Syntax Templates in Racket

RYAN CULPEPPER, Czech Technical University in Prague

One foundation of Scheme macro systems is the notation of syntax *patterns* and *templates*. Patterns are used to match and destructure terms, and templates are used to construct new terms. This paper presents extensions to Racket's template notation to support template splicing, conditional generation, and template metafunctions. The new template features complement `syntax-parse`'s previous extensions to the pattern-matching notation.

1 INTRODUCTION

Macro-by-Example (Kohlbecker and Wand 1987) is the foundation of the major Scheme macro notations, such as `syntax-rules` (Kelsey et al. 1998) and `syntax-case` (Dybvig et al. 1992; Sperber et al. 2009b). MBE specifies a pattern-based notation for defining macro transformations. A macro has a list of clauses, each clause consisting of an input *pattern* and an output *template*.[1] When the expander encounters a use of a macro, it tries each clause in order, and if the use matches an input pattern, code is emitted by instantiating the corresponding template with the pattern's variables bound to the appropriate subterms of the macro occurrence. For example, it allows the transformation of `let` into the immediate application of a `lambda` to be expressed thus:[2]

```
(define-syntax-rule (let ((i e) ...) b ...)
   ((lambda (i ...) b ...) e ...))
```

instead of thus:

```
(defmacro let (decls . body)
  `((lambda ,(map car decls) . ,body)
    . ,(map cadr decls)))
```

In the pattern and template above, the ellipsis (`...`) indicates repetition. In a pattern, it matches a sequence of terms, each of which must match the preceding subpattern, and the subpattern's variables' values are collected together. For example, if the `let` macro above is used as `(let ([x 1] [y 2] [z 3]) (+ x (* y z)))`, the pattern variable `i` is bound to the sequence of terms `x`, `y`, and `z`, and the pattern variable `e` is bound to the sequence of terms `1`, `2`, and `3`. In a template, an ellipsis produces a sequence of terms, each of which is constructed by instantiating the preceding subtemplate by implicitly `map`ping the subtemplate over the corresponding variables' values. Thus the template for `let` would be instantiated as `((lambda (x y z) (+ x (* y z))) 1 2 3)`.

Over time, the template language has accumulated extensions such as ellipsis escaping and multiple-ellipsis flattening (Dybvig 2003; Dybvig and Waddell 2016), and the representation of Scheme terms has changed from simple S-expressions to *syntax objects*, requiring the reinterpretation of MBE for the new term representation.

Around 2007, Racket added *keywords* (Flatt 2007), and since then it has become more common for Racket syntax to include non-parenthesized logical components, usually introduced or delimited by keywords. For example, here is a structure declaration:

```
(struct point (x y) #:property prop:custom-write print-point #:transparent)
```

[1]The original paper called these "input patterns" and "output patterns." The Scheme reports, starting with Clinger and Jonathan Rees (eds.) (1991), use the "pattern" and "template" terminology.

[2]Examples in this paper use Racket's macro-definition forms, including `define-syntax-rule` and `syntax-parse`.

Author's address: Ryan Culpepper Czech Technical University in Prague, ryanc@racket-lang.org.

The `#:property prop:custom-write print-point` clause attaches a struct type property that controls printing; the `#:transparent` clause affects equality and hashing.

As another example, here is some code for processing command-line options and arguments, adapted from the implementation of the `raco scribble` command (___ represents omitted code):

```
(command-line #:program (short-program+command-name)
  #:once-any
  [("--html") "generate HTML-format output file (the default)" ___]
  [("--pdf") "generate PDF-format output (via PDFLaTeX)" ___]
  #:once-each
  [("--dest") dir "write output in <dir>" ___]
  #:multi
  [("++style") file "add given .css/.tex file after others" ___]
  #:args (file) ___)
```

The `#:once-any`, `#:once-each`, and `#:multi` keywords control how subsequent command-line flags are used.

The implicit grouping is generally obvious to programmers. But it's cumbersome to parse using `syntax-case`.

Consequently, `syntax-parse` (Culpepper 2012) introduces an extended notion of patterns designed to accommodate syntaxes like the ones above. For example, a `struct` property clause can be described with the following pattern:

```
(~seq #:property prop:expr value:expr)
```

This kind of pattern is called a *head pattern*, because instead of matching a single term, it matches a sequence of terms from the head of a syntax list.

In the pattern above, `~seq` forms a head pattern that matches a sequence containing the following subpatterns, `#:property` is a keyword literal, and `:expr` is a *syntax class annotation* that says the preceding pattern variable matches only an expression (really, anything except a keyword). A syntax class is like a nonterminal for the pattern language; it encapsulates a pattern together with optional side condition checks, attribute computations, etc.

