JFP **22** (2): 181–216, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000093

Macros that Work Together

Compile-time bindings, partial expansion, and definition contexts

MATTHEW FLATT, RYAN CULPEPPER, DAVID DARAIS
University of Utah
(*e-mail:* mflatt@cs.utah.edu, ryan@cs.utah.edu, david.darais@gmail.com)

ROBERT BRUCE FINDLER
Northwestern University
(*e-mail:* robby@eecs.northwestern.edu)

Abstract

Racket is a large language that is built mostly within itself. Unlike the usual approach taken by non-Lisp languages, the self-hosting of Racket is not a matter of bootstrapping one implementation through a previous implementation, but instead a matter of building a tower of languages and libraries via macros. The upper layers of the tower include a class system, a component system, pedagogic variants of Scheme, a statically typed dialect of Scheme, and more. The demands of this language-construction effort require a macro system that is substantially more expressive than previous macro systems. In particular, while conventional Scheme macro systems handle stand-alone syntactic forms adequately, they provide weak support for macros that share information or macros that use existing syntactic forms in new contexts. This paper describes and models features of the Racket macro system, including support for general compile-time bindings, sub-form expansion and analysis, and environment management. The presentation assumes a basic familiarity with Lisp-style macros, and it takes for granted the need for macros that respect lexical scope. The model, however, strips away the pattern and template system that is normally associated with Scheme macros, isolating a core that is simpler, can support pattern and template forms themselves as macros, and generalizes naturally to Racket's other extensions.

1 Macros as a compiler-extension API

The progression from text pre-processors (such as the C pre-processor) to Lisp macros to Scheme macros is an evolution toward a wider compiler API – one that, at the Scheme end, exposes the compiler's management of lexical context. This widening of the API makes certain language extensions possible that were technically impossible before, such as a local transformer that reliably expands to a reference of an enclosing binding.

The classic example of a scope-respecting macro is `or`, which (in simplified form) takes two expressions. It returns the value of the first expression if it is not `#f` (i.e., false) or the value of the second expression otherwise:

```
(or e1 e2) ⇒ (let ([tmp e1]) (if tmp tmp e2))
```

The `tmp` binding in the expansion of `or` ensures that the first expression is evaluated only once. A Scheme macro system ensures that the `or` macro works as expected in a setting like this expression:

```
(let ([tmp 5]) (or #f tmp))
```

An expansion oblivious to scope would allow the `or`-introduced `tmp` binding to shadow the outer binding of `tmp`; the program would then produce `#f` (false) instead of `5`. Instead, Scheme's hygienic macro expander (Kohlbecker *et al.*, 1986) preserves the original apparent binding structure, and the `or`-introduced `tmp` does not shadow the outer `tmp`.

Although Scheme is best known for its pattern-matching macros (Kohlbecker & Wand, 1987; Clinger & Rees, 1991), the crucial addition in Scheme's macro API compared to Lisp is the *syntax object* data-type (Dybvig *et al.*, 1993; Sperber, 2009), along with an operator for quoting literal program fragments. A syntax object represents a program fragment and carries with it information needed to respect lexical scope. The `#'` quoting operator is like the `'` operator, but `#'` produces a syntax object that encapsulates the program fragment's *lexical context* – the bindings in scope where the quoted fragment occurs. For example, the syntax object produced by

```
(let ([x 1]) #'x)
```

records that the program fragment `#'x` occurred in the context of a particular binding of `x`. A syntax object's lexical context can be inspected through functions such as `free-identifier=?`, which determines whether two syntax objects correspond to identifiers that are bound in the same place:

```
> (let ([x 1]) (free-identifier=? #'x #'x))
#t
> (free-identifier=? (let ([x 1]) #'x)
                     (let ([x 1]) #'x))
#f
```

Functions like `free-identifier=?` are typically used within *procedural macros*, which are bound with `define-syntax` and can be arbitrary functions that transform a source syntax object into a new syntax object.

Racket builds on procedural macros and syntax objects while further expanding the compiler functionality that is available through macros. The Racket macro API exposes the compiler's general capability to bind and access compile-time information within a lexical scope, as well as the compiler's ability to expand a sub-expression's macros. This wider macro API enables language extensions that were technically impossible (or, at best, awkward to simulate) in the narrower macro API of earlier Scheme systems. Such extensions can be generally characterized as macros that cooperate by sharing compile-time information, and we describe several examples in Section 2.

Section 3, which is the bulk of the paper, presents a model of Racket macros. The full model is about three pages long. For its presentation, we build up the model in a way that imitates the historical evolution of macro systems. We start with a core language and basic parsing rules, then add scope-oblivious macros, next add tracking of lexical scope within syntax objects, and finally add support for sub-form expansion and definition contexts.

2 Cooperating macros

Macros in Racket cooperate with each other in many different ways, including the way that `define-struct` provides information for the `match` form, the way that the `class` form leverages `define` and `lambda`, and the way that `lambda` propagates information about definitions within its body to later definitions. These uses illustrate key tools for cooperation: compile-time bindings, sub-form expansion (both complete and partial), and definition contexts.

2.1 Structure definition and matching

Whereas Scheme has just one notion of compile-time information, the macro, Racket supports the binding of identifiers to arbitrary compile-time information. One such example is structure information, which is how `define-struct` communicates information about the shape of a structure declaration to the `match` pattern-matching form.

The `define-struct` form expands to a set of run-time definitions using `define`, plus a single compile-time binding using `define-syntax`. For example,

```
(define-struct egg (color size))
```

expands to the following definitions:

```
(define (make-egg c s) ....) ; Primitive egg constructor
(define (egg? v) ....)       ; Predicate to distinguish eggs
(define (egg-color e) ....)  ; Accessor for the color field
(define (egg-size e) ....)   ; Accessor for the size field
(define-syntax egg           ; Static information about eggs
  (make-struct-desc #'make-egg #'egg? ....))
```

The `make-egg` function is a constructor, the `egg?` function is a predicate, the `egg-color` function is a selector, and so on. The `egg` binding, meanwhile, associates a static description of the structure type – including references to its constructor, predicate, and selector functions – with the name `egg` for use in other macros. In general, the use of `define-syntax` does not always create a macro. If the value bound to the identifier introduced by `define-syntax` is a function, then the macro expander knows to call that function when it sees the identifier, but if the identifier is bound to something else, then using the identifier results in a syntax error.

Cooperating macros can, however, use the `syntax-local-value` function to extract the value bound to the identifier. In particular, the `match` pattern-matching form recognizes bindings to structure definitions using `syntax-local-value`, and it generates code that uses the predicate and selector functions. For example,

```
(define (blue-egg-size v)
  (match v
    [(egg 'blue s) s]))
```

expands to roughly

```
(define (blue-egg-size v)
  (if (and (egg? v) (eq? (egg-color v) 'blue))
      (egg-size s)
      (error "match: no matching case")))
```

The implementation of `match` uses `syntax-local-value` on the `egg` identifier to learn about its expected number of fields, its predicate, and its selector functions.

Using `define-syntax` for both macros and other compile-time bindings allows a single identifier to play multiple roles. For example, `make-struct-desc` in the expansion of Racket's `define-struct` macro produces a value that is *both* a structure description and a function.[1] Since the descriptor is also a function, it can act as a macro transformer when `egg` is used as an expression. The function behavior of a structure descriptor is to return the identifier of structure's constructor, which means that `egg` as an expression is replaced by the `make-egg` constructor, that is `(egg 'blue 1)` expands to `(make-egg 'blue 1)`. Overloading the `egg` binding in this way allows `egg`-constructing expressions and `egg`-matching patterns to have the same shape.

2.2 Patterns and templates

Macro transformers typically pattern match on uses of the macro to generate the macro's expansion. Although transformers could use the `match` form to match macro uses, Racket provides the `syntax-case` pattern-matching form, which is more specialized to the task of matching syntax fragments. The `syntax-case` form matches syntax objects, and the associated `syntax` form produces a syntax object using pattern variables that are bound by `syntax-case`. Arbitrary Racket code can occur between `syntax-case`'s binding of pattern variables and the `syntax` templates that use them.

For example, the following implementation of `defthunk` expands a use like

```
(defthunk f (random))
```

to

```
(define (f) (random))
```

The macro transformer receives the use of `defthunk` as an `in-stx` argument, which is a syntax object. The `syntax-case` form attempts to match `in-stx` to the pattern `(defthunk g e)`, which matches when the use of `defthunk` has exactly two

[1] A structure description is itself a structure, and a structure can have a `prop:procedure` property that determines how the structure behaves when applied to arguments.

sub-forms; in that case, `g` is bound to the first sub-form and `e` is bound to the second one:

```
(define-syntax defthunk
  (lambda (in-stx)
    (syntax-case in-stx ()
     [(defthunk g e)
      (if (identifier? (syntax g))
          (syntax (define (g) e))
          (error "need an identifier"))])))
```

The `(syntax g)` expression in the matching clause refers to the part of `in-stx` that matched `g` in the pattern, and the macro transformer checks that the `g` part of a use `in-stx` is an identifier. If so, the matching pieces `g` and `e` are used to assemble the macro expansion.

A challenge in implementing `syntax-case` and `syntax` is communicating the pattern variables bound by `syntax-case` to the uses in a `syntax` template. Since the right-hand side of a `syntax-case` clause can be an arbitrary expression, `syntax-case` cannot easily search for uses of `syntax` and replace pattern variables with match references. One way to handle this problem is to build `syntax-case` and `syntax` (or, at least, the notion of pattern variables) into the macro system. With generalized compile-time bindings like those in Racket, however, `syntax-case` can be implemented instead as a macro that binds each pattern variable to compile-time information describing how to access the corresponding matched value, and `syntax` checks each identifier in a template to determine whether it refers to such compile-time information.

For example, the above `syntax-case` clause is translated to the following:

```
(let ([tmp-g ... extract g from in-stx ...]
      [tmp-e ... extract e from in-stx ...])
  (if (and tmp-g tmp-e) ; if the pattern matched...
      (let-syntax ([g (make-pattern-var (syntax tmp-g) 0)]
                   [e (make-pattern-var (syntax tmp-e) 0)])
        (if (identifier? (syntax g))
            (syntax (define (g) e))
            (error "need an identifier")))
      (error "bad syntax")))
```

Ignore for the moment that this expansion itself is being used as compile-time code. The `syntax-case` form can be used in a run-time position, so think of the above expression as run-time code.

The `g` and `e` pattern variables in the original `syntax-case` form are represented in the expansion by compile-time records that contain references to the `tmp-g` and `tmp-e` variables that store the matched sub-forms. The records also store the ellipsis depth (Kohlbecker & Wand, 1987) of the pattern variables so that `syntax` can report mismatches at compile time. The `syntax` form checks each identifier in its template; if it is bound to a compile-time pattern variable record, it is translated to a reference to the corresponding run-time variable; otherwise, it is preserved as a literal syntax object. The inner `if` from therefore expands to

```
(if (identifier? tmp-g)
    (datum->syntax (list #'define (list tmp-g) tmp-e))
    (error "need an identifier"))
```

where the `datum->syntax` primitive converts list structure into a syntax object.

When `syntax-case` is used in a compile-time position, it binds pattern variables as meta-compile-time information, and pattern variables in templates are replaced by compile-time variables. This kind of phase shifting is straightforwardly handled by the Racket macro expander (Flatt, 2002).

2.3 Classes, definitions, and functions

The syntax of a Racket `class` expression is

```
(class superclass-expr decl-or-expr*)
```

The *superclass-expr* can be the built-in `object%` class[2] or any other class, but the *decl-or-expr* sequence is our primary interest. The sequence declares all of the fields and methods of the class, in addition to expressions that are evaluated when the class is instantiated (analogous to a constructor body).

A typical use of the `class` form defines some private fields and public methods. To make the syntax of `class` easier for Racket programmers to remember, the syntax for such declarations within a class builds on the standard `define` form normally used to define variables and functions. For example,

```
(define chicken%
  (class object%
    (define eggs empty)
    (public nesting? lay-egg)
    (define (nesting?)
      (not (empty? eggs)))
    (define (lay-egg color size)
      (set! eggs (cons (make-egg color size)
                       eggs)))
    ....))
```

defines a class `chicken%` that has a private field `eggs` and public methods `nesting?` and `lay-egg`.

