, which can then be submitted to a Scheme interpreter to execute `"P.scm"` successfully. The problem starts when the programmer tries to compile the program for later execution. Supplying `"loadP.scm"` to the compiler is useless, because the result is simply the compiled form of two `load` statements. A full compiler will be needed at run-time when `"P.scm"` is actually loaded.

The problem is that the compile-time code in `"E.scm"` is not distinguished in any way from the run-time code in `"P.scm"`, and the run-time `load` operation is abused as a configuration-time operation. The conventional solution is to decorate `"loadP.scm"` and similar files with `eval-when` annotations [7, 23] that designate the intended *phase* of an expression:

```
(eval-when (compile) (load "E.scm"))
(load "P.scm")
```

This solution has three major weaknesses. First, the resulting annotations are fragile; small changes to the program organization can render a set of annotations incorrect. For example, suppose that `"E.scm"` initially contains only macro definitions, but a run-time support function is added. The `eval-when` annotation must be augmented with `load` to properly load the run-time parts of `"E.scm"`. Second, for large examples with tall language towers and with library code written in different extensions of Scheme, the correct `eval-when` annotations can be difficult to discern. Indeed, annotating only `(load "E.scm")` is probably not the right strategy

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

if "E.scm" defines a mixture of macros and run-time functions. Third, an incorrect set of annotations can appear to work correctly (for a time) due to the accidental implementation of compile-time functionality by run-time code that happens to be loaded. In general, static checking cannot ensure that variable bindings are satisfied by code from the right phase.

For macros to serve as reliable compiler extensions, the programming model must clearly separate the compile-time and run-time phases of all code at all times. The phases may be interleaved for interactive evaluation, but compiling new code must not affect the execution of previously compiled code. Similarly, the amount of interleaving should not matter: code should execute the same if it is compiled all in advance, if it is compiled with interleaved execution, or if half the code is compiled today and the rest is compiled on a different machine tomorrow. Finally, when a complete application is compiled, the programming environment should be able to strip all compile-time code from the final deliverable.

Reliable Macros in MzScheme. The new macro and module system in MzScheme (the implementation language of the PLT Scheme suite) supports compilable macros in the above sense. More concretely, the system ensures that if a program works correctly when loaded interactively in the read-eval-print loop, then it works correctly when run through the compiler, run in the debugger, parsed by the syntax checker, or expanded for static analysis—and vice-versa. The implemented system is backed up by a formal model. The model explains module compilation and demonstrates how computational effects, including the introduction of variable bindings, are confined to a single phase.

The module system avoids the problems of `eval-when` by making module dependencies explicit (instead of relying on the side-effects of `load`), and by distinguishing compile-time dependencies from run-time dependencies. Moreover, the macro system enforces a separation between different phases, i.e., compile-time variables are never resolved to run-time values that happen to be loaded.

Figure 1 illustrates module and macro programming in MzScheme. The module `M` imports variables and syntax from `L` using `require`. These `L` imports can be used for implementing run-time expressions in `M`, such as the right-hand side of a definition for *f*. In addition, `M` imports from `R` using `require-for-syntax`. The `R` imports can be used in implementing compile-time expressions in `M`, such as the right-hand side of the macro definition *s*. Meanwhile, module `B` imports both `M` and `R` with `require`. Enforcing the separation of compile time and run time means instantiating `R` at least twice: once for compiling `B`, and once for running `B`. Furthermore, separating different compilations means instantiating `R` yet again to compile `B2`, and so on.

Proper module instantiation is part of the solution, but two indispensable features of Scheme macros further complicate enforcing a phase separation:

- Macro-generating macros — A macro expansion can generate an expression that is to be run in the same phase as its generator. Such macro-generating macros are critically important to implement language extensions that bind compile-time information. For example, a class-definition form must bind compile-time information about the class's methods.
- Lexical scope — In the context of macros, lexical scope means that a free identifier introduced by a macro expansion refers to its binding in the macro-definition context, not the macro-use context, while a free identifier in the macro use refers to its binding in the macro-use context (unless the programmer explicitly "breaks hygiene") [8, 14]. Free variables thus bound may refer to either run-time values or other macro transformers (which potentially generate transformer expressions).

In terms of Figure 1, these complications affect the striped box next to *s* within `M`. The implementation of *s* will contain templated expressions that are used in the output of the macro. Some of templated code will turn out to be compile-time code, bound by striped imports from `R`, but some templated code will turn out to be run-time code, bound by polka-dotted imports from `L`. Separating the different parts is not statically decidable.

Tracking such dependencies requires an extension of previously known macro-expansion techniques. Our extension tracks the phase and phase-specific binding of each transformed identifier to resolve bindings correctly and at a well-defined time.

Our users' initial experience with the new macro and module system has been overwhelmingly positive. Previously, after developing a program interactively, the programmer would embark on a lengthy process of adding `eval-when`-like annotations to the program, carefully tuning calls to `load`, and finally divining the proper sequence of command-line flags to push the code through the compiler or analyzer. Libraries frequently failed to load when incorporated into a program in a previously untried order. When loading or compilation failed, users were at a loss to explain the failure. All of these experiences are typical for users of Scheme and Lisp implementations, but no longer in MzScheme. Moreover, the implementation of MzScheme itself relies on syntactic extension and language towers to a much greater extent than before. The result is a substantially improved code base and easier experimentation with new language constructs.

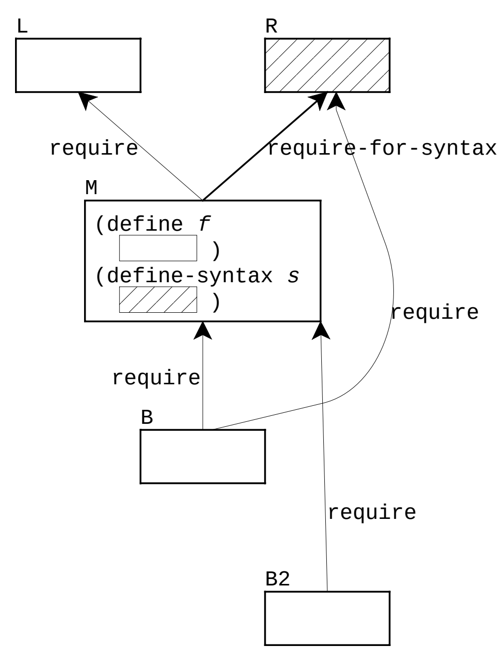

Figure 1. Example modules

Roadmap. Section 2 provides an overview of MzScheme macros and modules. Section 3 presents an example of syntactic extension that illustrates key problems in compiling macro-based code, and how MzScheme macros and modules solve the problems. Section 4 provides a few pragmatic details concerning macros and modules in MzScheme. Section 5 sketches a formal model with its phase-separation results. Section 6 summarizes related work.

2 A Macros and Modules Primer

In a module-based MzScheme program, all code resides within some module, whether the code implements a run-time function or a compile-time macro. The syntax of a `module` declaration is

```
(module module-name language-name
  body-element ...)
```

The *language-name* is usually `MzScheme`. In the `MzScheme` language, a *body-element* is either a definition, an expression (executed for its effect), a syntax definition, an import, or an export:

```
body-element ::= (define id expr⁰) | expr⁰
               | (define-syntax id expr¹)
               | (require req-spec ...)
               | (require-for-syntax req-spec ...)
               | (provide prov-spec ...)
```

The 0 superscript in $expr^0$ indicates that the expression is evaluated at run time, or "phase 0." The 1 superscript in $expr^1$ for `define-syntax` indicates that the expression is evaluated at compile time.