Similarly, the syntax of `command-line` can be described by the following pattern:

```
(_ n:maybe-name a:maybe-argv clause:flag-clause ... final:final-clause)
```

with some auxiliary syntax classes such as

```
(define-splicing-syntax-class maybe-name
  (pattern (~optional (~seq #:program name:expr))))
(define-splicing-syntax-class flag-clause
  (pattern (~seq #:once-any e:flag-entry ...) #:attr kind 'once-any)
  ___)
```

and so on. Splicing syntax classes encapsulate head patterns, and thus match sequence of terms, in contrast to normal syntax classes, which encapsulate normal (single-term) patterns. An `~optional` pattern tries to match its subpattern, but if that fails it matches an empty sequence of terms and the subpattern's variables are considered "absent". An `#:attr` clause creates an attribute whose value is computed when an instance of the syntax class is parsed, and the user of a syntax class can inspect its attributes. Pattern variables are essentially attributes that contain syntax objects (or lists of syntax objects for pattern variables in front of ellipses, and so on), but attributes can also carry compile-time AST structures, procedures to check context-sensitive constraints, and more.

The extended patterns supported by `syntax-parse` have proven useful. But they created an imbalance of expressiveness between patterns and templates. In particular, the examples above raise the following questions:

- How can templates produce non-parenthesized groups of terms?
- How can templates gracefully process "absent" pattern variables?
- How can computation be embedded in template instantiation, similar to how syntax classes and attributes allow embedding computation in pattern matching? (Specifically, `quasisyntax` is not sufficient, because it does not cooperate with ellipses.)

Each of these questions above motivates a new template feature. Section 2 introduces the new template features through examples. Section 3 gives an updated grammar of templates, and Section 4 describes the meaning and implementation of the extended template language.

2 NEW TEMPLATE FEATURES

This section introduces three new template features and gives examples that show how they interact with existing template features (mainly ellipses) and also how they interact with each other.

2.1 Splicing

The first extension allows a template to produce a non-parenthesized sequence of terms. Of course, it's trivial to produce, say, three non-parenthesized terms by writing a non-parenthesized group of three templates. But when a sequence of such groups must be produced, a more specialized feature is needed.

Consider Racket's `hash` function. It constructs a hash table from an argument list that contains alternating keys and values. For example, `(hash 'a 1 'b 2 'c 3)` has keys `'a`, `'b`, and `'c` mapped to values `1`, `2`, and `3`, respectively.

Suppose we want to write a macro `kw-hash` that takes keys in keyword form and produces a hash table with symbol keys. For simplicity, we'll do the actual conversion of keywords to symbols at run time. For example:

```
(kw-hash #:a 1 #:b 2 #:c 3)
⇒ (hash (keyword->symbol '#:a) 1 (keyword->symbol '#:b) 2 (keyword->symbol '#:c) 3)
```

The macro pattern can be written as follows:

```
(_ (~seq key:kw value:expr) ...)
```

But how can we construct a sequence of alternating (transformed) key and value expressions?

Previously, it required escaping to Racket. One way is to do the list manipulation by hand:

```
(define-syntax (kw-hash stx)
  (syntax-parse stx
    [(_ (~seq key:kw value:expr) ...)
     (with-syntax ([(arg ...)
                    (apply append
                           (map list
                                (syntax->list #'((keyword->symbol (quote key)) ...))
                                (syntax->list #'(value ...))))])
       #'(hash arg ...))]))
```

Another is to use multiple-ellipsis flattening, a common extension to MBE, where a template followed by multiple ellipses concatenates the results of the inner `map`s together without parentheses. For example:

```
(define-syntax (kw-hash stx)
  (syntax-parse stx
    [(_ (~seq key:kw value:expr) ...)
     (with-syntax ([([arg ...] ...) #'([(keyword->symbol (quote key)) value] ...)])
```

```
      #'(hash arg ... ...))]))
```

Racket's new template splicing form, written `~@`, eliminates the need for the auxiliary `with-syntax` binding and artificial extra ellipsis:

```
(define-syntax (kw-hash stx)
  (syntax-parse stx
    [(_ (~seq key:kw value:expr) ...)
     #'(hash (~@ (keyword->symbol (quote key)) value) ...)]))
```

Note: the splicing form generalizes multiple-ellipsis flattening in the following sense: `(T ... ...)` is equivalent to `((~@ T ...) ...)`.

The general form of a `~@` template is `(~@ . template)`; that is, the tail of the `~@`-template is interpreted as a subtemplate. The subtemplate must result in a proper syntax list (otherwise an error is signaled), and the contents are spliced into the enclosing template's instantiation. For example:

```
(with-syntax ([name #'point]
              [(fld ...) #'(x y)]
              [(clause ...)
               #'((#:property prop:custom-write print-point)
                  (#:transparent))])
  #'(struct name (fld ...) (~@ . clause) ...))
⇒ #'(struct point (x y) #:property prop:custom-write print-point #:transparent)
```

This allows, for examples, syntax classes to compute clauses by treating them as parenthesized terms; the macro can splice them in the template.

2.2 Try/Catch

The second extension provides better handling for absent pattern variable values.

The `syntax-parse` system provides the `attribute` form for directly accessing the value of an attribute. A pattern variable is just an attribute that normally contains a syntax object (or list thereof, etc). An absent pattern variable contains the value `#f` instead, and attempting to instantiate a template with an absent pattern variable raises an error. (Note that in Racket `#f` is distinct from a syntax object containing the value `#f`.)