More than making the syntax easier to remember, reusing `define` for field and method declarations means that syntactic forms that expand to `define` also can be used. For example, a variant of `define` might support optional arguments by expanding to the plain `define` form:

[2] In Racket, class names traditionally end in %.

```
(define/opt (lay-egg [color 'brown] [size 3])
  (set! eggs (cons (make-egg color size)
                   eggs)))
```

which expands to Scheme's `case-lambda` form to handle varying number of arguments:

```
(define lay-egg (case-lambda ....))
```

As another example, programmers using `class` often use a `define/public` form to declare a public method, instead of writing separate `define` and `public` forms. The `define/public` form expands to a sequence of `public` and `define` declarations.

Finally, although it is implicit in the function-shorthand uses of `define` above, the `class` form also reuses `lambda` for method declarations. For example, the `nesting?` method could have been written as

```
(define nesting?
  (lambda () (not (empty? eggs))))
```

Similar to `define`, any macro that expands to `lambda` can be used with a `define` (or a macro that expands to `define`) to describe a method.

In order for the `class` macro to properly expand, it must be able to detect all bindings and functions in its body. Specifically, the macro must see all definitions to build a table of fields and methods, and it must see the functions that implement methods so that it can insert the implicit `this` argument (which a method receives when it is called) into the method's argument list. Thus, to allow the use of declaration forms like `define/public`, the `class` macro must force the expansion of each *`decl-or-expr`* to expose the underlying uses of `define`, `lambda`, and `public`.

Scheme macro systems do not typically provide a way to force expansion of a sub-form in the way that `class` requires. Sub-forms are normally expanded only when they appear directly within a core syntactic form, after all of the surrounding macros have been expanded away. That is, when a macro transformer returns an expression that contains macro uses, the sub-expression macros are expanded iteratively. The `class` form, however, needs to force expansion of its sub-forms before producing its result.

The `class` form forces sub-expression expansion using the Racket `local-expand` function. The `local-expand` function takes a syntax object to expand, along with other arguments to be described later, and it returns the expanded form as a syntax object. The resulting syntax object can be inspected, transformed, and incorporated into a larger result by the macro transformer.

2.4 Internal definitions

The reuse of `define` in `class` has a precedent in standard Scheme: `define` can be used inside `lambda` and other block forms to create local definitions. For example,

```
(define (cook eggs)
  (define total-size (sum-eggs eggs))
  (if (< total-size 10)
      (cook-in-small-pan eggs)
      (cook-in-big-pan eggs)))
```

creates a local binding `total-size` that is available only with the function body. Local definitions like this are called *internal definitions*.

In a fully expanded program, internal definitions can be replaced with a `letrec` local binding form.[3] The process of macro expansion must somehow discover and convert internal definitions to `letrec` forms. Complicating this process, an internal definition can bind a macro instead of a run-time variable, or an internal definition can shadow the binding of an identifier from the enclosing environment. Each of such cases can affect the expansion of later forms in a function body, even affecting whether the form is treated as an internal definition or as an expression, as in the following case:

```
(define (cook-omelette eggs)
  (define-syntax-rule (define-box id)
    (define id (box empty)))
  (define-box best-eggs)
  (define-box left-overs)
  (take-best-eggs! eggs best-eggs leftovers)
  (values (make-omelette (unbox best-eggs))
          rest-eggs))
```

To handle the interaction of internal definitions and expressions, a syntactic form that allows internal definitions must partially expand each of its body sub-forms to determine which are definitions. Each macro definition must be installed immediately for use in expanding later body forms. If partial expansion reveals a run-time definition, expansion of the right-hand side of the definition must be delayed, because it might refer to bindings created later in the body (e.g., a forward reference to a function or macro that is defined later in the body).

These issues are typically resolved internal to a Scheme macro expander (Ghuloum & Dybvig, 2007; van Tonder, 2007) so that only built-in forms like `lambda` can accommodate internal definitions. Racket gives a macro transformer all of the tools it needs to implement internal-definition contexts: partial sub-form expansion, an explicit representation of definition contexts, and an operation to extend a definition context with bindings as they are discovered. Consequently, a `lambda` form that supports internal definitions can be implemented in terms of a simpler `lambda` that allows only expressions in its body. Similarly, the `class` form can support local macros among its field and method definitions, or a `lambda` variant can support definitions mixed with expressions in its body (instead of requiring all definitions first, as in the standard Scheme `lambda` form).

[3] In the current Scheme standard (Sperber, 2009), internal definitions are converted to a `letrec*` form. Racket's `letrec` form corresponds to the standard's `letrec*` form.

To perform *partial expansion* of their sub-forms, the `lambda` and `class` macros provide `local-expand` with a *stop list*, a list of identifiers to use as stopping points in expansion. For `lambda`, the stop list includes only the core syntactic forms, ensuring that all definition-producing macros are expanded into core definitions. The Racket specification pins down the set of core syntactic forms, and the corresponding identifiers are assembled in a library-provided list, which is sufficient to make most macros cooperate properly. The `class` macro uses a stop list that also includes identifiers like `#'public` and `#'override`, since those forms must be caught and interpreted by the `class` macro; they are meaningless to the Racket macro expander.[4] When macro uses nest and the corresponding transformers use partial expansion, the inner transformer's partial expansion is not affected by the stop list of the outer transformer, so macros need not be aware of the stop lists of other macros.

To support internal definitions, the `lambda` and `class` macros generate a new *definition context* value using the `syntax-local-make-definition-context` function. The macros provide this context value to `local-expand` along with the stop list to partially expand the body forms in the scope of the definitions uncovered so far. When the `lambda` or `class` macros detect a new definition via partial expansion, they install new bindings into the definition context using `syntax-local-bind-syntaxes`. When the macros detect a `define` form, they call `syntax-local-bind-syntaxes` with just the defined identifiers, which are added to the definition context as bindings for run-time variables. When the macros detect a `define-syntax` form, they call `syntax-local-bind-syntaxes` with identifiers and the corresponding compile-time expression, which is evaluated and associated with the identifiers as compile-time bindings.

2.5 Packages

Definition contexts and compile-time binding further enable the implementation of a local-module form as a macro. Racket's `define-package` form resembles the `module` form from Chez Scheme (Waddell & Dybvig, 1999) and the `structure` form of ML (Milner *et al.*, 1990). A set of definitions within a package can see each other, but they are hidden from other expressions. Exported identifiers listed after the package name become visible when the package is explicitly opened:

```
(define-package carton (eggs)
  (define egg1 (make-egg 'blue 1))
  (define egg2 (make-egg 'white 2))
  (define eggs (list egg1 egg2)))
....
(open-package carton)
```

To allow definitions within a package to see each other, the `define-package` form creates a definition context for the package body. The definition context does not escape the package body, so no other expressions can directly access the package contents.

[4] By convention, identifiers such as `public` are bound as macros that raise a syntax error when used incorrectly – that is, outside a `class` body.

Meanwhile, the package name is bound to a compile-time description of the contents so that `open-package` can make the exported names available in a later scope, and the package name itself can be exported and imported like any other binding. When a package is opened with `open-package`, the package's names are made available by new `define-syntax` bindings that redirect to the package's hidden definitions.

Naturally, packages can be defined within packages, which is supported in the macro API by allowing definition contexts to nest. Going even further, `define-package` supports a `define*` form that binds an identifier for only *later* expressions within the package body like ML's nested `val` bindings instead of Scheme's mutually recursive `define` bindings. Such variations on binding scopes are possible in Racket because the machinery of definition contexts is exposed in the macro API.

2.6 Tools

The DrRacket programming environment includes many tools that manipulate Racket programs and modules, including a debugger, a profiler, and a syntax checker. These tools all work by first expanding the program so that they need to handle only the core forms of the language. The tools are not macros, but they gain many of same sorts of benefits as cooperating macros by using an `expand` function that produces a syntax object.

A typical Scheme macro expander (Ghuloum & Dybvig, 2007; van Tonder, 2007) takes a syntax object and produces a raw S-expression (i.e., pairs and lists), but the `expand` function produces a syntax object for the expanded program. Through syntax objects, the original names of local variables are intact within an expanded program, while lexical-context information in the syntax object relates binding occurrences to bound uses. Another advantage is that various language extensions for manipulating syntax objects in macro transformers – notably the `syntax-case` form that gives the macro system its name – are also available for use by tools that process expanded programs.

Syntax objects thus serve as an intermediate representation of programs for all Racket tools, whether they simply inspect the program (as in the syntax checker, to show the program's binding structure via arrows overlaid on the source text) or transform the program (as in the profiler, to add instrumentation). To allow the latter, in particular, the output of the `expand` function must also be a suitable input to `expand`, and `expand` must be idempotent. Then a program transformer can introduce code into an expanded program and pass it to `eval`, which will re-expand the program – potentially expanding forms that were introduced by the transformer but leaving the previously expanded code intact.

3 Modeling macro expansion in Racket

This section builds up a formal model of Racket macro expansion. We build on a traditional Lisp perspective instead of assuming previous Scheme models as background. In part, this strategy is aimed at making the presentation as widely accessible as possible, but it also lets us adjust and simplify some core representation and expansion details for Scheme-style macros.

We begin with a core functional language without macros. We then create a surface language and add syntax objects representing terms in the surface language to the set of core-language values. We progressively extend the model with naive macros, macros with proper lexical scoping, and macros that communicate.

We use the following terminology to describe relationships between different components. The *reader* consumes a surface program in textual form and produces its representation as a syntax object. That representation is recursively *expanded* until all macros have been eliminated; the result is a “fully-expanded” syntax object. Finally, the fully expanded syntax object is *parsed* into a core-language AST, which may be evaluated.

The sequence of models is implemented and typeset using PLT Redex (Felleisen *et al.*, 2009). The sources are available as supplementary material online at http:dx.doi.org/10.1017/S0956796912000093

3.1 Core language

The core language of our model includes variables, function applications tagged with **APP**, and values. Values include functions formed with **FUN**, lists formed with **LIST**, symbols formed with a curly quote, primitive operations, and possibly other kinds of data,

$$
\begin{aligned}
\mathit{ast} &::= \mathit{var} \mid \mathbf{APP}(\mathit{ast}, \mathit{ast}, ...) \mid \mathit{val} \\
\mathit{var} &::= \mathbf{VAR}(\mathit{name}) \\
\mathit{val} &::= \mathbf{FUN}(\mathit{var}, \mathit{ast}) \mid \mathbf{LIST}(\mathit{val}, ...) \mid \mathit{atom} \\
\mathit{atom} &::= \mathit{sym} \mid \mathit{prim} \mid .... \\
\mathit{sym} &::= {'}\mathit{name} \\
\mathit{prim} &::= \mathbf{cons} \mid \mathbf{car} \mid \mathbf{cdr} \mid \mathbf{list} \mid .... \\
\mathit{name} &::= \text{a token such as } \texttt{x}, \texttt{egg}, \text{or } \texttt{lambda}
\end{aligned}
$$

We represent a *var* as a *name* wrapped with a **VAR** constructor. We use a **VAR** constructor to help distinguish names in general from names that are used to represent variables, since names are also used in the representation of symbols.

Primitive operations are treated as literals (written in boldface) for simplicity. For example, the term **cons** is the actual primitive operation itself, not a variable whose value is the operation. Primitive operations are applied using the same **APP** form as for applying functions, so **APP** allows multiple argument expressions, even though a **FUN** accepts only a single argument.