The `require` form imports bindings that are exported from another module. Bindings imported with `require` apply only to run-time expressions, i.e., the $expr^0$s in the module body. The `require-for-syntax` form is similar to `require`, but the imported bindings apply only to compile-time expressions, i.e., $expr^1$s.

The `provide` form exports a subset of a module's macro and variable bindings. Each exported binding must be either defined within the module with `define` or `define-syntax`, or imported into the module with `require`.

2.1 Using Modules

The following `Zoo` module provides functions for creating and manipulating zebra and lizard records:

```
(module Zoo MzScheme
  (provide zebra zebra? zebra-weight zebra-stripes
           lizard ———)
  ;; Creates a zebra record given its weight and stripes:
  (define (zebra weight stripes)
    (list 'zebra weight stripes))
  ;; Recognizes a zebra:
  (define (zebra? l)
    (and (list? l) (= 3 (length l))
         (eq? 'zebra (car l))))
  ;; Extracts a zebra's weight:
  (define (zebra-weight l)
    (list-ref l 1))
  ———
  (define (lizard weight length color)
    (list 'lizard weight length color))
  ———)
```

[A ——— represents elided code.] In a separate `Metrics` module, we can implement an *animal-weight* function using the functions from `Zoo`:

```
(module Metrics MzScheme
  (require Zoo)
  (provide animal-weight)
  (define (animal-weight a)
    (cond
     ((zebra? a) (zebra-weight a))
     ((lizard? a) (lizard-weight a)))))
```

When we invoke the `Metrics` module, the `Zoo` module is automatically executed, and it is executed before `Metrics`.

More generally, we define *invoke* on a module to mean executing the module's $expr^0$s, but only after executing the $expr^0$s of each `require`d module. The `require`-execution rule applies up the chain of modules, so that every module used (directly or indirectly) by an invoked module is executed before its importers. Unused modules are ignored, and modules used through multiple `require` paths are executed only once.[1]

2.2 Macros

In addition to exporting values, such as the *zebra* function, a module can export macros. For example, the `Zoo` module might provide a `zoo-switch` macro for conveniently dispatching on animal records, which we could then use to implement *animal-weight* more compactly as follows:

```
(define (animal-weight a)
  (zoo-switch a
   ((zebra w s) w)
   ((lizard w l c) w)))
```

The `Metrics` module is compiled by first loading the macro definitions of `Zoo`, which implies that `Zoo` must be compiled earlier. In other words, just as executing a module causes its imports to be executed first, compiling a module requires that its imports are compiled first. In addition, compiling a module executes the compile-time portions of imported modules to obtain macro transformers.

The `Zoo` module defines the `zoo-switch` macro using `define-syntax`:

```
(module Zoo MzScheme
  (provide zebra ——— lizard ——— zoo-switch)
  ———
  (define-syntax (zoo-switch stx)
    ———))
```

A macro is implemented as a transformer on syntax objects. The input syntax object (*stx* for `zoo-switch`) corresponds to the macro use, and the output syntax object represents the expansion. A syntax object is similar to an S-expression, except that it also encapsulates source-location and lexical information for each of its parts.

In the case of `zoo-switch`, every use of the macro must have two clauses—one for *zebra* and another for *lizard*—and the first clause must have two variables, while the second clause must have three variables. Thus, the *stx* argument must be a syntax object matching a particular shape. Input syntax is deconstructed using the pattern-matching `syntax-case` form [8]:

[1] The module-import relation must be acyclic. MzScheme provides a separate mechanism for defining *units* with mutually recursive references [9], and units are implemented with macros.

```
(define-syntax (zoo-switch stx)
  (syntax-case stx (zebra lizard)
    ((zoo-switch expr
       ((zebra w-name s-name) z-body ...)
       ((lizard w-name lt-name c-name) l-body ...))
     ———)))
```

In the `zoo-switch` pattern, `zebra` and `lizard` are literals (because they are listed before the pattern), and *`expr`*, *`w-name`*, *`s-name`*, and *`z-body`* are pattern variables. Within a pattern, ellipses (`...`) match a sequence of source sub-expressions to the preceding sub-pattern, so that each variable in the sub-pattern is bound to a list of successively matching source parts. Thus, the pattern for `zoo-switch` generates a list of *`z-body`*s when it matches, corresponding to the sequence of body expressions in the zebra clause.

The `zoo-switch` transformer must produce a `cond` expression whose clauses bind the variables provided in the macro use. After deconstructing syntax with `syntax-case`, a resulting syntax object is constructed with a `quote`-like `#'` form. Unlike `quote`, the content of `#'` can refer to pattern variables bound by `syntax-case`. Each pattern variable under `#'` is replaced by the matched sub-expression:

```
(define-syntax (zoo-switch stx)
  (syntax-case stx (zebra lizard)
    ((zoo-switch expr
       ((zebra w-name s-name) z-body ...)
       ((lizard w-name lt-name c-name) l-body ...))
     #'(let ((val expr))
         (cond
          ((zebra? val)
           (let ((w-name (zebra-weight val))
                 (s-name (zebra-stripes val)))
             z-body ...))
          ———)))))
```

Within a `#'`-quoted template, ellipses duplicate the preceding sub-template so that, for each duplication of the sub-template and for each variable in the sub-template, one source part is used from the variable's list of matching parts. Thus, the output expression for `zoo-switch` lists the same sequence of *`z-body`*s that matched the input pattern.

Free variables inside a `#'` template (that are not bound to pattern variables) obtain their bindings from the environment of the template, not the environment of the macro use. Thus, *`zebra-weight`* in the expansion of `zoo-switch` always refers to the definition in `Zoo`, even if the context of the use of `zoo-switch` has a different binding for *`zebra-weight`*.

2.3 Compilation and Phases

The result expression in a `syntax-case` clause need not be an immediate `#'` expression. Instead, the result expression may perform arbitrary computation at compile time. One common use for compile-time computation is error checking. For example, we can improve the `zoo-switch` macro by detecting multiple bindings of an identifier within a clause, as in the following expression:

```
(zoo-switch a
 ((zebra w w) w) ;; ← multiple bindings for w
 ((lizard w l c) w))
```

To implement the duplicate-variable check, the result part of the `syntax-case` clause for `zoo-switch` consists of a sequence of expressions: two to check for duplicate bindings in the two clauses, and one to generate the macro expansion.

```
(define-syntax (zoo-switch stx)
  (syntax-case stx (zebra lizard)
    ((zoo-switch expr
       ((zebra w-name s-name) z-body ...)
       ((lizard w-name lt-name c-name) l-body ...))
     (begin
       (check-dups #'(w-name s-name))
       (check-dups #'(w-name lt-name c-name))
       #'(let ((val expr))
           ———)))))
```

Many macros must check for duplicate variables, so we implement the *`check-dups`* function in its own `Check` module:

```
(module Check MzScheme
  (provide check-dups)
  (define (check-dups variables)
    ———))
```

To make *`check-dups`* available to the implementation of `zoo-switch`, `Zoo` must import `Check`. Since the function is needed at compile time, not at run time, `Zoo` imports `Check` using `require-for-syntax`:

```
(module Zoo MzScheme
  (require-for-syntax Check)
  ———
  (define-syntax (zoo-switch stx)
    ———))
```

Whenever the compile-time portion of `Zoo` is executed (e.g., to compile `Metrics`), the *run-time* portion of `Check` is executed, due to the `require-for-syntax` import. Thus, the *`check-dups`* function is available whenever the transformer for `zoo-switch` might be applied.