Thus one way to handle absent pattern variables is to insert explicit checks using `attribute`:

```
(define-syntax (command-line stx)
  (syntax-parse stx
    [(_ n:maybe-name ___)
     (with-syntax ([name (or (attribute n.name) #'"unknown program")])
       #'(___ name ___))]))
```

But this is cumbersome and—like all solutions outside of the template system—it interacts poorly with ellipses.

Another approach is to change the pattern so that the attribute is always defined. For example, we could define `maybe-name` thus:

```
(define-splicing-syntax-class maybe-name
  (pattern (~seq #:program name:expr))
  (pattern (~seq) #:with name #'"unknown program"))
```

The same effect can be achieved with `~optional` by specifying defaults:

```
(~optional (~seq #:program name:expr) #:defaults ([name #'"unknown program"]))
```

The flaw in this approach is small: it pushes a bit of the interpretation of the syntax from the template into the pattern. On the one hand, the option of moving interpretation into patterns is why syntax classes support attributes; on the other hand, it is irritating to be forced into that style of programming due to deficiencies in the template notation.

Racket's "try/catch" template form, written `~?`, solves this problem. A `~?` template initially tries to instantiate its first subtemplate. If that fails because of an absent pattern variable, it instantiates its second subtemplate instead. Using `~?` the fragment above can be rewritten as follows:

```
(define-syntax (command-line stx)
  (syntax-parse stx
    [(_ n:maybe-name ___)
     #'(___ (~? n.name "unknown program") ___)]))
```

This template form can be used together with the splicing form. For example, consider a `call-with-lock` procedure that takes a lock, a thunk to be called with the lock acquired, and an optional `#:fail` keyword argument with a failure thunk. We can write a `with-lock` "thunking" macro as follows:

```
(define-syntax (with-lock stx)
  (syntax-parse stx
    [(_ lock:expr (~optional (~seq #:fail fail:expr)) body:expr ...+)
     #'(call-with-lock lock
         (lambda () body ...)
         (~? (~@ #:fail (lambda () fail)) (~@)))]))
```

If the macro does not receive a `#:fail` keyword argument, it omits the argument to the procedure call entirely. Thus the macro doesn't need to duplicate the default handling done by the procedure.

It is common—as in the macro above—for a `~?` template's alternative branch to produce nothing at all. For convenience, the template `(~? T (~@))` can be abbreviated `(~? T)`.

2.3 Metafunctions

The third extension allows computation to be embedded within template instantiation.

Recall the `kw-hash` macro from section 2.1. For simplicity, we converted the keywords from the input into symbols at run time. What if we wanted to do that computation at compile time instead?

Such compile-time computation is typically done within `with-syntax` clauses—or by pushing the computation into patterns or syntax classes, where it may not necessarily belong. An alternative is to move the computation into the template itself using a template *metafunction*.[3] For example, here is a metafunction that expects a single keyword argument and converts it to an identifier:

```
(begin-for-syntax
  (define-metafunction (Keyword->Identifier stx)
    (syntax-parse stx
      [(_ k:kw) (datum->syntax #'k (keyword->symbol (syntax-e #'k)))])))
```

Using the metafunction, we can rewrite `kw-hash` as follows:

```
(define-syntax (kw-hash stx)
  (syntax-parser
    [(_ (~seq key:kw value:expr) ...)
     #'(hash (~@ (quote (Keyword->Identifier key)) value) ...)]))
```

[3] Template metafunctions were inspired by the metafunctions of Redex (Felleisen et al. 2009).

This example also demonstrates the cooperation of metafunctions and ellipses.

If a syntax pattern variable is just like a variable whose name is marked so that the `syntax` form knows to replace its occurrences in a template, likewise a template metafunction is just like a function whose name is marked so the `syntax` form knows to apply it. The function is given syntax and must produce syntax; it is applied to the result of instantiating the subtemplate, and the value it returns is the result of the metafunction template.

A metafunction is typically defined within a `begin-for-syntax` block so that it can be used in the implementation of a macro. The implementation of a macro is a compile-time (or "phase 1") expression; thus the templates it contains are compile-time expressions, thus they search the compile-time environment to recognize special identifiers such as `...`, `~@`, and the names of metafunctions. Note that unlike `define-syntax`, the name of the metafunction exists at the same phase level as the implementation. That is, the following are equivalent:

```
(define-template-metafunction (MF stx) body)
⇔ (begin (define-template-metafunction (MF stx) (mf-helper stx))
         (define (mf-helper stx) body))
```

Another common kind of compile-time computation is the synthesis of new identifiers based on macro arguments. For example, here is a macro that defines a struct type property, automatically defining the predicate and accessor names based on the given property name:

```
(define-struct-property prop:connection)
⇒ (define-values (prop:connection prop:connection? prop:connection-value)
    (make-struct-type 'prop:connection))
```