Evaluation of core language is standard (using substitution for functions):

$$
\begin{aligned}
\mathsf{eval}[\![\mathbf{APP}(\mathbf{FUN}(\mathit{var}, \mathit{ast}_{body}), \mathit{ast}_{arg})]\!] &= \mathsf{eval}[\![\mathit{ast}_{body}[\mathit{var} \leftarrow \mathsf{eval}[\![\mathit{ast}_{arg}]\!]]]\!] \\
\mathsf{eval}[\![\mathbf{APP}(\mathit{prim}, \mathit{ast}_{arg}, ...)]\!] &= \delta(\mathit{prim}, \mathsf{eval}[\![\mathit{ast}_{arg}]\!], ...) \\
\mathsf{eval}[\![\mathbf{APP}(\mathit{ast}_{op}, \mathit{ast}_{arg}, ...)]\!] &= \mathsf{eval}[\![\mathbf{APP}(\mathsf{eval}[\![\mathit{ast}_{op}]\!], \mathit{ast}_{arg}, ...)]\!] \\
\mathsf{eval}[\![\mathit{val}]\!] &= \mathit{val}
\end{aligned}
$$

The second case of eval defers the implementation of primitives to a δ relation, which covers at least the primitive operations on lists:

$$
\begin{aligned}
\delta(\mathbf{cons}, \mathit{val}_1, \mathbf{LIST}(\mathit{val}_2, ...)) &= \mathbf{LIST}(\mathit{val}_1, \mathit{val}_2, ...) \\
\delta(\mathbf{car}, \mathbf{LIST}(\mathit{val}_1, \mathit{val}_2, ...)) &= \mathit{val}_1 \\
\delta(\mathbf{cdr}, \mathbf{LIST}(\mathit{val}_1, \mathit{val}_2, ...)) &= \mathbf{LIST}(\mathit{val}_2, ...) \\
\delta(\mathbf{list}, \mathit{val}, ...) &= \mathbf{LIST}(\mathit{val}, ...) \\
&\ldots
\end{aligned}
$$

The language can contain other primitive operations, such as **+**, which are also given meaning through δ.

3.2 Syntax objects

The core language serves two roles: It is the target language into which a surface program is parsed, and it is also the language for implementing macro transformers. Consequently, the values of the core language must include representations of the surface-language fragments that macro transformers manipulate, that is, syntax objects.

To model syntax objects, we extend the core language's values with syntax objects and primitive operations on syntax objects:

$$\begin{aligned} val &::= .... \mid stx \\ prim &::= .... \mid \mathtt{\mathbf{stx\text{-}e}} \mid \mathtt{\mathbf{mk\text{-}stx}} \end{aligned}$$

The new primitive **`stx-e`** is short for Racket's `syntax-e`, and **`mk-stx`** is short for `make-syntax`.

Syntax objects, tagged with $\textbf{S}\small{\textbf{TX}}$ combine a value with lexical-context information *ctx*. The value must be either an atom or a list of syntax objects. We introduce lexical-context information later, and for now just use • for *ctx*.

$$\begin{aligned} stx &::= \textbf{Stx}(atom, ctx) \mid \textbf{Stx}(\textbf{List}(stx, ...), ctx) \\ id &::= \textbf{Stx}(sym, ctx) \\ ctx &::= \bullet \end{aligned}$$

The set of identifiers *id* is a subset of *stx* consisting of only those syntax objects that wrap a symbol.

3.2.1 Names, variables, symbols, and identifiers

The terms *name*, *variable*, *symbol*, and *identifier* are easily confused, but we use each term in a specific way. To recap,

- A *name*, such as `x`, is a member of some abstract set of tokens in the meta-language (i.e., a "meta-symbol" in the implementing language, as opposed to a symbol in the implemented language). Names are used in the representation of variables and symbols.
- A *variable*, such as (**VAR** `x`), is the formal argument of a function, or it is a reference to a function argument that is replaced by a value during evaluation. Variables appear only in ASTs.
- A *symbol*, such as `'x`, is a value during evaluation. A symbol can appear as a literal expression, but since a symbol is constructed using a curly quote, it is never mistaken for a variable and replaced with a value.
- An *identifier*, such as (**STX** `'x` •), is a symbol combined with a lexical context. Like a symbol, an identifier is a value during evaluation – especially during the evaluation of macro transformers.

A Lisp programmer may be tempted to think of variables as implemented with symbols. Indeed, when an interpreter is implemented in Lisp, a variable or symbol in the interpreted language is typically represented using a symbol in the interpreter. Our eval, in contrast, is a mathematical function; variables and symbols are therefore implemented by names, which are entities in the mathematical world where the eval function resides. We highlight the distinction between language and meta-language to clarify the concepts that are inherently connected within the language (e.g., symbols and identifiers), and that are related only by a representation choice in the meta-language (e.g., symbols and variables, both as names).

3.2.2 Readers and syntax objects

A *reader* consumes a textual representation of a surface program and produces a corresponding syntax object. For example, the reader would convert the source program

```
(lambda x x)
```

into its representation as a syntax object,

$$\textbf{STX}(\textbf{LIST}(\textbf{STX}(\texttt{'lambda}, \bullet), \textbf{STX}(\texttt{'x}, \bullet), \textbf{STX}(\texttt{'x}, \bullet)), \bullet)$$

We do not model the reader process that takes a sequence of characters for a source program and converts it into a value that represents the source; we work only with the syntax-object representation.

The following extension of δ models the new primitive **stx-e** and **mk-stx** operations on syntax objects:

$$\begin{aligned}
\delta(\textbf{stx-e}, \textbf{STX}(val, ctx)) &= val \\
\delta(\textbf{mk-stx}, atom, \textbf{STX}(val, ctx)) &= \textbf{STX}(atom, ctx) \\
\delta(\textbf{mk-stx}, \textbf{LIST}(stx, ...), \textbf{STX}(val, ctx)) &= \textbf{STX}(\textbf{LIST}(stx, ...), ctx) \\
\ldots &
\end{aligned}$$

That is, **stx-e** unwraps a syntax object by throwing away its immediate context, while **mk-stx** constructs a new syntax object by borrowing the context from an existing syntax object (which might have been a literal **STX** value in the original program or might itself have been constructed with **mk-stx**).

For example,

$$\begin{aligned}
&\mathsf{eval}[\![\textbf{APP}(\textbf{stx-e}, \textbf{APP}(\textbf{mk-stx}, \texttt{'x}, \textbf{STX}(\texttt{'y}, \bullet)))]\!] \\
&= \delta(\textbf{stx-e}, \mathsf{eval}[\![\textbf{APP}(\textbf{mk-stx}, \texttt{'x}, \textbf{STX}(\texttt{'y}, \bullet))]\!]) \\
&= \delta(\textbf{stx-e}, \delta(\textbf{mk-stx}, \texttt{'x}, \textbf{STX}(\texttt{'y}, \bullet))) \\
&= \delta(\textbf{stx-e}, \textbf{STX}(\texttt{'x}, \bullet)) \\
&= \texttt{'x}
\end{aligned}$$

3.2.3 Model vs. implementations

The core model's **FUN** AST form is close to `lambda` in Scheme, and the *sym* representation is similar to a quoted symbol in Scheme. The model's **list** primitive operation is analogous to a `list` function, while a **LIST** constant is more like a quoted list in Scheme. For example, the model AST

$$\textbf{LIST}(\texttt{'lambda}, \textbf{LIST}(\texttt{'x}), \texttt{'y})$$

is analogous to the Scheme expression

```
'(lambda (x) y)
```

Along the same lines, an **Stx** literal in the model AST is analogous to a syntax-quoted form in Scheme. For example,

Stx(**List**(**Stx**('`lambda`, •), **Stx**(**List**(**Stx**('`x`, •)), •), **Stx**('`y`, •)), •)

is analogous to

```
#'(lambda (x) y)
```

where `#'` is a shorthand for a `syntax` form in the same way that `'` is a shorthand for a `quote` form. Note that in Racket, the printed form of a syntax object reports its source location (if any) and the encapsulated expression text:

```
> #'(lambda (x) y)
#<syntax:1:0 (lambda (x) y)>
```

The **`stx-e`** model primitive corresponds to the `syntax-e` function in Racket, and **`mk-stx`** in the model is similar to `datum->syntax` with its arguments reversed:

```
> (syntax-e (datum->syntax #'y 'x))
'x
```

Applying `syntax-e` to a complex syntax object exposes pieces that might be manipulated with `car` and `cdr`. Often such pieces are reassembled with `datum->syntax`:

```
> (define stx #'(fun x y))

> (syntax-e stx)
'(#<syntax:1:0 fun> . #<syntax:1:0 (x y)>)
> (datum->syntax stx
                 (cons #'lambda
                       (cdr (syntax-e stx))))
#<syntax (lambda x y)>
```

The model is simpler and more primitive than Racket and Scheme in several ways. The `datum->syntax` function recurs into a list whose elements are not syntax objects, which is why the model's non-recurring **`mk-stx`** has a different name. In Racket, the core form without pattern variables is `quote-syntax`, and `syntax` expands to `quote-syntax` for literal program fragments. Standard `syntax-case` systems do not include Racket's `syntax-e` operation, although it is essentially the expose function from Dybvig *et al.* (1993); instead, the built-in pattern-matching notation is used to deconstruct syntax objects. The `syntax->datum` operation, meanwhile, recursively applies `syntax-e`, discarding lexical-context information on both the immediate syntax object and nested syntax objects.

Not all implementations of `syntax-case` associate a lexical context to a list or number. In Racket, consistently associating a lexical context to every program fragment gives the programmer control over the expansion of constants and application forms. Such control is beyond the scope of this paper, but our model is intended to accommodate those extensions that are used heavily in the implementation of Racket (e.g., to support functions with keyword arguments).

3.3 Parsing

For our purposes, we define *parsing* as the task of converting a syntax object to an AST that can be evaluated. We define a parser for a Scheme-like language as follows:

- A `lambda` form is parsed into a **FUN** *ast* node. Unlike in Scheme, `lambda` allows only a single argument and omits a set of parentheses around the argument.
- All literal values, even primitive operations (written in boldface), must be `quote`d in a source program; the quoted literals are parsed as *atom*s.
- A `syntax` form is parsed into an *stx* value (without support for pattern variables).
- A sequence of expressions grouped with parentheses is parsed as an **APP** node when the first element of the group is not the name of a primitive syntactic form (such as `lambda` or `quote`) or a macro.
- An identifier as an expression is parsed as a *var*.

For example, a function that accepts a single number argument to increment would be written in the surface language as

```
(lambda x (' + x '1))
```

which the reader converts to the *stx*

$$
\begin{array}{l}
\textbf{STX}(\textbf{LIST}(\textbf{STX}(\texttt{'lambda}, \bullet),\\
\qquad\qquad \textbf{STX}(\texttt{'x}, \bullet),\\
\qquad\qquad \textbf{STX}(\textbf{LIST}(\textbf{STX}(\textbf{LIST}(\textbf{STX}(\texttt{'quote}, \bullet),\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{STX}(\mathbf{+}, \bullet)), \bullet),\\
\qquad\qquad\qquad\qquad \textbf{STX}(\texttt{'x}, \bullet),\\
\qquad\qquad\qquad\qquad \textbf{LIST}(\textbf{STX}(\texttt{'quote}, \bullet),\\
\qquad\qquad\qquad\qquad\qquad \textbf{STX}(1, \bullet))),\\
\qquad\qquad\qquad \bullet)),\\
\qquad \bullet)
\end{array}
$$

and the job of the parser is to convert this *stx* to the *ast*

$$\textbf{FUN}(\textbf{VAR}(\texttt{x}), \textbf{APP}(\mathbf{+}, \textbf{VAR}(\texttt{x}), 1))$$

3.3.1 Symbol-driven parser

Ignoring macros, and also assuming that keywords like `lambda` are never shadowed, we could implement a parser from *stx*es to *ast*s with the following parse meta-function:

$$
\begin{array}{lcl}
\mathsf{parse}[\![\textbf{STX}(\textbf{LIST}(\textbf{STX}(\texttt{'lambda}, \bullet), \textbf{STX}(\texttt{'}\mathit{name}, \bullet), \mathit{stx}), \bullet)]\!] & = & \textbf{FUN}(\textbf{VAR}(\mathit{name}), \mathsf{parse}[\![\mathit{stx}]\!])\\
\mathsf{parse}[\![\textbf{STX}(\textbf{LIST}(\textbf{STX}(\texttt{'quote}, \bullet), \mathit{stx}), \bullet)]\!] & = & \mathsf{strip}[\![\mathit{stx}]\!]\\
\mathsf{parse}[\![\textbf{STX}(\textbf{LIST}(\textbf{STX}(\texttt{'syntax}, \bullet), \mathit{stx}), \bullet)]\!] & = & \mathit{stx}\\
\mathsf{parse}[\![\textbf{STX}(\textbf{LIST}(\mathit{stx}_{rator}, \mathit{stx}_{rand}, \ldots), \bullet)]\!] & = & \textbf{APP}(\mathsf{parse}[\![\mathit{stx}_{rator}]\!], \mathsf{parse}[\![\mathit{stx}_{rand}]\!], \ldots)\\
\mathsf{parse}[\![\textbf{STX}(\mathit{name}, \bullet)]\!] & = & \textbf{VAR}(\mathit{name})
\end{array}
$$

The clauses to define meta-functions in this paper are ordered so that the next-to-last clause of parse produces an **App** form when the initial identifier in a sequence is not `lambda`, `quote`, or `syntax`.