When the run-time portion of `Zoo` is executed, `Check` is ignored. Indeed, *`check-dups`* is not even bound in the run-time expressions of `Zoo`, so it cannot be used accidentally at run time. Similarly, if `Check` were imported with `require` instead of `require-for-syntax`, then *`check-dups`* would not be bound in the implementation of `zoo-switch`. Modules must not contain free variables, so incorrectly importing `Check` with `require` instead of `require-for-syntax` would lead to a syntax error for the free occurrences of *`check-dups`*.

In general, we define *visit* on a module to mean executing its $expr^1$s, but only after invoking each `require-for-syntax`ed module. As we see in the next section, visiting a module also visits the module's `require`d modules.

2.4 Execution and Phases

When a module is invoked, the need to invoke `require`d modules is obvious: before an expression within a module can be evaluated, imported variables must be first initialized. Furthermore, a chain of initialization dependencies, often in the form of a chain of function calls, forces a chain of invocations through `require`. For example, a `Zookeeper` module might import `Metrics` and call *`animal-weight`*, which in turn calls *`zebra?`* in `Zoo`.

Though less obvious, visiting a module must also visit `require`d modules, in case macro uses are chained. For example, `Metrics` might export a `zoo-weight-switch` macro that expands to `zoo-switch`, but exposes only the weight field in each clause:

```
(define-syntax (zoo-weight-switch stx)
  (syntax-case stx (zebra lizard)
    ((zoo-weight-switch expr
       ((zebra w-name) z-body ...)
       ((lizard w-name) l-body ...))
     #'(zoo-switch expr
        ((zebra w-name hide-s) z-body ...)
        ((lizard w-name hide-l hide-c) l-body ...)))))
```

If the Zookeeper module uses zoo-weight-switch, then the macro transformer from Metrics is applied, and the result is a zoo-switch expression. To continue expanding, the zoo-switch transformer from Zoo is called. Thus, the compile-time portion of Zoo must be executed whenever the compile-time portion of Metrics is executed.

3 Putting Macros and Modules to Work

Although we can define an animal-specific zoo-switch form that works with hand-rolled data structures, we would certainly prefer a general define-record form with a corresponding record-switch dispatching form. Indeed, many such record-declaration extensions to Scheme have been implemented [10, 13, 21, 27], but such implementations rarely provide compile-time checking for record-switch clauses. In the same way that *zoo-match* reports a syntax error when a clause has the wrong number of variables, record-switch should trigger a syntax error when a clause mentions an undefined datatype or lists the wrong number of fields for a datatype.

In this section, we introduce a define-record form and a co-operating record-switch form that detects ill-formed switch clauses and rejects them at compile time. This syntax checking forces a level of communication between the implementations of define-record and record-switch that is characteristic of sophisticated syntactic extensions. At the same time, the implementation of the communication channel exposes common problems in compiling with sophisticated syntactic extensions.

3.1 Record Definition and Dispatch

A typical record-declaration form for Scheme generates a constructor procedure for creating instances of the record, a predicate procedure for recognizing instances of the record, and a field-selector procedure for each field in the record. For our purposes, we choose the following simple syntax:

```
(define-record constructor-name predicate-name
               field-selector-name ...)
```

The ellipses indicate a sequence of *field-selector-name*s, and the number of *field-selector-name*s determines the number of fields in the record (and thus the number of arguments to the constructor procedure).

If we implement define-record in a Record module, we can re-implement Zoo as:

```
(module Zoo MzScheme
  (require Record)
  (provide zebra ——— lizard ———)
  (define-record zebra zebra?
     zebra-weight zebra-stripes)
  (define-record lizard lizard?
     lizard-weight lizard-length lizard-color))
```

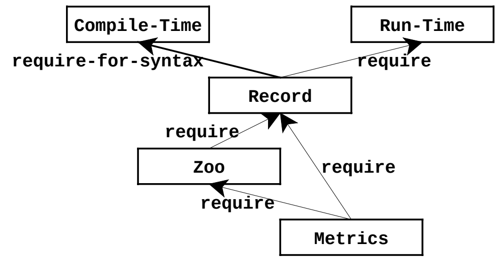

Figure 2. Modules defined in Section 3

Using the record-based predicate and field-accessor procedures, a programmer can define an *animal-weight* function like our original version in Section 2. In many cases, however, a pattern-matching form for record dispatch is especially convenient. Hence, we implement an additional form, record-switch:

```
(record-switch expr
  ((constructor-name local-field-var ...) body-expr)
  ...)
```

where the initial *expr* produces the value to match, each *constructor-name* is the name of a record constructor whose definition is in scope, and one *local-field-var* is provided for each field in the corresponding record type. Each *local-field-var* is bound to its field value within the case's *body-expr*.

If we implement record-switch alongside define-record in Record, we can revise Metrics as follows:

```
(module Metrics MzScheme
  (require Record Zoo)
  (provide animal-weight)
  (define (animal-weight a)
    (record-switch a
      ((zebra w s) w)
      ((lizard w l c) w))))
```

Our key constraint for record-switch concerns error handling. If a programmer writes

```
(define (bad-animal-weight a)
  (record-switch a
    ((zebra w s a b c d e) w) ; too many fields
    ((lizard w l c) w)))
```

then the definition must be rejected as illegal syntax. More generally, if a record-switch expression mentions a record *constructor-name* that has not been defined, or if the number of field variables does not match the number of fields in the definition of *constructor-name*, then record-switch must report an error with a precise diagnosis of the mismatch. Furthermore, we require that the error is reported at compile time, which is before the record-switch expression is evaluated (if ever).

3.2 Implementing Records

The main part of the Record module defines two syntactic transformers using define-syntax:

```
(module Record MzScheme
  ———
  (provide define-record record-switch)
  (define-syntax (define-record stx) ———)
  (define-syntax (record-switch stx) ———)))
```

The following sketch shows the pattern-matching parts of `define-record` and `record-switch`:

```
(module Record MzScheme
  ———
  (define-syntax (define-record stx)
    (syntax-case stx ()
     ((define-record c-name p-name f-name ...)
      (begin ———
       #'(define-values (c-name p-name f-name ...)
           ———)))))
  (define-syntax (record-switch stx)
    (syntax-case stx ()
     ((record-switch expr
        ((c-name f-local-name ...) body)
        other ...)
      (begin ———
       #'(let ((val expr))
           ;; Is val an instance of c-name?
           (if ———
               ;; Yes: evaluate the body.
               (let ((f-local-name ———) ...) body)
               ;; No: try other cases.
               (record-switch val other ...)))))
     ((record-switch expr)
      #'(error "no matching pattern:" expr)))))
```

Using ellipses, the pattern for `define-record` generates a list of *f-name*s when it matches, and the multiple-definition output lists the same sequence of *f-name*s. The pattern for `record-switch` similarly matches a number of local field names for the first switch clause, plus any number of additional clauses; the extra clauses are processed through a recursive use of the macro. Eventually, `record-switch` is used with no clauses (matching the second pattern), and the generated expression reports a failed pattern match if it is reached at run time.[2]

The implementation of `define-record` and `record-switch` requires computation at both compile time and run time. At compile time, `define-record` must store record definitions with field information, and `record-switch` must consult stored information to generate uses of the predicate and field selectors (or to compute an appropriate error message). At run time, a `define-record` form must generate a record type with its constructor, predicate, and selector procedures, and a `record-switch` form must pattern-match records.