It could be implemented with a helper metafunction like the following, which joins multiple identifiers into one using the lexical context (hygiene information) from the first identifier.

```
(begin-for-syntax
  (define-template-metafunction (Join stx)
    (syntax-parse stx
      [(_ a:id b:id ...)
       (datum->syntax #'a
                      (string->symbol
                       (apply string-append
                              (map symbol->string
                                   (syntax->datum #'(a b ...))))))])))
```

Using the `Join` metafunction, we can define `define-struct-property` thus:

```
(define-syntax (define-struct-property stx)
  (syntax-parse stx
    [(_ name)
     #'(define-values (name (Join name ?) (Join name -value))
         (make-struct-type-property (quote name)))]))
```

We can add an optional argument to the metafunction that lets the user specify what syntax object to draw the lexical context from when creating the new identifier. And we can use `~optional` and `~?` to match and process the optional argument:

```
(begin-for-syntax
  (define-template-metafunction (Join stx)
    (syntax-parse stx
```

```
    [(_ (~optional (~seq #:lctx lctx)) a:id b:id ...)
     (datum->syntax #'(~? lctx a) ___ same as above ___)]))
```

With this version, we could implement the name synthesis for a form like `define-struct`:

```
(define-syntax (define-struct stx)
  (syntax-parse stx
    [(_ name:id (fld:id ...) ___)
     #'(define-values ((Join #:lctx name make- name) (Join name ?) (Join name - fld) ...)
         ___)]))
```

Metafunctions can be recursive. For example, several syntactic forms in Racket have `*` variants where each clause is in the scope (static or dynamic) of the preceding clause. Here is a metafunction that helps implement such forms via a syntactic `Fold` over the clause list:

```
(begin-for-syntax
  (define-template-metafunction (Fold stx)
    (syntax-parse stx
      [(_ op (arg0 . args) base)   #'(op arg0 (Fold op args base))]
      [(_ op () base)              #'base])))
(define-syntax (let* stx)
  (syntax-parse stx
    [(_ ([var:id rhs:expr] ...) body:expr ...+)
     #'(Fold let (([var rhs]) ...) (let () body ...))]))
(define-syntax (parameterize* stx)
  (syntax-parse stx
    [(_ ([param:expr rhs:expr] ...) body:expr ...+)
     #'(Fold parameterize (([param rhs]) ...) (let () body ...))]))
```

3 THE TEMPLATE LANGUAGE

This section gives a semi-formal description of the extended template language. This section also discusses some issues that, while not novel, affect the implementation of templates in Racket.

3.1 Templates and Head Templates

The Scheme reports (Clinger and Jonathan Rees (eds.) 1991; Kelsey et al. 1998; Shin et al. 2013; Sperber et al. 2009a) specify templates through two nonterminals: "templates" and "template elements"—the latter is a template followed by zero or more ellipses. We generalize this distinction to *templates* and *head templates*. When instantiated, a template yields a single term; a head template yields a sequence of terms (*not* a term that contains a list).

The following grammar describes the syntax of templates (`T`) and head templates (`H`):

```
T = PVar                H = T                    PVar = pattern variable
  | Literal               | H ...
  | (... T)               | (~@ . T)                MF = metafunction identifier
  | (H . T)               | (~? H H)
  | (~? T T)              | (~? H)           Literal = other identifier or atomic datum
  | (MF . T)
```

Pattern variables, literals (including `()`), and ellipses are standard. Note that the definition of `H` allows ellipses to follow any head template, including one that already has ellipses. The form `(... T)` is an *ellipsis escape*—within the subtemplate, occurrences of `...`, `~@`, and `~?` are treated as literals.

The head of a pair template is a head template (thus the name). Ellipses form head templates, and they are allowed to follow arbitrary head templates, including those that already have trailing ellipses. The splicing form (`~@`) is a kind of head template, because it must occur at the head of a syntax list. For example, `(1 . (~@ 2 3))` is not a syntactically valid template. On the other hand, `((~@ . 1) 2)` is syntactically valid, although it will raise an error on instantiation.

This grammar is ambiguous: the template `(T1 ... T2)` could be interpreted like `T1 ...` followed by `T2`—that is, with the ellipses representing repetition—or like `(T1 . (... T2))`—that is, with the ellipses used to escape `T2`. Such ambiguities are always interpreted as repetition, not escaping.

A second problem with this grammar is that it describes S-expression structure, but it is important for syntax templates to correctly handle *syntax object* structure. This is especially true in Racket, which relies on a specific discipline of syntax object placement and the preservation of information from templates to instantiations.

3.2 Syntax Objects

To support their hygienic expansion algorithm, Dybvig et al. (1992) introduced a new data type for representing terms: the *syntax object*. (In this paper, I use the term *syntax object* in the Racket sense, which corresponds to what R6RS calls a *wrapped syntax object* (Sperber et al. 2009a).) Roughly, the new term representation consists of S-expressions enriched with syntax metadata ("wrappings") that carry information for the hygiene algorithm. Different Schemes have chosen different disciplines for where syntax objects occur and when they are introduced by the reader and macro expander. For example, the portable `syntax-case` implementation (Dybvig and Waddell 2016) uses them sparingly, only when they are needed for the correctness and efficiency of the hygiene algorithm.