The parse function uses a strip meta-function to implement `quote` by stripping away lexical context:

$$\begin{aligned}
\mathsf{strip}[\![\mathbf{Stx}(\mathit{atom}, \mathit{ctx})]\!] &= \mathit{atom} \\
\mathsf{strip}[\![\mathbf{Stx}(\mathbf{List}(\mathit{stx}, ...), \mathit{ctx})]\!] &= \mathbf{List}(\mathsf{strip}[\![\mathit{stx}]\!], ...)
\end{aligned}$$

The difference between a `quote` form and a `syntax` form is that the latter does not strip lexical context from the input representation.

3.3.2 Identifier-driven parser

When we add lexical-context information to *stx* (instead of just using $\bullet$), parse will need to take that information into account instead of simply looking for identifiers named `lambda`, `quote`, and `syntax`. To prepare for that change, we refine parse as follows, deferring identifier resolution to a resolve meta-function. For now, resolve simply extracts the *name* in an identifier, but we will refine it later to use the lexical-context information of an identifier.[5]

$$\mathsf{resolve}[\![\mathbf{Stx}(\text{'}\mathit{name}, \bullet)]\!] = \mathit{name}$$

$$\begin{aligned}
&\mathsf{parse}[\![\mathbf{Stx}(\mathbf{List}(\mathit{id}_{lambda}, \mathit{id}_{arg}, \mathit{stx}_{body}), \mathit{ctx})]\!] &&= \mathbf{Fun}(\mathbf{Var}(\mathsf{resolve}[\![\mathit{id}_{arg}]\!]), \mathsf{parse}[\![\mathit{stx}_{body}]\!]) \\
&\quad \text{where } \mathtt{lambda} = \mathsf{resolve}[\![\mathit{id}_{lambda}]\!] && \\
&\mathsf{parse}[\![\mathbf{Stx}(\mathbf{List}(\mathit{id}_{quote}, \mathit{stx}), \mathit{ctx})]\!] &&= \mathsf{strip}[\![\mathit{stx}]\!] \\
&\quad \text{where } \mathtt{quote} = \mathsf{resolve}[\![\mathit{id}_{quote}]\!] && \\
&\mathsf{parse}[\![\mathbf{Stx}(\mathbf{List}(\mathit{id}_{syntax}, \mathit{stx}), \mathit{ctx})]\!] &&= \mathit{stx} \\
&\quad \text{where } \mathtt{syntax} = \mathsf{resolve}[\![\mathit{id}_{syntax}]\!] && \\
&\mathsf{parse}[\![\mathbf{Stx}(\mathbf{List}(\mathit{stx}_{rator}, \mathit{stx}_{rand}, ...), \mathit{ctx})]\!] &&= \mathbf{App}(\mathsf{parse}[\![\mathit{stx}_{rator}]\!], \mathsf{parse}[\![\mathit{stx}_{rand}]\!], ...) \\
&\mathsf{parse}[\![\mathit{id}]\!] &&= \mathbf{Var}(\mathsf{resolve}[\![\mathit{id}]\!])
\end{aligned}$$

The parse meta-function in our model serves the same role as the parse meta-function in the model of Dybvig *et al.* (1993). Unlike the *Dybvig et al.* (1993) model, where parse is mutually recursive with an expand meta-function, our parse function works only on fully expanded terms, and we define a separate expansion process that both consumes and produces a syntax object. This difference paves the way for sub-form expansion (which must expand without parsing), and it also reflects the use of syntax objects as a general-purpose intermediate format in Racket (as discussed in Section 2.6).

3.4 Expansion

The next step to modeling Scheme macro expansion is to create an *expander* that takes a syntax object for a source program and returns a syntax object for the expanded program.

[5] Notation: We use "where" in the parse meta-function to write side conditions that more conventionally would be written with "if," whereas "where" more conventionally binds meta-variables. In later meta-functions, we use "where" as in PLT Redex to generalize and unify conventional "where" and "if" clauses. That is, "where" is a pattern-matching form that binds italicized meta-variables, and it also acts as a side condition by requiring a match.

The expander sits between the reader and the parser so that it starts with a syntax object that may have macro definitions and uses, and it produces a syntax object that fits the limited shape of syntax objects that are recognized by the parser. In addition to recognizing macro definitions and uses, the expander will have to recognize all of the forms that the parser recognizes; it nevertheless defers the production of an AST to the parser so that the result of the expander can be used for further expansion in some contexts.

Even without introducing macros, the expander has a role in preparing a source program: The parse meta-function assumes that a `lambda` identifier always indicates a function form, but we want our source language to be like Scheme, where any identifier can be used as a local variable name – even `lambda`. The expander therefore must rename formal arguments of a function to ensure that they do not shadow the identifiers that parse uses as key words.

The expander is implemented as an expand meta-function. To handle shadowing, and eventually to handle macro bindings, a compile-time environment ξ is provided to each use of expand. This environment maps *name*s to *transform*s, and expand normally starts with an environment that maps `lambda` to the FUN transform, `quote` to the QUOTE transform, and `syntax` also to the QUOTE transform. (The parse meta-function treats `quote` and `syntax` differently, but they turn out to be the same at the level of expand.) A *transform* can also be an identifier tagged with VAR, which represents a variable bound by an enclosing function,

$$
\begin{array}{l}
\xi ::= \text{a mapping from } \mathit{name} \text{ to } \mathit{transform} \\
\mathit{transform} ::= \textsc{Fun} \mid \textsc{Quote} \mid (\textsc{Var}\ \mathit{id})
\end{array}
$$

Each case for expand is similar to a corresponding case in parse, except that `quote` and `syntax` are collapsed into a single case:

$$
\begin{array}{ll}
\text{expand}[\![\mathbf{STX}(\mathbf{LIST}(id_{lam}, id_{arg}, stx_{body}), ctx), \xi]\!] & = \mathbf{STX}(\mathbf{LIST}(id_{lam}, id_{new}, stx_{expbody}), ctx) \\
\quad \text{where } \textsc{Fun} = \xi(\text{resolve}[\![id_{lam}]\!]),\ name_{new} = \text{fresh},\ id_{new} = \mathbf{STX}(\text{'}name_{new}, \bullet), & \\
\qquad\quad \xi_{new} = \xi + \{\text{resolve}[\![id_{arg}]\!] \rightarrow (\textsc{Var}\ id_{new})\},\ stx_{expbody} = \text{expand}[\![stx_{body}, \xi_{new}]\!] & \\
\text{expand}[\![\mathbf{STX}(\mathbf{LIST}(id_{quote}, stx), ctx), \xi]\!] & = \mathbf{STX}(\mathbf{LIST}(id_{quote}, stx), ctx) \\
\quad \text{where } \textsc{Quote} = \xi(\text{resolve}[\![id_{quote}]\!]) & \\
\text{expand}[\![\mathbf{STX}(\mathbf{LIST}(stx_{rator}, stx_{rand}, \ldots), ctx), \xi]\!] & = \mathbf{STX}(\mathbf{LIST}(stx_{exprator}, stx_{exprand}, \ldots), ctx) \\
\quad \text{where } stx_{exprator} = \text{expand}[\![stx_{rator}, \xi]\!],\ stx_{exprand}\ \ldots = \text{expand}[\![stx_{rand}, \xi]\!]\ \ldots & \\
\text{expand}[\![id, \xi]\!] & = id_{new} \\
\quad \text{where } (\textsc{Var}\ id_{new}) = \xi(\text{resolve}[\![id]\!]) &
\end{array}
$$

A significant difference from parse is that the `lambda` case of expand generates a new name for the formal argument in a `lambda` form, which ensures that the expanded program does not use any parse-recognized names as variables. The `lambda` case maps the original name to the new one in the environment for the `lambda` form's body. Correspondingly, the case in expand for expanding a variable reference installs the new name in place of the original, which it finds by consulting the environment.

As an example, the source

```
(lambda lambda lambda)
```

expands to the identity function essentially as follows:

$$
\begin{aligned}
&\mathsf{expand}[\![\texttt{(lambda lambda lambda)}, \xi_0]\!] \\
&= \texttt{(lambda lambda2 } \mathsf{expand}[\![\texttt{lambda}, \xi_0\texttt{+\{lambda}\to\texttt{lambda2\}}]\!]\texttt{)} \\
&= \texttt{(lambda lambda2 lambda2)}
\end{aligned}
$$

To make the expansion trace above more readable, identifiers are reduced to their resolve results, lexical-context information is dropped, ξ_0 stands for the initial environment, and other obvious simplifications are applied.

3.5 Binding and using macros

To support macros, we extend the source language with a `let-syntax` form that is a simplified version of Scheme's macro-binding forms. Our `let-syntax` will bind a single identifier to a macro transformer function for use in the `let-syntax` body expression. For example, the following source program defines and uses a `thunk` macro to delay evaluation of an expression until it is applied to a (dummy) argument:

```
(let-syntax
  thunk (lambda e
          (′ mk-stx
           (′ list  (syntax lambda) (syntax a)
                (′ car  (′ cdr  (′ stx-e e))))
           e))
   ((thunk (′ +  ′1 ′2)) ′0))
```

The `e` argument to the macro transformer is the representation of the use of the macro `(thunk (′ + ′1 ′2))`. The transformer extracts the `(′ + ′1 ′2)` sub-expression from this representation using **`stx-e`**, **`cdr`**, and **`car`** on `e`. The transformer then places the sub-expression into a `lambda` expression using **`list`** and **`mk-stx`**, producing a representation of `(lambda a (′ + ′1 ′2))`.

Support for macros in the expander requires a new LET-SYNTAX transform to serve as a binding for `let-syntax`. Furthermore, the expansion of a `let-syntax` form binds an identifier to a compile-time value:

$$\textit{transform} ::= \ldots.\ \mid (\textsc{Var}\ \textit{id}) \mid \textit{val}$$

The expander needs new cases for evaluating core-form expressions during the process of expansion. No changes are needed to *ast* or parse to support macros, however, since the expander eliminates all uses of macros. The new expander cases include all of the old cases, plus cases for macro bindings and macro applications. The macro-binding case implements the new LET-SYNTAX transform:

$$
\begin{aligned}
&\mathsf{expand}[\![\textbf{STX}(\textbf{LIST}(id_{ls}, id_{mac}, stx_{rhs}, stx_{body}), ctx), \xi]\!] = \mathsf{expand}[\![stx_{body}, \xi_1]\!] \\
&\quad \text{where } \textsc{Let-Syntax} = \xi(\mathsf{resolve}[\![id_{ls}]\!]), \\
&\qquad\quad \xi_1 = \xi\texttt{+}\{\mathsf{resolve}[\![id_{mac}]\!] \to \mathsf{eval}[\![\mathsf{parse}[\![stx_{rhs}]\!]]\!]\}
\end{aligned}
$$

In this case, to evaluate the right-hand side of a `let-syntax` form, the right-hand side is first parsed. Using parse directly reflects the fact that this model does not cover macro transformers that are implemented in terms of macros (except that a macro expansion can include uses of macros).[6] The parsed right-hand side is then evaluated, and the result is bound in the compile-time environment while the `let-syntax` body is expanded.