To make the separation especially clear, we place the compile-time functions in a `Compile-Time` module, and the run-time support in a `Run-Time` module. The `Compile-Time` module defines a table to hold record-definition information:

```
(module Compile-Time MzScheme
  (provide (all-defined))) ;; Export everything.
  (define table null)
  (define (register-def c-name p-name f-names)
    (set! table (cons (list c-name p-name f-names)
                      table)))
  (define (check-constructor c-name f-names)
    ;; Find c-name in table, and raise a syntax error
    ;; if it's not there or if the field count differs.
    ———)
  (define (constructor->predicate c-name)
    ———) ;; Find c-name in table, return p-name.
  (define (constructor->field-accessors c-name)
    ———)) ;; Find c-name in table, return f-names.
```

[2] An alternative design is to put a set of record definitions together in a named datatype, so that missing clauses can be reported at compile time [10] as in ML.

The `Run-Time` module defines the tag and procedure generators:

```
(module Run-Time MzScheme
  (provide (all-defined))
  (define counter 0)
  (define (generate-unique-tag)
    (set! counter (+ counter 1))
    counter)
  (define (make-record-procs tag f-names)
    ———)) ;; Return multiple procedure values.
```

The `Record` module brings the two together with `require` and `require-for-syntax`:

```
(module Record MzScheme
  (require-for-syntax Compile-Time)
  (require Run-Time)
  (provide define-record record-switch)
  (define-syntax (define-record stx) ———)
  (define-syntax (record-switch stx) ———))
```

Implementing the rest of `Compile-Time` and `Run-Time` is straightforward, so we concentrate on completing the `Record` module.

3.2.1 *First Attempt (Failure)*

Naively, `define-record` might use *register-def* to register a constructor-name mapping before generating the expanded expression:

```
(define-syntax (define-record stx)
  (syntax-case stx ()
   ((define-record c-name p-name f-name ...)
    (begin
     (register-def #'c-name #'p-name #'(f-name ...))
     #'(define-values (c-name p-name f-name ...)
         ———)))))
```

To see why this strategy fails, consider compiling the `Zoo` and `Metrics` modules in separate Scheme sessions. Since `Metrics` imports `Zoo`, `Zoo` must be compiled first. While compiling `Zoo`, *zebra* and *lizard* are added to a table of record definitions, but the compiled uses of `define-record` do not mention *register-def*. Instead, the compile-time table of registrations disappear when the compilation of `Zoo` is complete. Later, when `Metrics` is compiled in a new Scheme session, the table of record registrations is created afresh, and neither *zebra* nor *lizard* is registered.

A key feature of the MzScheme module system is that compiling `Metrics` will fail *even when the modules are compiled in the same session*. Thus, the implementor of the `define-record` macro is alerted to the problem immediately, rather than at some later point where separate compilation (or even separate syntax checking) becomes important.

3.2.2 *Second Attempt (Success)*

To work with MzScheme's module system, `define-record` must permanently attach record registrations to `Zoo` as compile-time information. With the registrations so attached, executing the compile-time portion of `Zoo` for compiling `Metrics` (because `Metrics` imports `Zoo` with `require`) will reinstate the *zebra* and *lizard* registrations.

Macro-generating macros provide `define-record` with a mechanism to attach compile-time information to `Zoo`. If the `define-record`'s macro expansion is a new macro definition, then the new macro definition is attached to `Zoo` as a compile-time expression. Technically, `define-record` can generate a dummy macro definition that calls *register-def* instead of producing a transformer procedure. For readability, we use a `begin-for-syntax` form instead:

```
(define-syntax (define-record stx)
  (syntax-case stx ()
   ((define-record c-name p-name f-name ...)
    #'(begin
       (begin-for-syntax
         ;; Register the record on every compilation:
         (register-def #'c-name #'p-name
                       #'(f-name ...)))
       (define-values (c-name p-name f-name ...)
         (let ((tag (generate-unique-tag)))
          (make-record-procs tag '(f-name ...))))))))
```

The body of a `begin-for-syntax` expression is executed at compile time, just like the right-hand side of `define-syntax`. Consequently, the expansion of `define-record` in the compiled form of `Zoo` will contain a compile-time registration of *zebra*. When `Metrics` is compiled, the import of `Zoo` triggers the execution of `Zoo`'s compile-time expressions, thus registering *zebra*.

Indeed, each individual time that `Metrics` is compiled, the compile-time portions of `Zoo` and `Record` are executed afresh. Since the compile-time portion of `Record` imports `Compile-Time`, then `Compile-Time` is also executed afresh when `Metrics` is compiled. This fresh execution of `Compile-Time` explains why the first attempt at implementing `define-record` triggers a predictable compile-time error. Even when `Zoo` and `Metrics` are compiled in the same Scheme session, they are compiled with different executions of `Compile-Time`, and thus with different record tables.

3.3 Phase Separation

Besides losing a phase-specific calculation too early, as in the first attempt at implementing `define-record`, a programmer might inadvertently mingle compile-time and run-time operations in a macro. For example, the programmer might forget the `begin-for-syntax` wrapper around the use of *register-def*:

```
(define-syntax (define-record stx)
  (syntax-case stx ()
   ((define-record c-name p-name f-name ...)
    #'(begin
       (register-def #'c-name #'p-name
                     #'(f-name ...))
       (define-values (c-name p-name f-name ...)
         (let ((tag (generate-unique-tag)))
          (make-record-procs tag '(f-name ...))))))))
```

In this case, the macro result makes no sense: *register-def* is used in a run-time position, but the only binding of *register-def* refers to a compile-time function. MzScheme flags a syntax error for the resulting expression, because the *register-def* variable is free in the run-time portion of `Record`.

The syntax check is important. The *register-def* function might actually exist at compile time if compilation is interleaved with run time (as in a typical read-eval-print loop). Even in that case, the use of *register-def* must be disallowed, so that interleaved compilation produces the same result as separate compilation.

The detection of an identifier's phase occurs relatively late in the macro-expansion process. For example, in the output of the correct `define-record`, the phase of the *register-def* identifier is determined *after* the output is generated, when it is found to be in `begin-for-syntax`.

In general, the phase of a templated identifier cannot be determined statically from the `#'`-quoted template. For example, we might define a `my-begin-syntax` macro instead of using `begin-for-syntax`:

```
(define-syntax (define-record stx)
  (syntax-case stx ()
   ((define-record c-name p-name f-name ...)
    #'(begin
       (my-begin-syntax
         (register-def #'c-name #'p-name
                       #'(f-name ...)))
       (define-values (c-name p-name f-name ...)
         (let ((tag (generate-unique-tag)))
          (make-record-procs tag '(f-name ...))))))))
```

In this case, the `my-begin-syntax` expression must be expanded to discover that *register-def* is used at compile time. A perverse implementation of `my-begin-syntax` might even dynamically choose to put its body in a compile-time context or a run-time context.

To permit identifier resolution in the proper phase, each identifier must carry *two* versions of its lexical information, one for each phase. This new twist on lexically scoped macros is the key to supporting simple and reliable compilation.

Separating phases begs the question of which phase contains the Scheme implementation's kernel procedures. After all, functions such as `cons` and $+$ are often needed both at compile time and at run time. The answer is that any module (including the one for core Scheme) can exist in multiple phases, but each phase contains a distinct execution of the module. In particular, the `MzScheme` language declaration for `Record` effectively imports core Scheme forms with both `require` and `require-for-syntax`, but the two instantiations of core Scheme are separate; the compile-time `cons` is (in principle) unrelated to the run-time `cons`. More generally, the MzScheme module system allows a module to import a single identifier from two different modules for two different phases.

4 MzScheme Details and Pragmatics

In practice, every module in MzScheme is placed within its own file, and modules refer to each other through relative file paths and library paths. For example, `Zoo` would be placed in a `"zoo.scm"` file, and `Metrics` would import it with `(require "zoo.scm")`. Library paths rely on a mechanism similar to the `CLASSPATH` environment variable that Java implementations use to find libraries.