In contrast, Racket uses them more and with a specific placement discipline, and its support for language-oriented programming relies heavily on that discipline. For example, Racket uses the hygiene information (or "lexical context") attached to the first pair of a function application form to look up the implicit `#%app` macro used to expand the application. This mechanism gives Racket language modules control over the meaning of application; for example, the `racket` language provides an application syntax that handles keyword arguments, and the `lazy` language provides an application syntax that forces the operator and delays the arguments. Similarly, there is an implicit `#%datum` macro that handles atomic literals; `racket`'s default implementation expands into `quote`. Racket also extends syntax metadata to include source locations and arbitrary *syntax properties* as an additional communication channel between macros, the expander, and other language tools.

For pattern-based macros to work correctly with the expander's hygiene algorithm, syntax metadata must be propagated from template to the syntax its produces. For example, instantiating the template `(f X)`, where `X` is a pattern variable, must propagate the metadata on `f`. In Racket, it must also propagate the metadata on the list structure `(f _)`, so that Racket can determine the correct application syntax transformer to use. In contrast, the metadata on `X` should not be propagated to the term substituted for `X`. Similarly, in the template `(Y ...)` the metadata on the ellipsis identifier is not propagated, but the metadata on the whole template is.

To be precise, let us write `(mkstx W C)` for a syntax object wrapping the pair or atom `C` with syntax metadata `W`. The following grammar describes the constraints Racket places on syntax objects—specifically, where the syntax nodes occur; we leave the structure of metadata (`W`) unspecified:

```
Syntax = (mkstx W Atom)
       | (mkstx W (cons Syntax StxList))

StxList = '()
        | Syntax
        | (cons Syntax StxList)
```

Racket's `read-syntax` procedure places syntax nodes (`mkstx` constructors) around every complete S-expression. For example, `read-syntax` produces different syntax objects for the following two S-expression notations usually considered equivalent:

```
(1 2)      ⇒ (mkstx _ (cons (mkstx _ 1) (cons (mkstx _ 2) '())))
(1 2 . ()) ⇒ (mkstx _ (cons (mkstx _ 1) (cons (mkstx _ 2) (mkstx _ '()))))
```

On the other hand, `datum->syntax` inserts the fewest syntax nodes necessary to produce a value valid according to `Syntax`. So `(datum->syntax #f '(1 2))` produces a value with the shape of the first line above.

The template written `(f X)` is represented in Racket as the following syntax object:

```
(mkstx Wp (list (mkstx Wf 'f) (mkstx Wx 'X)))
```

When `X` is bound to `val-of-X` (which must also be a syntax object), instantiating the template produces the following syntax:

```
(mkstx Wp (list (mkstx Wf 'f) val-of-X))
```

In the rest of the paper, we will drop the metadata field in examples intended to illustrate only the placement of syntax objects—for example, `(mkstx '())` instead of `(mkstx _ '())` or `(mkstx W '())`.

3.3 Variables and Ellipses

In order for a template to be legal, it must not only follow the grammar from Section 3.1, it must also use variables and ellipses correctly. These rules are standard; we repeat them here for completeness.

A variable must be used at a depth at least as great as the depth where it was bound. A variable use's depth is the number of ellipses it occurs within. For example, in the template `(X (Y (Y Z) ...) ...)`, `X` occurs at depth 0, the first `Y` occurs at depth 1, and the second `Y` and the `Z` occur at depth 2. The following example is illegal:

```
(with-syntax ([(X ...) ___]) #'X)
```

because `X` is bound at depth 1 and used at depth 0.

An ellipsis must have at least one participating variable. A variable participates in an ellipsis's iteration if the variable occurs *relative to the ellipsis template* at a depth less than or equal to its binding depth. For example, the following example is illegal:

```
(with-syntax ([(X ...) ___]) #'((X ...) ...))
```

because the outer ellipsis has no participating variables; the occurrence of `X` occurs at depth 2 with respect to the outer ellipsis, and `X` is bound at depth 1.

Note that the following template is—perhaps surprisingly—legal:

```
(with-syntax ([(X ...) ___]) #'((X X ...) ...))
```

Both occurrences of `X` occur at depth *at least* 1, and both ellipses have an occurrence of `X` driving the iteration. A variable used at a depth greater than its binding depth participates in the innermost iterations.[4]

Note that template instantiation may fail even for a valid template, if an ellipsis has multiple iteration variables bound to lists of different lengths.

4 TEMPLATE INSTANTIATION

This section describes the meaning of templates via compilation.

[4] I don't know why this behavior was chosen. On the one hand, it is incompatible with the straightforward instantiation algorithm from the original MBE paper. On the other hand, it means that a valid template doesn't change meaning when placed in some (valid) template context.