The case for a macro application is triggered when the compile-time environment maps a name to a function value. Invocation of the macro applies the value from the environment to the macro-use source form. After the macro produces a value (which must be a syntax object), the expander is again applied to the result.

$$\begin{aligned}&\mathsf{expand}[\![stx_{macapp}, \xi]\!] = \mathsf{expand}[\![\mathsf{eval}[\![\textbf{APP}(val, stx_{macapp})]\!], \xi]\!]\\ &\quad \text{where } \textbf{STX}(\textbf{LIST}(id_{mac}, stx_{arg}, ...), ctx) = stx_{macapp},\\ &\qquad val = \xi(\mathsf{resolve}[\![id_{mac}]\!])\end{aligned}$$

Since we have not yet added lexical-context information to syntax objects, the macro system at this point resembles a traditional Lisp `defmacro` system. For example, using the `thunk` macro as defined above, the expression

```
(((lambda a (thunk ('+ a '1))) '5) '0)
```

produces `1` instead of `6` because the `a` binding introduced by the `thunk` macro captures `a` in the expression supplied to `thunk`. That is, the `thunk` macro does not respect the lexical scope of the original program. The expander produces this result for the `lambda` form roughly as follows, in an environment ξ_0 that maps `thunk` to the transformer:

$$\begin{aligned}&\mathsf{expand}[\![\texttt{(lambda a (thunk ('+ a '1)))}, \xi_0]\!]\\ &= \texttt{(lambda a2 } \mathsf{expand}[\![\texttt{(thunk ('+ a '1))}, \xi_1 = \xi_0 + \{\texttt{a} \rightarrow \texttt{a2}\}]\!]\texttt{)}\\ &= \textit{... calling the } \texttt{thunk} \textit{ transformer ...}\\ &= \texttt{(lambda a2 } \mathsf{expand}[\![\texttt{(lambda a ('+ a '1))}, \xi_1]\!]\texttt{)}\\ &= \texttt{(lambda a2 (lambda a3 } \mathsf{expand}[\![\texttt{('+ a '1)}, \xi_2 = \xi_1 + \{\texttt{a} \rightarrow \texttt{a3}\}]\!]\texttt{))}\\ &= \textit{... expanding the body, no more extensions to } \xi_2 \textit{ ...}\\ &= \texttt{(lambda a2 (lambda a3 ('+ a3 '1)))}\end{aligned}$$

3.6 Tracking lexical context

To change the macro system so that macro transformers respect lexical scope, we introduce lexical-context information into syntax objects.

3.6.1 Scope examples

The first challenge in tracking binding through macro expansion is illustrated by the following example:

```
((lambda x
   (let-syntax m (lambda stx (syntax x))
     (lambda x
       ('+ (m) x))))
 1)
```

[6] Although expand could be applied to transfomer expressions using the current compile-time environment as in Dybvig *et al.* (1993), doing so mixes binding phases in a way that is not true to Racket or allowed by the current Scheme standard (Sperber, 2009). The model is instead easily generalized to support expansion of transformer expressions through modules and phases (Flatt, 2002).

The expansion of (m) carries a reference to the outer x into the scope of the inner x. Proper lexical scoping demands that the two xs are kept distinct.

At first glance, the solution is simply to capture the compile-time environment ξ in either the m binding, the (lambda stx (syntax x)) closure, or the (syntax x) syntax object. That way, when the x that is introduced by the expansion of m is further expanded, the captured environment is used instead of the current compile-time environment. The captured environment then correctly maps x to the binding from the outer lambda.

Although the intuition is appealing, a simple environment-capturing approach does not work in general, because identifiers introduced by a macro expansion can appear in binding positions as well as use positions. For example, in

```
(lambda x
  (let-syntax n (lambda stx
                  ; expand (n e) to (lambda x (' + e x))
                  (' mk-stx
                   (' list (syntax lambda) (syntax x)
                        (' mk-stx
                         (' list (syntax ' +)
                              (' car (' cdr (' stx-e x)))
                              (syntax x))
                            stx))
                   stx))
    (n '1)))
```

the expansion of (n '1) is (lambda x (' + '1 x)). If the last x simply carried a compile-time environment from its source (syntax x) expression, then x would refer to the outermost x binding instead of the one bound by the new lambda in the expansion of (n '1).

The difference between the (n '1) and (m) examples is that (m) introduces x *after* the lambda that should bind x has been expanded, while (n '1) introduces x *before* the lambda that should bind x is expanded. More generally, lambda and let-syntax forms can nest arbitrarily, and macros can expand to definitions of macros so that identifier bindings and introductions can be interleaved arbitrarily. This combination of *local macros* and *macro-generating macros* defeats a simple capturing of the compile-time environment to bind macro-introduced identifiers.

We can more easily account for identifier binding by renaming identifiers in a syntax object instead of trying to delay the substitution through an environment. That is, whenever the expander encounters a core binding form like lambda, it applies a renaming to the syntax object instead of merely recording the binding in the compile-time environment. When the expander encounters the first lambda in the example containing (m), it renames the binding to x_1:

```
((lambda x1
  (let-syntax m (lambda stx (syntax x1))
    (lambda x1
      (' + (m) x1))))
 1)
```

The macro-binding `m` is similarly renamed to $\texttt{m}_1$. When the expander later encounters the inner `lambda`, it renames $\texttt{x}_1$ further to $\texttt{x}_2$.

```
(lambda x₂
  ('+ (m₁) x₂))
```

Since $\texttt{x}_1$ is renamed $\texttt{x}_2$ only within the inner `lambda` form, $(\texttt{m}_1)$ expands to a use of $\texttt{x}_1$, which still refers to the outer `lambda` binding.

The example with `(n '1)` works similarly, where the outer `lambda`'s binding is renamed to $\texttt{x}_1$, along with all instances of `x` quoted as syntax in the `n` transformer. The expansion of $(\texttt{n}_1\ \texttt{'1})$ is then $(\texttt{lambda}\ \texttt{x}_1\ (\texttt{'+}\ \texttt{'1}\ \texttt{x}_1))$ so that the macro-introduced $\texttt{x}_1$ is bound by the macro-introduced binding of $\texttt{x}_1$ – both of which will be immediately renamed by the expander to $\texttt{x}_2$.

Renaming is a step in the right direction, but it turns out to be only half of the story. Consider a variation of the `(n '1)` example with `x` in place of `'1`:

```
(lambda x
  (let-syntax n ....
    (n x)))
```

The `x` in `(n x)` should refer to the outer `lambda` binding. According to our story so far, renaming leads to $(\texttt{n}_1\ \texttt{x}_1)$, which expands to $(\texttt{lambda}\ \texttt{x}_1\ (\texttt{'+}\ \texttt{x}_1\ \texttt{x}_1))$, at which point the $\texttt{x}_1$ from $(\texttt{n}_1\ \texttt{x}_1)$ is inappropriately captured by the macro-introduced binding of $\texttt{x}_1$.

To avoid this kind of incorrect capture, Dybvig *et al.* (1993) build on the technique of Kohlbecker *et al.* (1986). The key is to track syntax objects that are newly introduced by a macro expansion versus syntax objects that were originally provided to the macro expansion. Specifically, the result of a macro transformer is *marked* in such a way that a mark sticks to parts of the expansion that were introduced by the macro, while parts that were present in the macro use are unmarked. Representing marks as superscripts, the expansion of $(\texttt{n}_1\ \texttt{x}_1)$ becomes $(\texttt{lambda}^2\ \texttt{x}_1{}^2\ (\texttt{'+}^2\ \texttt{x}_1\ \texttt{x}_1{}^2))$, since the `lambda`, binding $\texttt{x}_1$, `'+`, and last $\texttt{x}_1$ are all introduced by the macro expansion, while the next-to-last $\texttt{x}_1$ was present in the use $(\texttt{n}_1\ \texttt{x}_1)$.

Marks, as represented by superscripts, are not treated as a part of a name in the same way as renamings, as represented by subscripts. In particular, lookup in a compile-time environment ignores marks, so $\texttt{lambda}^2$ indicates FUN in the same way as `lambda`. Marks affect renamings, however: A renaming applies only to identifier uses that have the same current name and marks as the binding identifier. Thus, when the expander encounters $(\texttt{lambda}^2\ \texttt{x}_1{}^2\ (\texttt{'+}\ \texttt{x}_1\ \texttt{x}_1{}^2))$, it renames $\texttt{x}_1{}^2$ to $\texttt{x}_2$, leaving the unmarked $\texttt{x}_1$ alone so that the result is correctly $(\texttt{lambda}\ \texttt{x}_2\ (\texttt{'+}\ \texttt{x}_1\ \texttt{x}_2))$.

3.6.2 Marks and renames as lexical context

In the model, instead of subscripts and superscripts, marks and renames are attached to a syntax object through the lexical-context part of a syntax object. Renames are *not* implemented by changing the symbol within an identifier because the original symbol is needed if the identifier turns out to be quoted. For example, in

```
(lambda x
  'x)
```

the expander renames `x` to $\texttt{x}_1$, but the body of the `lambda` form should produce the symbol `'x`, not the symbol $\texttt{'x}_1$. Meanwhile, the expander cannot simply skip `quote` forms when renaming because some quoted forms may not become apparent until macro expansion is complete. By putting renaming information into the lexical-context part of an identifier, the original symbol is intact for quoting.

To support mark and rename information in lexical context, we add two productions to the grammar of *ctx*:

$$\begin{aligned} ctx &::= \bullet \mid \textsc{Mark}(ctx, mrk) \mid \textsc{Rename}(ctx, id, name) \\ mrk &::= name \end{aligned}$$

The MARK and RENAME constructors each build on an existing context. A MARK adds a fresh mark, where a mark is implemented as a name, although integers would work just as well. A RENAME record maps a particular identifier (with its own renamings and marks intact) to a fresh name.

The mark and rename meta-functions push MARK and RENAME records down to all *ctx* chains in a syntax object:

$$\begin{aligned}
&\mathsf{mark}[\![\mathbf{Stx}(atom, ctx), mrk]\!] &&= \mathbf{Stx}(atom, \textsc{Mark}(ctx, mrk)) \\
&\mathsf{mark}[\![\mathbf{Stx}(\mathbf{List}(stx, ...), ctx), mrk]\!] &&= \mathbf{Stx}(\mathbf{List}(\mathsf{mark}[\![stx, mrk]\!], ...), \\
& &&\qquad \textsc{Mark}(ctx, mrk)) \\
&\mathsf{rename}[\![\mathbf{Stx}(atom, ctx), id, name]\!] &&= \mathbf{Stx}(atom, \textsc{Rename}(ctx, id, name)) \\
&\mathsf{rename}[\![\mathbf{Stx}(\mathbf{List}(stx, ...), ctx), id, name]\!] &&= \mathbf{Stx}(\mathbf{List}(\mathsf{rename}[\![stx, id, name]\!], ...), \\
& &&\qquad \textsc{Rename}(ctx, id, name))
\end{aligned}$$

When a transformer expands a macro use, only syntax objects that were introduced by the macro should be marked, while syntax objects that were part of the macro use should remain unmarked. The technique of Dybvig *et al.* (1993) is to mark the input of a macro transformer using a fresh key, mark the result of the transformer again with the same key, and treat double marks as canceling each other. This canceling behavior is reflected in the marksof meta-function, which extracts the set of non-canceled marks from an identifier:

$$\begin{aligned}
&\mathsf{marksof}[\![\mathbf{Stx}(sym, \bullet)]\!] &&= () \\
&\mathsf{marksof}[\![\mathbf{Stx}(sym, \textsc{Mark}(ctx, mrk))]\!] &&= mrk \oplus (mrk_2\ ...) \\
&\quad \text{where } (mrk_2\ ...) = \mathsf{marksof}[\![\mathbf{Stx}(sym, ctx)]\!] && \\
&\mathsf{marksof}[\![\mathbf{Stx}(sym, \textsc{Rename}(ctx, id_2, name_2))]\!] &&= \mathsf{marksof}[\![\mathbf{Stx}(sym, ctx)]\!]
\end{aligned}$$

$$\begin{aligned}
mrk_1 \oplus (mrk_1\ mrk_2\ ...) &= (mrk_2\ ...) \\
mrk_1 \oplus (mrk_2\ ...) &= (mrk_1\ mrk_2\ ...)
\end{aligned}$$

The marksof function only needs the *ctx* part of an identifier, but we define it on identifiers as a convenience.