In a `module` declaration

```
(module module-name language-name
  body-element ...)
```

language-name refers to another module, and the built-in module `MzScheme` is only one possible choice. The syntax and semantics of the *body-element*s are determined by *language-name*. In other words, the module body starts with no syntax or variable bindings, and *language-name* is used as an initial import to introduce bindings for the module body, including bindings for `define`, `provide`, and `require`.

4.1 Definitions, Imports, and Exports

As indicated in Section 2, a `body-element` in the `MzScheme` language is either a definition, an expression, a syntax definition, an import, or an export:

```
body-element ::= (define id expr^0) | expr^0
               | (define-syntax id expr^1)
               | (require req-spec ...)
               | (require-for-syntax req-spec ...)
               | (provide prov-spec ...)
```

The grammar for *expr* extends the standard Scheme grammar [14], including `let-syntax`, which introduces a local macro:

```
expr^n ::= id | (lambda (id ...) expr^n) | (expr^n ...)
         | (syntax-case expr^n ———) | #'template
         | (let-syntax ((id expr^{n+1}) ...) expr^n)
         | ———
```

Within `let-syntax`, the $n+1$ superscript for each binding expression indicates that the expression is evaluated one phase earlier than the `let-syntax` body.

The `require` form imports either all of the bindings of a module, prefixed versions, a subset, a prefixed subset, or renamed bindings:

```
req-spec ::= module-name | (prefix pfx-id module-name)
       | (all-except module-name identifier ...)
       | (prefix-all-except pfx-id module-name id ...)
       | (rename module-name local-id export-id)
```

The `provide` form can export bindings individually (optionally with renaming), and bindings originating from a particular module can be exported as a group:

```
prov-spec ::= id | (rename local-id export-id)
            | (all-from module-name)
            | (all-from-except module-name id ...)
            | (all-defined)
            | (all-defined-except id ...)
```

Unexported module definitions are private to the module.

A module can contain any number of `require`, `provide`, and `require-for-syntax` declarations, in any order. A macro use can expand to `require`, `require-for-syntax`, and `provide` declarations, as well as definitions and expressions. The scope of every imported or defined name covers the entire module body. No name can be multiply defined, and free variables are disallowed.

Since local and imported macros can expand to additional definitions and imports, a module's body is partially expanded to discover all definitions. As a consequence of disallowing multiple definitions for an identifier, a successful partial expansion leads to an unambiguous expansion.

All variables within a module must be bound, whether in a run-time position or in a compile-time position. At run time, modules import and export variables, as opposed to values, which means that assignments to a variable with `set!` are visible outside the module. Imported variables cannot be mutated with `set!`, so if a variable is not mutated within its defining module, it is immutable. This restriction is enforced during compilation, which allows the compiler to perform optimizations based on immutable bindings.[3]

[3]Thanks to Kent Dybvig for recommending this restriction.

4.2 Compilation and Invocation

As a module is compiled, the module itself is visited (i.e., the right-hand $expr^1$ of each `define-syntax` declaration in the module is evaluated immediately).

Since the `require-for-syntax` form triggers an invocation during a syntax-invocation, `require-for-syntax` forces a certain amount of interleaving of compilation and execution. Furthermore, due to the phase-shifting nature of `let-syntax`, macro expansion can involve many concurrent phases of compilation in an arbitrarily tall "tower of expanders".

Nevertheless, the state for each phase is kept separate through lexical scoping and the phase-specific binding of imports. The value of a $expr^n$ variable cannot be accessed by $expr^{n-1}$ code, or vice-versa. Furthermore, invocation of a compiled module does not require any syntax-invocations. In particular, after the main module for a program is compiled, the compiler can strip all compile-time code from the program (i.e., $expr^1$s), including entire modules that are used only through `require-for-syntax`.

4.3 Syntax Primitives

A syntax object is a first-class value, and syntax objects can exist at run time as well as compile time, but they are used primarily at compile time. Built-in operations support the deconstruction of a syntax object, the composition of new syntax objects from old ones, and the comparison of binding properties for two identifier syntax objects (e.g., determining whether they refer to the same lexical binding). The `syntax-case` form in MzScheme expands to an expression that uses the built-in operations to deconstruct and pattern-match syntax objects.

The `quote-syntax` primitive form is similar to `quote`, except that it generates syntax-object constants instead of lists and symbols. The `#'` template form expands to an expression that uses `quote-syntax` on the portions of the template that do not refer to pattern variables. Meanwhile, `syntax-case` communicates pattern-variable bindings to `#'` in roughly the same way that `define-record` communicates to `record-switch`.

5 A Model of Compilation

Our formal model of MzScheme's macro and module system builds on Dybvig et al.'s model [8]. Here, we provide a sketch of the model and its key results, which demonstrate various separation properties.

The model is a simplification of MzScheme in several ways. First, every module is implemented in a fixed base language. Second, modules export all definitions, and no renaming is allowed on export or import. Third, the order of declarations in a module body is fixed (`require-for-syntax` declarations are first, etc.), and macro applications cannot expand to imports or definitions. Despite these simplifications, the model includes both `require-for-syntax` and `let-syntax`, so that the model covers phase-sensitive lexical scope, macro-defining macros, and interleaved execution of phases.

Source Grammar. A source program consists of a sequence of `module` declarations followed by a single (`invoke` *mod id*). The final `invoke` declaration triggers the execution of the module *mod*, and extracts the computed value for that module's *id* variable.

prog	::=	*decl* ... (`invoke` *mod id*)
decl	::=	(`module` *mod*
		(`require-for-syntax` *mod*) ...
		(`require` *mod*) ...
		(`define-syntax` *id s-exp*) ...
		(`define` *id s-exp*) ...)
s-exp	::=	*stx* | *prim* | (*s-exp* ...)
stx	::=	an identifier with lexical info (see Figure 3)
prim	::=	a primitive value or operator
id	::=	an identifier
mod	::=	a module name

Each module declaration contains a sequence of for-syntax imports, a sequence of normal imports, a sequence of syntax definitions, and a sequence of normal definitions. The expressions in definitions are arbitrary syntax objects, represented by the *s-exp* non-terminal, at least until they are parsed.

Core Language Expressions. Parsing and macro expansion are intertwined, so that *s-exp* is as much as we can write for a true grammar of source expression. In the absence of macros and ignoring shadowing, however, the core grammar of expressions is as follows:

base-s-exp	::=	(`app` *base-s-exp base-s-exp* ...)
	|	(`lambda` (*id*) *base-s-exp*)
	|	(`let-syntax` (*id base-s-exp*) *base-s-exp*)
	|	(`macro-app` *id base-s-exp* ...)
	|	(`quote-syntax` *s-exp*)
	|	*prim* | *id*

This core source language consists of function applications (written with an explicit `app`), functions, local macro definitions, macro uses (written with an explicit `macro-app`), quoted literals, primitives, and variable references. The `app`, `lambda`, etc. names are not keywords; they are merely “bound” in the initial environment to mean the primitive application form, the primitive function form, etc., respectively.

Executable Grammar. Parsing and compiling an input *s-exp* produces an executable *c-exp*. Compiling a sequence of source `module` declarations produces a sequence of compiled `cmodule` declarations:

cprog	::=	*cdecl* ... (`invoke` *mod id*)
cdecl	::=	(`cmodule` *mod*
		(`require-for-syntax` *mod*) ...
		(`require` *mod*) ...
		(`define-syntax` *id c-exp*) ...
		(`define` *id c-exp*) ...)
c-exp	::=	(`app` *c-exp c-exp* ...) | *id* | *mod.id.p* | *val*
val	::=	(`lambda` (*id*) *c-exp*) | (`lit` *s-exp*)

Our target language thus consists of functions, function applications, lexical variable references, module variable references *mod.id.p* (i.e., for a certain module, variable, and phase), and literal constants. Constants encapsulate lexical-context information, which is useful when the constant is used in a macro implementation.