4.1 Abstract Syntax

The following grammar describes abstract syntax trees for templates (`T`) and head templates (`H`):

```
T = (t:restx Syntax T)                    H = (h:t T)
  | (t:var Id)                              | (h:dots H MapVars)
  | (t:const Atom)                          | (h:splice T)
  | (t:append H T)                          | (h:orelse H H)
  | (t:orelse T T)
  | (t:metafun Id Syntax T)          MapVars = (list (cons Id Id) ⋯)
```

The `t:var`, `t:const`, `t:append`, `t:orelse`, and `t:metafun` variants correspond directly to the pattern variable, literal, pair, try/catch, and metafunction productions of the grammar from Section 3.1. Escaped templates do not need an AST node; escaping only affects parsing.

The pattern variable environment maps pattern variable names to two pieces of information: the ellipsis depth from the variable's binding and an identifier referring to an ordinary variable that holds the pattern variable's value. In the examples, we write `X-var` to refer to the ordinary variable holding the value of pattern variable `X`. If `X` is a depth-0 pattern variable, an occurrence of `X` is represented by `(t:var X-var)`. For pattern variables of higher depth, the `t:var` node contains an iteration variable from the enclosing `h:dots`.

A `h:dots` node stores an association list mapping list-valued "source" variables to iteration variables, and pattern variables within the ellipsis template with non-zero depth are represented as `t:var` nodes with the iteration variable name. For example, if `X` has depth 1 and `Y` has depth 0, the template `((X Y) ...)` has an AST like the following:

```
(t:restx _
  (t:append
    (h:dots
      (h:t (t:append (h:t (t:var X-iter1-var))
             (t:append (h:t (t:var Y-var))
               (t:const '()))))
      (list (cons X-var X-iter1-var)))
    (t:const '())))
```

The ASTs for nested ellipses must use distinct iteration variables for uses of the same pattern variable at different levels, such as the two `X`s in `((X X ...) ...)`, where `X` has depth 1.

A `t:metafun` node contains a reference to the metafunction's implementation, the syntax of the metafunction call in the original template, and the "argument" subtemplate. The call syntax is used to construct the syntax passed to the metafunction; like a macro, a metafunction gets a single syntax argument that begins with the metafunction's own name.

A `t:restx` ("re-syntax") AST node represents a syntax object in the template whose metadata must be propagated to the template's result during instantiation. Recall from section 3.2 that not all syntax objects in templates cause metadata propagation. Here are some examples:

- The template written `(X)` is represented by the syntax `(mkstx (list (mkstx 'X)))`.
  It parses as `(t:restx _ (t:append (h:t (t:pvar X-var)) (t:const '())))`.
- The template written `(X . ())` is represented by the syntax `(mkstx (cons (mkstx 'X) (mkstx '())))`.
  It parses as `(t:restx _ (t:append (h:t (t:pvar X-var)) (t:restx _ (t:const '()))))`.
- The template written `(~? X Y)` is represented by `(mkstx (list (mkstx '~?) (mkstx 'X) (mkstx 'Y)))`, and it is parsed as `(t:orelse (t:var X-var) (t:var Y-var))`.

```
; compile-t : Template -> (Expressionof Syntaxish)
(define (compile-t t)
  (match t
    [(t:restx stx t)  `(restx (quote-syntax ,stx) ,(compile-t t))]
    [(t:var var)      `(check-syntax ,var)]
    [(t:const datum)  `',datum]
    [(t:append h t)   `(append ,(compile-h h) ,(compile-t t))]
    [(t:orelse t1 t2) `(orelse (lambda () ,(compile-t t1)) (lambda () ,(compile-t t2)))]
    [(t:metafun mf stx t)
     `(apply-metafun ,(metafunction-var mf) (quote-syntax ,stx) ,(compile-t t))]))

; compile-h : HeadTemplate -> (Expressionof (Listof Syntax))
(define (compile-h h)
  (match h
    [(h:t t)          `(list ,(compile-t t))]
    [(h:splice t)     `(check-splice ,(compile-t t))]
    [(h:orelse h1 h2) `(orelse (lambda () ,(compile-h h1)) (lambda () ,(compile-h h2)))]
    [(h:dots h mapvars)
     (define vars (map cdr mapvars))
     (define srcs (map (lambda (src) `(check-list ,src)) (map car mapvars)))
     `(apply append (map (lambda ,vars ,(compile-h h)) ,@srcs))]))
```

Fig. 1. Compiling template ASTs

The template written `(~? X . (Y))` is represented by `(mkstx (cons (mkstx '~?) (cons (mkstx 'X) (mkstx (list (mkstx 'Y))))))`, but it is parsed into exactly the same AST as the first version. Essentially, the extra syntax node is on part of the argument list to `~?`, not on a subtemplate itself.

- Likewise, `(X ...)` and `(X . (...))` also have the same ASTs.

4.2 A Naive Implementation

Figure 1 shows a simple compiler from template ASTs to Scheme/Racket code. It assumes that names such as `quote` and `append` have their standard bindings, and it relies on helper functions such as `restx`, `orelse`, and `check-syntax`, some of which are shown in figure 2.