Finally, the redefined resolve meta-function traverses a *ctx* to interpret marks and renamings. The crucial clause in resolve handles a RENAME record, which renames if the source identifier of the rename is consistent with the resolution of the rest of the *ctx*. The two are consistent when they correspond to the same name after nested renamings and have the same set of marks:

$$
\begin{aligned}
&\mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, \bullet)]\!] &&= name\\
&\mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, \text{MARK}(ctx, mrk))]\!] &&= \mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, ctx)]\!]\\
&\mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, \text{RENAME}(ctx, id, name_{new}))]\!] &&= name_{new}\\
&\quad \text{where } name_1 = \mathsf{resolve}[\![id]\!], name_1 = \mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, ctx)]\!],\\
&\qquad \mathsf{marksof}[\![id]\!] = \mathsf{marksof}[\![\textbf{S}\text{TX}(\text{'}name, ctx)]\!]\\
&\mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, \text{RENAME}(ctx, id, name_2))]\!] &&= \mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, ctx)]\!]
\end{aligned}
$$

The resolve function otherwise ignores marks, which is why a macro introduced but never renamed `lambda`^{2},

$$\textbf{S}\text{TX}(\text{'}\texttt{lambda}, \text{MARK}(\bullet, mrk_2))$$

resolves the same as a plain `lambda`,

$$\textbf{S}\text{TX}(\text{'}\texttt{lambda}, \bullet)$$

Note that in the first RENAME case of resolve, when both *id* and (STX *name ctx*) resolve to $name_1$, and when $name_1$ is itself the result of renaming, then *id* and (STX *name ctx*) must have the same marks after the renaming – or else the renaming to $name_1$ would not apply. (Since the expander generates a fresh name for each renaming, any renaming to $name_1$ will be the same renaming, and hence it requires the same marks wherever it applies.) We can exploit this fact to implement a shortcut in marksof: If a renaming to a given $name_1$ is encountered, then ignore any remaining marks because the results for both identifiers will be the same.

To support the shortcut, a revised marksof accepts a traversal-stopping name, the last case of marksof is split into matching and non-matching cases for the name, and the third case of resolve changes to provide the name to marksof:

$$
\begin{aligned}
&\mathsf{marksof}[\![\textbf{S}\text{TX}(sym, \bullet), name]\!] &&= ()\\
&\mathsf{marksof}[\![\textbf{S}\text{TX}(sym, \text{MARK}(ctx, mrk)), name]\!] &&= mrk \oplus (mrk_2 \ldots)\\
&\quad \text{where } (mrk_2 \ldots) = \mathsf{marksof}[\![\textbf{S}\text{TX}(sym, ctx), name]\!]\\
&\mathsf{marksof}[\![\textbf{S}\text{TX}(sym, \text{RENAME}(ctx, id_2, name)), name]\!] &&= ()\\
&\mathsf{marksof}[\![\textbf{S}\text{TX}(sym, \text{RENAME}(ctx, id_2, name_2)), name]\!] &&= \mathsf{marksof}[\![\textbf{S}\text{TX}(sym, ctx), name]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\ldots\\
&\mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, \text{RENAME}(ctx, id, name_{new}))]\!] = name_{new}\\
&\quad \text{where } name_1 = \mathsf{resolve}[\![id]\!],\\
&\qquad name_1 = \mathsf{resolve}[\![\textbf{S}\text{TX}(\text{'}name, ctx)]\!],\\
&\qquad \mathsf{marksof}[\![id, name_1]\!] = \mathsf{marksof}[\![\textbf{S}\text{TX}(\text{'}name, ctx), name_1]\!]\\
&\ldots
\end{aligned}
$$

As it turns out, this shortcut particularly simplifies the implementation definition contexts, as explained later in Section 3.8.

3.6.3 Adapting the expander

With the machinery of marks and renames in place, we can adapt our `defmacro`-style macro model to a Scheme-style model by changing the macro-application, `lambda`, and `let-syntax` cases of expand.

A revised macro-application case for expand shows the before-and-after marking operations that track the parts of a syntax object that are introduced by a macro expansion:

$$\begin{array}{l} \mathsf{expand}[\![stx_{macapp}, \xi]\!] = \mathsf{expand}[\![\mathsf{mark}[\![stx_{exp}, mrk_{new}]\!], \xi]\!] \\ \quad \text{where } \textbf{S}\textbf{TX}(\textbf{L}\textbf{IST}(id_{mac}, stx_{arg}, ...), ctx) = stx_{macapp}, \\ \qquad val = \xi(\mathsf{resolve}[\![id_{mac}]\!]), mrk_{new} = \text{fresh}, \\ \qquad stx_{exp} = \mathsf{eval}[\![\textbf{A}\textbf{PP}(val, \mathsf{mark}[\![stx_{macapp}, mrk_{new}]\!])]\!] \end{array}$$

The revised `lambda` case generates a renaming for the formal argument of the function, and then it uses rename to apply the renaming to the body of the `lambda` form:

$$\begin{array}{l} \mathsf{expand}[\![\textbf{S}\textbf{TX}(\textbf{L}\textbf{IST}(id_{lam}, id_{arg}, stx_{body}), ctx), \xi]\!] = \textbf{S}\textbf{TX}(\textbf{L}\textbf{IST}(id_{lam}, id_{new}, stx_{expbody}), ctx) \\ \quad \text{where } \text{FUN} = \xi(\mathsf{resolve}[\![id_{lam}]\!]), name_{new} = \text{fresh}, \\ \qquad id_{new} = \mathsf{rename}[\![id_{arg}, id_{arg}, name_{new}]\!], \xi_{new} = \xi + \{name_{new} \rightarrow (\text{VAR } id_{new})\}, \\ \qquad stx_{expbody} = \mathsf{expand}[\![\mathsf{rename}[\![stx_{body}, id_{arg}, name_{new}]\!], \xi_{new}]\!] \end{array}$$

The environment is still extended to record that the generated name corresponds to a variable. More generally, the model uses lexical context information to represent the *identity* of bindings, but the compile-time environment still represents the *meanings* of bindings.

Since `let-syntax` introduces a local binding in the same sense as `lambda`, it must rename the local variable in the same way:

$$\begin{array}{l} \mathsf{expand}[\![\textbf{S}\textbf{TX}(\textbf{L}\textbf{IST}(id_{ls}, id, stx_{rhs}, stx_{body}), ctx), \xi]\!] = \mathsf{expand}[\![\mathsf{rename}[\![stx_{body}, id, name_{new}]\!], \xi_{new}]\!] \\ \quad \text{where } \text{LET-SYNTAX} = \xi(\mathsf{resolve}[\![id_{ls}]\!]), name_{new} = \text{fresh}, \\ \qquad \xi_{new} = \xi + \{name_{new} \rightarrow \mathsf{eval}[\![\mathsf{parse}[\![stx_{rhs}]\!]]\!]\} \end{array}$$

With the new expand cases, the `thunk` example of the previous section expands with proper handling of lexical scope:

$$\begin{array}{l} \quad \mathsf{expand}[\![(\texttt{lambda a (let-syntax} \\ \qquad\qquad\qquad \texttt{thunk (lambda e } \textbf{....}\ \textbf{S}\textbf{TX}(\texttt{a}, \bullet)\ \textbf{....}) \\ \qquad\qquad\qquad (\texttt{thunk } ('\texttt{+ } \textbf{S}\textbf{TX}(\texttt{a}, \bullet)\ '1)))), \\ \qquad \xi_0]\!] \\ = (\texttt{lambda } \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(\bullet, \texttt{a}, \texttt{a2})) \\ \quad \mathsf{expand}[\![(\texttt{let-syntax} \\ \qquad\qquad \texttt{thunk (lambda e } \textbf{....}\ \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(\bullet, \texttt{a}, \texttt{a2}))\ \textbf{....}) \\ \qquad\qquad (\texttt{thunk } ('\texttt{+ } \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(\bullet, \texttt{a}, \texttt{a2}))\ '1))), \\ \qquad \xi_1 = \xi_0 + \{\texttt{a2} \rightarrow \text{VAR}\}]\!]) \\ = \textit{... evaluating the } \texttt{thunk} \textit{ binding ...} \\ = (\texttt{lambda } \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(\bullet, \texttt{a}, \texttt{a2})) \\ \quad \mathsf{expand}[\![(\texttt{thunk } ('\texttt{+ } \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(\bullet, \texttt{a}, \texttt{a2}))\ '1)), \\ \qquad \xi_2 = \xi_1 + \{\texttt{thunk} \rightarrow \textbf{....}\}]\!]) \\ = \textit{... calling the } \texttt{thunk} \textit{ transformer ...} \\ \quad \textit{the macro-introduced } \texttt{a} \textit{ has the marked context ctx} \\ = (\texttt{lambda } \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(\bullet, \texttt{a}, \texttt{a2})) \\ \quad \mathsf{expand}[\![(\texttt{lambda } \textbf{S}\textbf{TX}(\texttt{a}, ctx, =, \text{MARK}(\text{RENAME}(\bullet, \texttt{a}, \texttt{a2}), mrk_1)) \\ \qquad\quad ('\texttt{+ } \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(\bullet, \texttt{a}, \texttt{a2}))\ '1)), \\ \qquad \xi_2]\!]) \\ \quad \textit{and the rename to } \texttt{a3} \textit{ applies to } \texttt{a} \textit{ with marked context ctx} \\ = (\texttt{lambda } \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(\bullet, \texttt{a}, \texttt{a2})) \\ \quad (\texttt{lambda } \textbf{S}\textbf{TX}(\texttt{a}, \text{RENAME}(ctx, \textbf{S}\textbf{TX}(\texttt{a}, ctx), \texttt{a3})) \\ \qquad \mathsf{expand}[\![('\texttt{+ } id_{body}\ '1), \xi_3 = \xi_2 + \{\texttt{a3} \rightarrow \text{VAR}\}]\!])) \end{array}$$

$$
\begin{array}{l}
\quad \textit{where } id_{body} = \mathbf{STX}(\texttt{a}, \textsc{Rename}(\textsc{Rename}(\bullet, \texttt{a}, \texttt{a2}), \mathbf{STX}(\texttt{a}, ctx), \texttt{a3})) \\
= \textit{... expanding the body ...} \\
= (\texttt{lambda}\ \mathbf{STX}(\texttt{a}, \textsc{Rename}(\bullet, \texttt{a}, \texttt{a2})) \\
\qquad (\texttt{lambda}\ \mathbf{STX}(\texttt{a}, \textsc{Rename}(ctx, \mathbf{STX}(\texttt{a}, ctx), \texttt{a3})) \\
\qquad\quad ('\!+\ \mathsf{expand}[\![id_{body}, \xi_3]\!]\ '1))) \\
\quad \textit{where the } \texttt{a3} \textit{ renaming does not apply, since } id_{body} \textit{ is not marked} \\
= (\texttt{lambda}\ \mathbf{STX}(\texttt{a}, \textsc{Rename}(\bullet, \texttt{a}, \texttt{a2})) \\
\qquad (\texttt{lambda}\ \mathbf{STX}(\texttt{a}, \textsc{Rename}(ctx, \mathbf{STX}(\texttt{a}, ctx), \texttt{a3})) \\
\qquad\quad ('\!+\ \mathbf{STX}(\texttt{a}, \textsc{Rename}(\bullet, \texttt{a}, \texttt{a2}))\ '1)))
\end{array}
$$

3.7 Compile-time bindings and local expansion

At this point, our model covers macros as they are available in many Scheme implementations. We now add two new primitives that reflect the expanded macro API of Racket: **`lvalue`** (short for `syntax-local-value`) for accessing arbitrary compile-time bindings, and **`lexpand`** (short for `local-expand`) for forcing the expansion of a sub-form.