The evaluation of *c-exp*s is defined in the usual manner, with rules for primitives such as

$$\langle(\texttt{app}\ (\texttt{lit}\ \textsf{car})\ (\texttt{lit}\ (\textit{s-exp}_0\ \textit{s-exp}_1 \ldots\ \textit{s-exp}_n))), S\rangle$$
$$\longrightarrow \langle(\texttt{lit}\ \textit{s-exp}_0), S\rangle$$

where S is the store. Primitives can consult, extend, or modify the store. Invoking or visiting a module also extends the store. Evaluating a variable reference *mod.id.p* accesses a module-installed binding in the store.

Module Compilation. The compile-module function compiles an entire source module, given a sequence of previously compiled modules that are available for import:

$$\textsf{compile-module} : \textit{decl} \times \textit{cdecl-list} \rightarrow \textit{cdecl}$$

The module-compilation function does not consume or produce a store. Instead, it starts from an empty store, reflecting the separate compilation of separate modules, and the separation of compile-time state from run-time state.

Using the fresh store, compile-module visits `required` modules and updates the store with imported bindings. The compile-module function also invokes `require-for-syntax`ed modules.

After visiting and invoking imported modules, compile-module annotates the *s-exp*s in the body of the module to record the imports and definitions of the module. The annotation includes an appropriate phase: 0 for local definitions and `require` imports, 1 for `require-for-syntax` imports. Next, expressions are compiled from the right-hand side of all `define-syntax` declarations using compile-expr (defined below) with phase 1; if any macro uses state-modifying primitives, the store is updated in the process. The store is then updated with the resulting syntax-transformer bindings, and all expressions from right-hand side of `define` declarations are compiled using compile-expr with phase 0. Finally, both sets of compiled expressions are collected into a compiled module.

Expression Compilation. An expression is parsed, expanded, and compiled at once with a recursive compile-expr function:

$$\textsf{compile-expr} : \textit{s-exp} \times p \times E \times S \rightarrow \textit{c-exp} \times S$$

This function compiles the source expression *s-exp* for execution in phase p. The environment E maps identifiers to locally bound syntax transformers, and the store S contains compile-time state, as well as bindings for invoked and visited modules (e.g., bindings for imported syntax). The result of compilation is a pair consisting of a compiled expression and an updated store.

Figure 3 defines $[\![\textit{s-exp}]\!]^p_{E,S}$, which is shorthand for applying compile-expr to *s-exp*, p, E, and S. The result is an expression–store pair $\langle \textit{c-exp}, S\rangle$. In the process of parsing an *s-exp*, compile-expr adds `mark` and `subst` annotations to maintain lexical scope. A `mark` annotation effectively records whether a binding was introduced by a macro, so that it does not accidentally capture variables at the macro-use site. A `subst` annotation effectively α-renames an identifier, so that variables introduced by a macro are not accidentally captured at the macro-use site. (The original source must have no such annotations.) For more information about `mark` and `subst`, see Dybvig et al. [8].

Parsing does not add new `reqd` annotations. Instead, the compile-module function (defined above) records module bindings with `reqd` annotations before passing body expressions to compile-expr.

The main step in compiling an expression ($\textit{stx}_0$ $\textit{s-exp}_1$ ... $\textit{s-exp}_n$) is to determine the meaning of $\textit{stx}_0$ based on its lexical information, the environment, the store, and the current phase. For example, if $\textit{stx}_0$ resolves to the free symbol `lambda`, then the expression is compiled as a function. If $\textit{stx}_0$ resolves to an identifier bound to a macro transformer, then the transformer function is applied to (`lit` ($\textit{stx}_0$ $\textit{s-exp}_1$ ... $\textit{s-exp}_n$)) to produce a new *s-exp* and updated

Syntax objects:

$$
\begin{array}{lcl}
\mathit{stx} & ::= & \mathit{id} \mid (\texttt{mark}\ \mathit{stx}\ \mathit{mrk}) \mid (\texttt{subst}\ \mathit{stx}\ \mathit{stx}\ \mathit{id}\ p) \mid (\texttt{reqd}\ \mathit{stx}\ \mathit{mod}\ \mathit{id}\ p) \\
\mathit{mrk} & ::= & \text{a mark} \\
p & ::= & \text{a phase number}
\end{array}
$$

The compile-expr function:

$$
\begin{array}{lcl}
[\![(\mathit{stx}_0\ \mathit{s\text{-}exp}_0\ \mathit{s\text{-}exp}_1 \ldots\ \mathit{s\text{-}exp}_n)]\!]^p_{E,S_0} & = & \langle(\texttt{app}\ \mathit{c\text{-}exp}_0\ \ldots\ \mathit{c\text{-}exp}_n), S_{n+1}\rangle \\
\quad \text{if } \mathbf{resolve}^p(\mathit{stx}_0) = \langle\texttt{app},\texttt{free}\rangle & & \quad \text{where } \langle \mathit{c\text{-}exp}_i, S_{i+1}\rangle = [\![\mathit{s\text{-}exp}_i]\!]^p_{E,S_i} \\
\\
[\![(\mathit{stx}_0\ (\mathit{stx})\ \mathit{s\text{-}exp})]\!]^p_{E,S} & = & \langle(\texttt{lambda}\ (\mathit{id})\ \mathit{c\text{-}exp}), S'\rangle \\
\quad \text{if } \mathbf{resolve}^p(\mathit{stx}_0) = \langle\texttt{lambda},\texttt{free}\rangle & & \quad \text{where } \mathit{s\text{-}exp}' = \mathbf{subst}(\mathit{s\text{-}exp}, \mathit{stx}, \mathit{id}, p) \\
 & & \quad \text{and } \langle \mathit{c\text{-}exp}, S'\rangle = [\![\mathit{s\text{-}exp}']\!]^p_{E,S} \\
 & & \quad \text{and } \mathit{id} \text{ is fresh} \\
\\
[\![(\mathit{stx}_0\ (\mathit{stx}\ \mathit{s\text{-}exp}_1)\ \mathit{s\text{-}exp}_2)]\!]^p_{E,S} & = & [\![\mathit{s\text{-}exp}'_2]\!]^p_{E\cup\{\mathit{id}=\mathit{val}\},S''} \\
\quad \text{if } \mathbf{resolve}^p(\mathit{stx}_0) = \langle\texttt{let-syntax},\texttt{free}\rangle & & \quad \text{where } \langle \mathit{c\text{-}exp}'_1, S'\rangle = [\![\mathit{s\text{-}exp}_1]\!]^{p+1}_{\emptyset,S} \\
 & & \quad \text{and } \langle \mathit{val}, S''\rangle = \mathbf{eval}(\mathit{c\text{-}exp}'_1, S') \\
 & & \quad \text{and } \mathit{s\text{-}exp}'_2 = \mathbf{subst}(\mathit{s\text{-}exp}_2, \mathit{stx}, \mathit{id}, p) \\
 & & \quad \text{and } \mathit{id} \text{ is fresh} \\
\\
[\![(\mathit{stx}_0\ \mathit{stx}\ \mathit{s\text{-}exp}_0)]\!]^p_{E,S} & = & [\![\mathit{s\text{-}exp}_3]\!]^p_{E,S'} \\
\quad \text{if } \mathbf{resolve}^p(\mathit{stx}_0) = \langle\texttt{macro-app},\texttt{free}\rangle & & \quad \text{where } \mathit{s\text{-}exp}_1 = \mathbf{mark}(\mathit{s\text{-}exp}_0, \mathit{mrk}) \\
\quad \text{and } (\mathbf{resolve}^p(\mathit{stx}) = \langle \mathit{id}, \texttt{lexical}^p\rangle & & \quad \text{and } \langle(\texttt{lit}\ \mathit{s\text{-}exp}_2), S'\rangle = \\
\qquad \text{and } E(\mathit{id}) = \mathit{val}) & & \qquad \mathbf{eval}(\ (\texttt{app}\ \mathit{val}\ (\texttt{lit}\ \mathit{s\text{-}exp}_1)), S) \\
\qquad \text{or } (\mathbf{resolve}^p(\mathit{stx}) = \langle \mathit{mod.id}, \texttt{module}\rangle & & \quad \text{and } \mathit{s\text{-}exp}_3 = \mathbf{mark}(\mathit{s\text{-}exp}_2, \mathit{mrk}) \\
\qquad\quad \text{and } S(\mathit{mod.id.p}) = \langle \mathit{val}, \texttt{macro}\rangle) & & \quad \text{and } \mathit{mrk} \text{ is fresh} \\
\\
[\![(\mathit{stx}_0\ \mathit{s\text{-}exp})]\!]^p_{E,S} & = & \langle(\texttt{lit}\ \mathit{s\text{-}exp}), S\rangle \\
\quad \text{if } \mathbf{resolve}^p(\mathit{stx}_0) = \langle\texttt{quote-syntax},\texttt{free}\rangle & & \\
\\
[\![\mathit{stx}]\!]^p_{E,S} & = & \langle \mathit{id}, S\rangle \\
\quad \text{if } \mathbf{resolve}^p(\mathit{stx}) = \langle \mathit{id}, \texttt{lexical}^p\rangle & & \\
\quad \text{and } \mathit{id} \notin \mathrm{dom}(E) & & \\
\\
[\![\mathit{stx}]\!]^p_{E,S} & = & \langle \mathit{mod.id.p}, S\rangle \\
\quad \text{if } \mathbf{resolve}^p(\mathit{stx}) = \langle \mathit{mod.id}, \texttt{module}\rangle & & \\
\quad \text{and } S(\mathit{mod.id.p}) \neq \langle \mathit{val}, \texttt{macro}\rangle & & \\
\\
[\![\mathit{prim}]\!]^p_{E,S} & = & \langle(\texttt{lit}\ \mathit{prim}), S\rangle
\end{array}
$$