We use `quote` for literal atoms and `quote-syntax` for literal syntax objects. In `restx`, we use the third and fourth arguments to Racket's `datum->syntax` transfer source location and syntax properties from the original template syntax to the instantiation.

Due to `syntax-parse`'s generalization of pattern variables to allow absent values and values other than syntax, pattern variable references, including the references in ellipsis `map`s, must be checked for proper values. Checking is done one level at a time so that handling of absent values occurs at the right place.[5] For example, consider the template `((~? X Y) ...)`; in each iteration, `X` may be present or absent, independent of its presence or absence in other iterations. When a checking helper function detects an absent variable value, it calls a handler that escapes to the nearest enclosing `orelse`; if there is no enclosing `orelse`, the handler raises an error. The handler

[5]The implementation in Racket also allows promises, which are forced during checking.

```
; restx : Syntax Any -> Syntax
(define (restx stx v) (datum->syntax stx v stx stx))

; apply-metafun : (Syntax -> Any) Syntax Syntaxish -> Syntax
(define (apply-metafun mf stx v)
  (define result (mf (datum->syntax stx (cons (stx-car stx) v))))
  (if (syntax? result) result) (error ___))

; current-absent-handler : (Parameterof (-> (escapes)))
; check-{list,syntax} and orelse cooperate via this handler for absent variables
(define current-absent-handler (make-parameter (lambda () (error ___))))

; check-list : Any -> List (or escapes)
(define (check-list v)
  (if (list? v) v (if (not v) ((current-absent-handler)) (error ___))))

; check-syntax : Any -> Syntax (or escapes)
(define (check-syntax v)
  (if (syntax? v) v (if (not v) ((current-absent-handler)) (error ___))))

; orelse : (-> X) (-> X) -> X, where X is either Syntax or (Listof Syntax)
(define (orelse proc1 proc2)
  ((let/ec escape
     (parameterize ((current-absent-handler (lambda () (escape proc2))))
       (let ([result1 (proc1)]) (lambda () result1))))))
```

Fig. 2. Helper functions for compiled templates

could also be implemented using a standard exception system with a new exception subtype. Note that `~?` only catches absent variables, not other errors like splicing a non-list or mismatches in ellipsis variable lengths.

Metafunction application combines the result of instantiating the argument template with the original metafunction identifier to form the syntax object that gets passed to the metafunction's implementation.

4.3 Implementation Improvements

The implementation above is straightforward, but there are many ways it can be improved.

Eliminate useless variable checks. The possibility of absent or bad syntax variable values only arises from a few features of `syntax-parse`. Pattern variables bound using those features are marked as needing checks, and the template compiler can omit checks from other pattern variables, including all variables bound using `syntax-case` and `syntax-rules`.

Prune syntax constants. The syntax constants passed to `restx` are used only as a source of their lexical context, source location, and syntax properties; their contents don't matter. Instead of storing the whole (potentially large) syntax object `stx`, the metadata can be transferred to a small syntax object, such as `(datum->syntax stx 'STX stx stx)`, and that can be stored instead. Likewise, the syntax constant for a metafunction application can be trimmed to the outer list and its first element (the metafunction identifier).

Recognize lists. The naive compiler, following the naive AST design, gleefully produces code of the form `(append (list _) _)` for every simple pair template. These can be peephole-optimized during compilation, but it is helpful for other optimizations to add new AST variants for these: `t:list` and `t:list*`.

Coalesce syntax constants. Given a constant template like `(lambda (x) 1)`, the naive compiler produces code that builds the term up from atomic pieces. Instead, it should recognize that the whole template is a syntax constant. We overload `t:const` to store both atomic data and syntax constants; the latter is compiled using `quote-syntax` rather than `quote`.

Recognize single-term ellipsis subtemplates. In general, the subtemplate of an ellipsis is a head pattern; most often, however, it is an `h:t` with an ordinary template. By specializing that case, the compiler can omit a `(list _)` wrapper within the `map` and an `(apply append _)` outside of it.

Specialize `(X ...)` Furthermore, if the subtemplate of an ellipses is just a pattern variable, and if the pattern variable is "trusted" (its list and syntax checks can be omitted), then the entire `map` expression can be eliminated, since the variable already stores a list of syntax values.

4.4 Substitution: Optimizing for Space

One of my goals for the new template implementation was to avoid increasing the size of compiled templates—that is, the size of the serialized bytecode. In addition to the optimizations listed above, there is another optimization that was important for keeping compile sizes no larger than the previous implementation.

It is generally more efficient for Racket to serialize one large syntax constant (`quote-syntax` expression) than to serialize its components individually. Roughly, the hygiene algorithm and bytecode serializer face time/space trade-offs in sharing syntax metadata, and less sharing occurs across different syntax constants.