The new primitives are available only during the application of a macro transformer, so we add them to a new set of atoms *tprim*:

$$
\begin{array}{rcl}
atom & ::= & .... \mid tprim \\
tprim & ::= & \mathbf{lvalue} \mid \mathbf{lexpand}
\end{array}
$$

Evaluation of a *tprim* application does not use δ because it relies on the expansion context. In particular, application of **`lvalue`** extracts a value from the compile-time environment, and **`lexpand`** must cancel any mark introduced for the current expansion before starting a nested expansion. We therefore revise eval to accept a compile-time environment and mark in addition to the expression to evaluate.

To evaluate the use of **`lvalue`**, the argument expression is evaluated and must produce an identifier, and the identifier must be mapped to a value in the current compile-time environment, in which case that value is the result of the **`lvalue`** call:

$$
\begin{array}{l}
\mathsf{eval}[\![\mathbf{APP}(\mathbf{lvalue}, ast), \xi, mrk]\!] \ = \ \xi(\mathsf{resolve}[\![id_{result}]\!]) \\
\ \text{where } id_{result} = \mathsf{eval}[\![ast, \xi, mrk]\!]
\end{array}
$$

The essence of **`lexpand`** is that eval for an application of **`lexpand`** must use expand. In addition, **`lexpand`** requires two bookkeeping steps:

- Before forcing expansion of the given syntax object, **`lexpand`** applies a mark to cancel the one from the enclosing macro application, and then it adds the mark back after nested expansion (to be canceled again when the enclosing expansion completes). By removing and restoring the mark for an outer expansion that is in progress, **`lexpand`** avoids interference between the original expansion and the sub-form expansion.
- To enable partial expansion, the stop list provided to **`lexpand`** creates new bindings in the compile-time environment to a STOP transform. In addition, in much the same way that **`lexpand`** removes the current expansion's mark before starting a sub-form expansion, existing STOP transforms are removed from the compile-time environment by using nostops in case a macro transformer that is invoked via **`lexpand`** itself calls **`lexpand`** :

$$\text{nostops}[\![\xi]\!] = \{\textit{var} \rightarrow \textit{transform} \mid \xi(\textit{var}) = \textit{transform} \text{ and } \textit{transform} \neq \text{STOP}\}$$

The eval rule for **`lexpand`** puts these steps together along with the evaluation of arguments to **`lexpand`** :

$$\text{eval}[\![\mathbf{APP}(\texttt{lexpand}, \textit{ast}_{\textit{expr}}, \textit{ast}_{\textit{stops}}), \xi, \textit{mrk}]\!] = \text{mark}[\![\text{expand}[\![\text{mark}[\![\textit{stx}, \textit{mrk}]\!], \xi_{\textit{stops}}]\!], \textit{mrk}]\!]$$
$$\text{where } \textit{stx} = \text{eval}[\![\textit{ast}_{\textit{expr}}, \xi, \textit{mrk}]\!], \mathbf{LIST}(\textit{id}_{\textit{stop}}, ...) = \text{eval}[\![\textit{ast}_{\textit{stops}}, \xi, \textit{mrk}]\!],$$
$$\xi_{\textit{stops}} = \text{nostops}[\![\xi]\!] + \{\text{resolve}[\![\textit{id}_{\textit{stop}}]\!] \rightarrow \text{STOP}\} \ ...$$

Expansion of a form that has a STOP transform is the same as for a QUOTE transform, except that multiple sub-forms are allowed inside the form:

$$\text{expand}[\![\mathbf{STX}(\mathbf{LIST}(\textit{id}_{\textit{stop}}, \textit{stx}, ...), \textit{ctx}), \xi]\!] = \mathbf{STX}(\mathbf{LIST}(\textit{id}_{\textit{stop}}, \textit{stx}, ...), \textit{ctx})$$
$$\text{where STOP} = \xi(\text{resolve}[\![\textit{id}_{\textit{stop}}]\!])$$

To illustrate, the program

```
(let-syntax
  public (lambda e ('syntax-error ))
 (let-syntax
   class (lambda e
          ((lambda e2
             ('car ('cdr ('stx-e e2))))
           ('lexpand ('car ('cdr ('stx-e e)))
              ('list (syntax public)))))
   (class (public '8))))
```

simulates how `public` in the class system makes sense only within a `class` form (otherwise it reports a syntax error), while `class` locally expands its body stopping at `public` forms. The program expands to `8` roughly as follows (omitting lexical-context information, since it is not directly relevant to the example):

expand⟦(`let-syntax public` ....
  (`let-syntax class` .... (`class` (`public` '8)))),
 ξ_0⟧
= ... *evaluate transformer expression for* `public` ...
= expand⟦(`let-syntax class` .... (`class` (`public` '8))),
 $\xi_1 = \xi_0$+{`public`→....}⟧
= ... *evaluate transformer expression for* `class` ...
= expand⟦(`class` (`public` '8)), $\xi_2 = \xi_1$+{`class`→....}⟧
= ... *apply the* `class` *transformer* ...
= eval⟦...., ('**`lexpand`** (`syntax` (`public` '8))
  ('**`list`** (`syntax public`))), ...,
 ξ_2, mrk_1⟧
 expansion stops immediately at `public` *:*
= eval⟦...., (`syntax` (`public` '8)), ..., ξ_2, mrk_1⟧
= ... *transformer strips the* `public` *form away* ...
= eval⟦(`syntax` '8), ξ_2, mrk_1⟧
= expand⟦'8, ξ_2⟧

Local expansion is consistent with full expansion only when the stop list is either empty or when the stop list contains at least the primitive binding forms. If a stop list omits a binding form, but includes a form that can wrap a reference to a bound variable, then a partial expansion can produce a different result than full expansion. This effect is illustrated in the following example:

```
(let-syntax
  stop (lambda e (′car (′cdr (′stx-e e))))
 (let-syntax
   ex (lambda e
        (′lexpand (′car (′cdr (′stx-e e)))
             (′list (syntax stop))))
  (ex (lambda x (let-syntax
                  arg (lambda e (syntax (stop x)))
                 (arg))))))
```

When the `ex` macro forces the expansion of `(lambda x ....)` and stops at uses of `stop`, the result is essentially `(lambda` x_2 `(stop` x_2`))`, where both x_2s are really `x`s with RENAME wrappers to redirect `x` to some x_2. The latter `x`, however, also has a MARK due to introduction by the local `arg` macro. Re-expanding `(lambda` x_2 `(stop` x_2`))` therefore produces `(lambda` x_3 `(stop` x_2`))`; since the marks on the two x_2s are not the same, the new x_3 binding does not capture the inner x_2.

3.8 Definition contexts

To support definition contexts, we add two new expansion-time primitives: **`new-defs`** (short for `syntax-local-make-definition-context`) for creating new contexts, and **`def-bind`** (short for `syntax-local-bind-syntaxes`) for binding names in a context.

$$tprim ::= .... \mid \texttt{new-defs} \mid \texttt{def-bind}$$

A definition context is similar to a RENAME record in a syntax object except that the set of renamings associated with the context is extensible imperatively. Updates of a definition context require a definition-context store Σ with addresses σ. A DEFS wrapper for syntax objects encapsulates a σ, and DEFS can also tag a σ to form a value. Within Σ, σ maps identifiers to renamed variables,

$$val ::= .... \mid \mathrm{DEFS}(\sigma)$$
$$ctx ::= .... \mid \mathrm{RENAME}(ctx, id, name, \sigma) \mid \mathrm{DEFS}(ctx, \sigma)$$

$$\sigma ::= addr \mid \mathrm{NULL}$$
$$\Sigma ::= \text{definition-context store}, \sigma \rightarrow (id \rightarrow sym)$$
$$S ::= \text{set of } \sigma$$

When resolve encounters a DEFS wrapper, it unpacks the wrapper into a sequence of RENAME wrappers. For reasons explained below, the generated RENAME wrappers must record the source definition context. Thus, RENAME is extended above to include an address σ . The special address NULL is used for RENAME wrappers that originate from `lambda` or `let-syntax` renamings.

The expand cases must be revised to take a Σ argument and produce a resulting $\langle stx, \Sigma\rangle$ tuple, and eval must similarly consume and produce a Σ. For the existing cases, the Σ is simply carried through, including to the resolve and parse meta-functions. The new case in eval for a **new-defs** form, however, extends the definition-context store with a new, empty definition context:

$$\text{eval}[\![\textbf{APP}(\texttt{new-defs}), \xi, mrk, \Sigma]\!] = \langle \text{DEFS}(\sigma), \xi, \Sigma + \{\sigma \rightarrow \varnothing\}\rangle$$
$$\text{where } \sigma = \text{fresh}$$

A new eval case handles the use of **def-bind** to extend a definition context with an identifier that corresponds to a run-time binding. A run-time binding is just like one created by `lambda`, and **def-bind** similarly generates a fresh variable and maps the original identifier to the new variable for syntax objects. While the expansion of `lambda` applies the renaming to a body expression, evaluation of **def-bind** records the renaming in a definition context. Evaluation of **def-bind** also extends the compile-time environment to indicate that the generated variable maps to itself, just like the expansion of `lambda`

$$\begin{aligned}
&\text{eval}[\![\textbf{APP}(\texttt{def-bind}, ast_{defs}, ast_{id}), \xi, mrk, \Sigma]\!] = \langle 0, \xi_2 + \{name_{new} \rightarrow (\text{VAR } id_{new})\}, \Sigma_3\rangle \\
&\quad \text{where } \langle \text{DEFS}(\sigma), \xi_1, \Sigma_1\rangle = \text{eval}[\![ast_{defs}, \xi, mrk, \Sigma]\!], \\
&\qquad \langle id, \xi_2, \Sigma_2\rangle = \text{eval}[\![ast_{id}, \xi_1, mrk, \Sigma_1]\!], name_{new} = \text{fresh}, \\
&\qquad id_{new} = \text{rename}[\![id, id, name_{new}]\!], \\
&\qquad \Sigma_3 = \Sigma_2 + \{\sigma \rightarrow \Sigma_2(\sigma) + \{\text{mark}[\![id, mrk]\!] \rightarrow name_{new}\}\}
\end{aligned}$$

When **def-bind** is used to bind an identifier to a compile-time value (including a macro transformer), the given compile-time expression must be evaluated, and then its result can be bound in the environment. Like the evaluation case for binding variables, the case for binding a compile-time value generates a fresh variable, maps it in the definition context, and extends the compile-time environment. In this case, however, the extended compile-time environment contains a compile-time value, instead of just a variable

$$\begin{aligned}
&\text{eval}[\![\textbf{APP}(\texttt{def-bind}, ast_{defs}, ast_{id}, ast_{stx}), \xi, mrk, \Sigma]\!] = \langle 0, \xi_4 + \{name_{new} \rightarrow val\}, \Sigma_5\rangle \\
&\quad \text{where } \langle \text{DEFS}(\sigma), \xi_1, \Sigma_1\rangle = \text{eval}[\![ast_{defs}, \xi, mrk, \Sigma]\!], \\
&\qquad \langle id, \xi_2, \Sigma_2\rangle = \text{eval}[\![ast_{id}, \xi_1, mrk, \Sigma_1]\!], \langle stx, \xi_3, \Sigma_3\rangle = \text{eval}[\![ast_{stx}, \xi_2, mrk, \Sigma_2]\!], \\
&\qquad \langle val, \xi_4, \Sigma_4\rangle = \text{eval}[\![\text{parse}[\![\text{defs}[\![\text{mark}[\![stx, mrk]\!], \sigma]\!], \Sigma_3]\!], \xi_3, mrk, \Sigma_3]\!], \\
&\qquad name_{new} = \text{fresh}, id_{new} = \text{rename}[\![id, id, name_{new}]\!], \\
&\qquad \Sigma_5 = \Sigma_4 + \{\sigma \rightarrow \Sigma_4(\sigma) + \{\text{mark}[\![id, mrk]\!] \rightarrow name_{new}\}\}
\end{aligned}$$