Recording substitutions and marks:

$$
\begin{array}{lcll}
\mathbf{subst}(\mathit{stx}_1, \mathit{stx}_2, \mathit{id}, p) & = & (\texttt{subst}\ \mathit{stx}_1\ \mathit{stx}_2\ \mathit{id}\ p) & \\
\mathbf{subst}(\mathit{prim}, \mathit{stx}_2, \mathit{id}, p) & = & \mathit{prim} & \\
\mathbf{subst}(\ (\mathit{stx}_1 \ldots\ \mathit{stx}_n), \mathit{stx}, \mathit{id}, p) & = & (\mathit{stx}'_1 \ldots\ \mathit{stx}'_n) & \text{where } \mathit{stx}'_i = \mathbf{subst}(\mathit{stx}_i, \mathit{stx}, \mathit{id}, p) \text{ for } i \in [1,n] \\
\\
\mathbf{mark}(\mathit{stx}, \mathit{mrk}) & = & (\texttt{mark}\ \mathit{stx}\ \mathit{mrk}) & \\
\ldots & & &
\end{array}
$$

Identifier resolution:

$$
\begin{array}{lcl}
\mathbf{resolve}^p(\mathit{id}) & = & \langle \mathit{id}, \texttt{free}\rangle \\
\mathbf{resolve}^p(\ (\texttt{mark}\ \mathit{stx}\ \mathit{mrk})\ ) & = & \mathbf{resolve}^p(\mathit{stx}) \\
\mathbf{resolve}^p(\ (\texttt{subst}\ \mathit{stx}_1\ \mathit{stx}_2\ \mathit{id}\ p')\ ) & = & \begin{cases} \langle \mathit{id}, \texttt{lexical}^{p'}\rangle & \text{if } \mathbf{marksof}(\mathit{stx}_1) = \mathbf{marksof}(\mathit{stx}_2) \\ & \text{and } \mathbf{resolve}^0(\mathit{stx}_1) = \mathbf{resolve}^0(\mathit{stx}_2) \\ \mathbf{resolve}^p(\mathit{stx}_1) & \text{otherwise} \end{cases} \\
\mathbf{resolve}^p(\ (\texttt{reqd}\ \mathit{stx}\ \mathit{mod}\ \mathit{id}\ p')\ ) & = & \begin{cases} \langle \mathit{mod.id}, \texttt{module}\rangle & \text{if } \mathbf{resolve}^p(\mathit{stx}) = \langle \mathit{id}, \texttt{free}\rangle \\ & \text{and } p = p' \\ \mathbf{resolve}^p(\mathit{stx}) & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcll}
\mathbf{marksof}(\mathit{id}) & = & \emptyset & \\
\mathbf{marksof}(\ (\texttt{mark}\ \mathit{stx}\ \mathit{mrk})\ ) & = & \{\mathit{mrk}\} \uplus \mathbf{marksof}(\mathit{stx}) & \text{where } \uplus \text{ is exclusive union} \\
\mathbf{marksof}(\ (\texttt{subst}\ \mathit{stx}_1\ \mathit{stx}_2\ \mathit{id}\ p)\ ) & = & \mathbf{marksof}(\mathit{stx}_1) & \\
\mathbf{marksof}(\ (\texttt{reqd}\ \mathit{stx}\ \mathit{mod}\ \mathit{id}\ p)\ ) & = & \mathbf{marksof}(\mathit{stx}) &
\end{array}
$$

Figure 3. Expression parsing, expansion, and compilation

store; the new *s-exp* and store are sent back into the compile-expr function. If stx_0 resolves to the free symbol `let-syntax`, then a sub-expression is sent to compile-expr with phase $p+1$, the result is bound in E, and the body sub-expression is compiled with the new environment in phase p.

Module Invocation. All modules are compiled as if they will be invoked in phase 0 (the phase shows up in literals), but a `require-for-syntax`ed module must be invoked in phase 1, a `require-for-syntax`ed module of a `require-for-syntax`ed module must be invoked in phase 2, and so on. Thus, invocation requires a phase-shifting operation on compiled expressions; $\langle\langle c\text{-}exp \rangle\rangle_p$ shifts *c-exp* by p phases.

The visit function augments a store by executing the syntax portion of a module for some phase p, given the collection of compiled modules so far:

$$\mathsf{visit} \; : \; mod \times p \times cdecl\text{-}list \times S \rightarrow S$$

Every `require` import in *mod* triggers a recursive visit in phase p. Every `require-for-syntax` import in *mode* triggers an invoke in phase $p+1$, as well as a recursive visit in phase $p+1$. Finally, each phase-1 expression in *mod* is shifted by p and evaluated, and the store is updated with syntax bindings that name the module, the defined identifier, and the phase p.

The invoke function performs the corresponding action for the run-time part of a module:

$$\mathsf{invoke} \; : \; mod \times p \times cdecl\text{-}list \times S \rightarrow S$$

Every `require` import in *mod* triggers a recursive invoke in phase p. Afterwards, each phase-0 expression in *mod* is shifted by p and evaluated, and store is updated with variable bindings that name the module, the defined identifier, and the phase p. For invoke, `require-for-syntax` imports are ignored, and visit is never used.