Consider the template `(define X 10)`. With the optimizations discussed in the previous section, the corresponding AST and generated code would be the following:

```
(t:restx (mkstx ___)
  (t:list (list (t:const (mkstx define)) (t:var X-var) (t:const (mkstx 10)))))
⇒ (restx (quote-syntax STX)
      (list (quote-syntax define) X-var (quote-syntax 10)))
```

But we can treat it instead as a "starting point" syntax constant together with substitution instructions to be performed during template instantiation, as follows:

```
(t:subst (mkstx (list (mkstx 'define) (mkstx '_) (mkstx 10)))
  (list (sub:skip) (sub:elem (t:var X-var))))
⇒ (subst (quote-syntax (define _ 10)) 1 'elem X-var)
```

The `subst` procedure takes a syntax object containing a (possibly improper) list and processes each element in the list. An element can be retained as a constant (`(sub:const)`), replaced with a single element generated by a template (`(sub:elem T)`), or replaced with a sequence of elements generated by a head template (`(sub:append H)`). If the template has an improper tail that is not constant, such as the template `(let () . BODY)`, the tail can be replaced with the result of its template using `(sub:tail T)`.

We can achieve even more sharing by building recursion into our little substitution interpreter. For example, consider the template `(if (PRED x) x #f)`. Without explicit support for recursion, that can be expressed as two nested substitutions:

```
(t:subst (mkstx (list (mkstx 'if) (mkstx '_) (mkstx 'x) (mkstx #f)))
  (list (sub:skip)
        (sub:elem (t:subst (mkstx (list (mkstx '_) (mkstx 'x)))
                           (list (sub:elem (t:var 'PRED-var)))))))
```

```
T = ....
  | (t:const Syntax)
  | (t:list (list T ···))
  | (t:list* (list T ···) T)
  | (t:subst Syntax (list Subst ···))
```

```
Subst = (sub:const)
      | (sub:elem T)
      | (sub:append H)
      | (sub:tail T)
      | (sub:recur-elem (list Subst ···))
      | (sub:recur-tail (list Subst ···))
```

Fig. 3. Extended abstract syntax

When an element's subtemplate is another substitution node, we can merge the inner starting point syntax to its original position in the outer starting point and use `sub:recur-elem` with the inner list of substitutions:

```
(t:subst
  (mkstx (list (mkstx 'if) (mkstx (list (mkstx '_) (mkstx 'x))) (mkstx 'x) (mkstx #f)))
  (list (sub:skip) (sub:recur-elem (list (sub:elem (t:var PRED-var))))))
```

When the `subst` interpreter sees `'recur-elem`, it recurs with the subsequent list of substitutions, using the current element as the starting point of the recursive call.

Figure 3 gives a summary of the changes to the template IR. We have essentially arrived at a hybrid compiler/interpreter approach. In fact, an earlier implementation of this template language simply used interpreter for every template variant, but managing the explicit environment data structures for ellipsis iterations—and making it reasonably fast—was a headache, and the code was inelegant. Racket is better at managing variables and environments than I am.

5 CONCLUSION

This paper presents three extensions to the syntax templates of Scheme and Racket. These extensions complement `syntax-parse`'s extension to syntax patterns, which were themselves motivated by Racket's introduction of keywords and nonparenthesized grouping in syntactic forms. The new features are straightforward to implement.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 695412).

BIBLIOGRAPHY

William Clinger and Jonathan Rees (eds.). Revised[4] Report on the Algorithmic Language Scheme. 1991. http://www.scheme-reports.org/

Ryan Culpepper. Fortifying macros. *Journal of Functional Programming* 4-5(22), pp. 224–243, 2012.

R. Kent Dybvig. *The Scheme Programming Language*. 3rd edition. MIT Press, 2003. https://www.scheme.com/tspl3/ Section 8.2 discusses macro templates.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), pp. 295–326, 1992. http://dx.doi.org/10.1007/BF01806308

R. Kent Dybvig and Oscar Waddell. Pre-R6RS Portable syntax-case. 2016. No longer online. Retrieved from https://web.archive.org/web/20160414073020/http://www.cs.indiana.edu/syntax-case/old-psyntax.html.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 1st edition. The MIT Press, 2009.

Matthew Flatt. keyword arguments, take 2. plt-scheme mailing list, 2007. https://lists.racket-lang.org/users/archive/2007-June/018964.html

R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11(1), 1998. https://doi.org/10.1023/A:1010051815785

Eugene E. Kohlbecker and Mitchell Wand. Macro-by-Example: Deriving Syntactic Transformations from their Specifications. In *Proc. Symposium on Principles of Programming Languages (POPL '87)*, pp. 77–84, 1987. http://doi.acm.org/10.1145/41625.41632

Alex Shin, John Cowan, and Arthur A. Glecker (eds.). Revised^7 Report on the Algorithmic Language Scheme. 2013. http://www.scheme-reports.org/

Michael Sperber, Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised^6 Report on the Algorithmic Language Scheme—Standard Libraries. 2009a. http://www.r6rs.org/final/r6rs-lib.pdf

Michael Sperber, Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised^6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming* 19(S1), pp. 1–301, 2009b. https://doi.org/10.1017/S0956796809990074