Program Execution. Executing a program means first compiling each of the program's modules, one by one, with compile-module. For each compilation, modules already compiled are available as imports. After compiling all modules, the main module designated by (`invoke` *mod id*) is executed with invoke in a fresh initial store. The result of the the program is the value of *mod.id.0* in the store.

Formal Results. The formal model makes certain separation properties immediately apparent:

1. State modifications during module compilations do not affect each other or the final execution, since module compilation neither consumes nor produces a store.
2. All phase 1 code can be stripped before execution of the designated main module with no effect on the result, since applying invoke with phase 0 executes only phase 0 code.

6 Related Work

Lexically scoped macros. Kohlbecker et al.'s definition of *hygienic macros* [15] initiated a chain of research in Scheme macros, leading to the `syntax-case` system of Dybvig et al. [8]. Notable points along the way include Bawden and Rees's syntactic closures [2] and Clinger and Rees's lexically scoped, pattern-matching macros [4].

Our work builds directly on the `syntax-case` model. In the original model, a local phase separation exists via `let-syntax`, though the model does not explain how out-of-phase errors are detected and reported. Our model fills this small gap while generalizing the model to cover module phases.

Lexical macro systems are not restricted to Lisp dialects. For example, Maya [1] extends Java with support for lexically scoped syntax transformers. Maya transformers are implemented in Maya, which means that they can perform arbitrary computation, and that they can be implemented in an extended variant of Maya. Macro-generating macros are limited, however, by the separation of transformer definition (as a normal Java class) from transformer use (through a `use` clause names an already-compiled class) to achieve a phase separation.

Module systems. Curtis and Rauen's module system for Scheme [5] allows modules to export both variables and syntax, but syntax transformers must be implemented in plain Scheme. Syntax transformers may keep state, and the restrictions on such state (in terms of what is guaranteed to work) seem to match ours, but Curtis and Rauen provide no information on how to enforce the restrictions.

The Scheme48 module system [20] supports the compile-time import of variables for macro transformers by wrapping an import declaration with `for-syntax`; such compile-time imports bind only compile-time code within the module. However, templated identifiers in macros appear to be statically assigned a run-time status, which causes problems for macro-defining macros that are defined within a module. Furthermore, a module is instantiated only once within a session, even if it is used in multiple phases or for compiling multiple modules in the session, which means that state can be preserved accidentally across module compilations.

Dybvig and Waddell [25] integrate lexically scoped macros with a `module` construct for Chez Scheme [7], but they do not distinguish phases for module imports; programmers must manage the difference between compilation and interactive evaluation with `load`, `visit`, and `eval-when`. Unlike MzScheme's `module` form, the Chez `module` form works in any definition position. (It can be implemented as a macro in MzScheme, except for the `import-only` form that hides lexical bindings.)

Dylan [22] provides pattern-matching macros that respect module scope, but macros cannot perform arbitrary computation.

Organizing language towers. Queinnec [19] defines a protocol for macro expansion that supports a tower of languages. The protocol is independent of the macro-definition language and expansion function. MzScheme essentially automates the protocol through the module language, while integrating lexically scoped macros into the tower.

Other Work. Staged evaluation languages, such as $\lambda^{\Box}$ [6] and MetaML [16], support programs that generate and combine program fragments, much like a macro transformer. Such program-manipulating programs serve a different purpose than macros, because they do not extend the syntax of a language processed by compilers and other programming tools. Staged evaluation can be a platform for constructing macro systems, however, as exemplified by the compilation of MacroML [11] to MetaML.

Languages that support dynamic compilation, such as 'C [18], are similar to staged-evaluation languages, but that they have no phase distinction. Dynamically generated and compiled code is meant to be executed along with the program-manipulating host code.

7 Conclusion

A language that allows macro transformers to perform arbitrary computation must enforce a separation between computations: run time versus compile time, as well as the compile time of one module versus the compile time of another. Without an enforced separation, the meaning of a code fragment can depend on the order in which code is compiled and executed. At best, programmers must work hard to manage the dependencies. At worst, and more commonly, the dependencies are too subtle for programmers to manage correctly, and they cannot expect predictable results when combining libraries in new ways or when using new programming tools.

The MzScheme macro system enforces the separation of run-time and compile-time computations. This enforcement does not restrict the kinds of macros that can be implemented. Instead, MzScheme enables the implementation of sophisticated, cooperating syntactic extensions through well-defined channels of communication. We have demonstrated this expressiveness through a small `define-record` and `record-case` example, and the same techniques apply for implementing other constructs: classes for object-oriented programming, component definition and linking constructs, `lex` and `yacc` forms, and forms for static typing.

From the Scheme programmer's perspective, MzScheme modules and macros work in the obvious way for most tasks. Indeed, users report a short learning curve for putting `module` to work. More complex tasks require careful reasoning, and future work remains in providing precise and clear feedback for phase violations. Most important, however, is that phase violations never pass undetected. In practical terms, this means that extension producers can be confident of their extensions, and extension consumers spend no time wrestling with command-line flags or configuration parameters.

Acknowledgments: I would like to thank Robby Findler, Matthias Felleisen, and the rest of PLT for the discussions that launched and refined this work. Thanks to Mike Sperber for many module-related discussions. Thanks to Wilson Hsieh, Eric Eide, Alastair Reid, and Jason Baker for feedback on the paper.

8 References

[1] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 270–281, June 2002.

[2] A. Bawden and J. Rees. Syntactic closures. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 86–95, July 1988.

[3] H. Bromley. *Lisp Lore: A Guide to Programming the Lisp Machine*. Kluwer Academic Publishers, 1986.

[4] W. Clinger and J. Rees. Macros that work. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 155–162, Jan. 1991.

[5] P. Curtis and J. Rauen. A module system for Scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 13–28, 1990.

[6] R. Davies and F. Pfenning. A modal analysis of staged computation. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 258–270, Jan. 1996.

[7] R. K. Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998.

[8] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[9] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

[10] D. P. Friedman, M. Wand, and C. P. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2001.

[11] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proc. ACM International Conference on Functional Programming*, pages 74–85, Sept. 2001.

[12] T. P. Hart and T. G. Evans. Notes on implementing lisp for the m-460 computer. In E. C. Berkeley and D. G. Bowbrow, editors, *The Programming language LISP: Its Operation and Applications*, pages 191–203. MIT Press, 1964.

[13] R. Kelsey. SRFI 19: Defining record types, 1999. `http://srfi.schemers.org/srfi-9/`.

[14] R. Kelsey, W. Clinger, and J. Rees (Eds.). The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), Sept. 1998.

[15] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 151–181, Aug. 1986.

[16] E. Moggi, W. Taha, Z. El-Abidine Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *Proc. European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, Apr. 1999.

[17] D. A. Moon. *MACLISP Reference Manual*. MIT Press, 1974.

[18] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Computing Systems*, 21(2):324–369, Mar. 1999.

[19] C. Queinnec. Macroexpansion reflective tower. In *Proc. Reflections Conference*, pages 93–194, Apr. 1996.

[20] J. Rees. Another module system for Scheme, 1994. Scheme48 documentation.

[21] M. Serrano. *Bigloo: A "practical Scheme compiler"*, 2002.

[22] A. Shalit. *The Dylan Reference Manual*. Addison-Wesley, 1996.

[23] G. L. Steele Jr. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[24] G. L. Steele, Jr. and G. J. Sussman. The revised report on SCHEME: A dialect of LISP. Technical Report 452, MIT, Jan. 1978.

[25] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 203–213, Jan. 1999.

[26] D. Weise and R. F. Crew. Programmable syntax macros. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.

[27] A. K. Wright. *Pattern Matching for Scheme*, Mar. 1996.