A definition context is associated with an expression by extending **lexpand** to accept a definition context as its last argument. The definition context is applied to the given expression before it is expanded (using a defs meta-function that is like rename and mark) so that expansion uses the context. Less obviously, the definition context is also added to the result of local expansion. For syntax objects introduced by local expansion, the second addition ensures that if the introduced syntax objects correspond to a definition, then the definition's binding will use the correct lexical context

$$\begin{aligned}
&\text{eval}[\![\textbf{APP}(\texttt{lexpand}, ast_{expr}, ast_{stops}, ast_{defs}), \xi, mrk, \Sigma]\!] = \langle \text{mark}[\![\text{defs}[\![stx, \sigma]\!], mrk]\!], \xi_3, \Sigma_4\rangle \\
&\quad \text{where } \langle stx_{expr}, \xi_1, \Sigma_1\rangle = \text{eval}[\![ast_{expr}, \xi, mrk, \Sigma]\!], \\
&\qquad \langle \textbf{LIST}(id_{stop}, ...), \xi_2, \Sigma_2\rangle = \text{eval}[\![ast_{stops}, \xi_1, mrk, \Sigma_1]\!], \\
&\qquad \langle \text{DEFS}(\sigma), \xi_3, \Sigma_3\rangle = \text{eval}[\![ast_{defs}, \xi_2, mrk, \Sigma_2]\!], \\
&\qquad \xi_{stops} = \text{nostops}[\![\xi_3]\!] + \{\text{resolve}[\![id_{stop}, \Sigma_3]\!] \rightarrow \text{STOP}\} \ ..., \\
&\qquad stx_{new} = \text{defs}[\![\text{mark}[\![stx_{expr}, mrk]\!], \sigma]\!], \langle stx, \Sigma_4\rangle = \text{expand}[\![stx_{new}, \xi_{stops}, \Sigma_3]\!]
\end{aligned}$$

Given the extended definition of *ctx*, a natural extension of resolve is to add a DEFS clause while generally extending resolve to accept the current store Σ. The new DEFS clause could simply unpack the wrapper into a set of RENAME wrappers based on the content of the definition context in the store, then recur:

$$\begin{aligned}
&\mathsf{renames}[\![\sigma, (), ctx]\!] &&= ctx \\
&\mathsf{renames}[\![\sigma, ((id\ name)\ (id_2\ name_2)\ ...), ctx]\!] &&= \mathsf{renames}[\![\sigma, ((id_2\ name_2)\ ...), \\
& && \qquad \mathrm{RENAME}(ctx, id, name, \sigma)]\!] \\
&\mathsf{resolve}[\![\mathbf{STX}(val, \mathrm{DEFS}(ctx, \sigma)), \Sigma]\!] &&= \mathsf{resolve}[\![\mathbf{STX}(val, ctx_{new}), \Sigma]\!] \\
&\quad \text{where } \{id \rightarrow name_{new}\ ...\} = \Sigma(\sigma), \\
&\qquad ctx_{new} = \mathsf{renames}[\![\sigma, ((id\ name_{new})\ ...), ctx]\!]
\end{aligned}$$

This simple extension of resolve does not work because it does not terminate when σ is part of a cycle in Σ. A cycle is created in Σ for most definition contexts because each defined identifier is placed into the context where bindings occur. More complex cycles are created when definition contexts are nested, as in nested `define-package` forms (see Section 2.5). For example, in the Racket expression

```
(define-package p ()
  (define x 1)
  (define-package q ()
    (define x 2)))
```

the two defined identifiers must resolve to different bindings, say, `x1` and `x3`. The corresponding syntax objects are roughly

$$\mathbf{STX}(\mathtt{x}, \mathrm{DEFS}(\mathrm{DEFS}(\bullet, \sigma_1), \sigma_1))$$
$$\mathbf{STX}(\mathtt{x}, \mathrm{DEFS}(\mathrm{DEFS}(\mathrm{DEFS}(\mathrm{DEFS}(\bullet, \sigma_1), \sigma_2), \sigma_2), \sigma_1))$$

$$\begin{aligned}
\Sigma = \{&\sigma_1 \leftarrow \{\mathbf{STX}(\mathtt{x}, \mathrm{DEFS}(\bullet, \sigma_1)) \leftarrow \mathtt{x1}, \\
&\qquad \mathbf{STX}(\mathtt{x}, \mathrm{DEFS}(\mathrm{DEFS}(\mathrm{DEFS}(\mathrm{DEFS}(\bullet, \sigma_1), \sigma_2), \sigma_2), \sigma_1)) \leftarrow \mathtt{x3}\} \\
&\sigma_2 \leftarrow \{\mathrm{DEFS}(\mathrm{DEFS}(\mathrm{DEFS}(\bullet, \sigma_1), \sigma_2), \sigma_2) \leftarrow \mathtt{x2}\}\}
\end{aligned}$$

The first identifier has σ_1 twice because the identifier appears in both the first `define` form and its expansion. The second identifier has nested σs because it appears before and after both the outer expansion of `define-package` and the inner expansion of `define`. In Σ, the σ_1 binding reflects the final identifiers. The σ_2 binding reflects the state of the second `x` by the time it was bound for the inner package. It was put into the σ_1 context during the expansion of the `define-package` form, and then put into the σ_2 context before and after expanding the inner `define`. The inner expansion created the temporary binding `x2`, but it was later subsumed by the `x3` binding for the enclosing context.

To accommodate cycles within Σ, resolve must keep track of the contexts that are already being used toward a renaming. For an initial call to resolve, no contexts are already being used. When a DEFS tag is unpacked into RENAMEs, then the corresponding context is already being used for the purposes of checking targets of renamings in the branches of the wrapper (i.e., the *id* in each RENAME wrapper). However, the context is not yet used for the spine of the lexical-context wrapper, because if no renaming applies among the unpacked ones, a later DEFS wrapper for the same definition context might apply. Thus, resolve accepts two sets of definition contexts to be skipped: one for the spine, and another for branches that are rename targets,

$$\text{resolve}[\![stx, \Sigma]\!] = \text{resolve*}[\![stx, \Sigma, \varnothing, \varnothing]\!]$$

....

$$\text{resolve*}[\![\mathbf{STX}(\text{'}name, \text{RENAME}(ctx, id_{orig}, name_{new}, \sigma)), \Sigma, S_{spine}, S_{branch}]\!] = name_{new}$$
$$\text{where } name_1 = \text{resolve*}[\![id_{orig}, \Sigma, S_{branch}, S_{branch}]\!],$$
$$name_1 = \text{resolve*}[\![\mathbf{STX}(\text{'}name, ctx), \Sigma, \{\sigma\} \cup S_{spine}, S_{branch}]\!],$$
$$\text{marksof}[\![id_{orig}, name_1]\!] = \text{marksof}[\![\mathbf{STX}(\text{'}name, ctx), name_1]\!]$$
$$\text{resolve*}[\![\mathbf{STX}(\text{'}name, \text{DEFS}(ctx, \sigma)), \Sigma, S_{spine}, S_{branch}]\!] = \text{resolve*}[\![\mathbf{STX}(\text{'}name, ctx), \Sigma, S_{spine}, S_{branch}]\!]$$
$$\text{where } \sigma \in S_{spine}$$
$$\text{resolve*}[\![\mathbf{STX}(\text{'}name, \text{DEFS}(ctx, \sigma)), \Sigma, S_{spine}, S_{branch}]\!] = \text{resolve*}[\![\mathbf{STX}(\text{'}name, ctx_{new}), \Sigma, S_{spine}, \{\sigma\} \cup S_{branch}]\!]$$
$$\text{where } \{id \to name_{new} \ldots\} = \Sigma(\sigma),\ ctx_{new} = \text{renames}[\![\sigma, ((id\ name_{new}) \ldots), ctx]\!]$$

Finally, we revise marksof to handle DEFS wrappers, taking care to properly support renaming of identifiers that are bound in the definition context. Consider the following Racket example:

```
(lambda ()
  (define x 1)
  (define-syntax m (lambda (stx) #'(list x)))
  (m))
```

When a definition context is used to expand the body forms of this `lambda`, then all identifiers acquire the definition context. The local expansion of `(m)`, furthermore, produces an `x` that has a mark in addition to the definition context. If the `x` from `define` is then used as a `letrec` binding to continue expansion, then the extra mark on the `x` from `(m)` could prevent it from being bound by the `letrec`. This is the same potential problem as described at the end of Section 3.7, and it occurs because `(m)` is expanded only far enough to discover that it acts an expression rather than a definition.

To avoid this problem, resolve must ignore marks that are introduced during partial expansion for identifiers that are bound by the partial expansion's definition context. Ignoring such marks simulates a complete expansion, which would replace a marked variable with the fresh name that is used for its binding. Since **lexpand** adds the definition context to both its argument and result, two instances of the definition context serve to bracket the marks that should be ignored later by resolve. Combining this idea with the observation from Section 3.6.2 that marks *after* the definition-context renaming can be ignored for a further renaming, it suffices to make marksof ignore all marks after the first instance of a definition-context renaming:

$$\text{marksof}[\![\mathbf{STX}(val, \text{DEFS}(ctx, \sigma)), \Sigma, name]\!] = ()$$
$$\text{where } name \in \text{rng}(\Sigma(\sigma))$$
$$\text{marksof}[\![\mathbf{STX}(val, \text{DEFS}(ctx, \sigma)), \Sigma, name]\!] = \text{marksof}[\![\mathbf{STX}(val, ctx), \Sigma, name]\!]$$

4 Related work

Our model builds directly on the model of Dybvig *et al.* (1993), adding extensions for compile-time bindings, partial expansion, and definition contexts. Another difference in our model is that the expander, which maps syntax objects to syntax objects, is decoupled from the parser, which maps syntax objects to an executable AST. This change allows us to model `local-expand`, but it also reflects Racket's pervasive use of syntax objects as a basis for program analysis and transformation (see Section 2.6). Previous models of Scheme macros do not account for the handling of internal-definition contexts (e.g., in the body of `lambda`); *ribs*, as described informally by Waddell and Dybvig (1999), sound similar to our definition contexts, but no model is provided, and the construct is not accessible to macro transformers.

Our model inherits one drawback of the Dybvig *et al.* (1993) model: Whether marks and renamings actually make macros respect lexical scope as intended is hardly apparent. Other models of macros face similar problems:

- Gasbichler (2006) attacked the gap between specification and mechanism in his model of macros based on explicit substitutions, but Gasbichler's model treats pattern-variable bindings differently than other bindings, which turns out not to work completely right for macro-generating macros, thus leaving a gap in the explanation.
- The λ_m calculus of Herman (2010) creates a tight correspondence between specification and behavior for a restricted subset of `syntax-rules` macros. The λ_m calculus uses a custom-type system to specify the binding structure of a macro's arguments. Expressions are annotated with the new bindings brought into scope, and macros with ambiguous scoping rules are disallowed. The calculus does not handle the flexibility and power of `syntax-case` macros, and the type system would require significant extension to represent the essence of local expansion and definition contexts.
- Other frameworks for lexically scoped macros, notably syntactic closures (Bawden and Rees, 1988) and explicit renaming (Clinger, 1991), use a notion of lexical context that more directly maps to the programmer's view of binding scopes. Unfortunately, the more direct representation moves binding information into the expansion environment; in the case of syntactic closures, it tangles the representation of syntax and expansion environments, and in the case of explicit renaming, identifier comparisons depend on a compile-time environment. Our goals require a purely "syntactic" representation of syntax, which can be locally expanded, transported into a new context, and then re-expanded.

An important direction for further research is to find a model with the syntactic advantage of Dybvig *et al.* (1993), but with a more obvious connection to the usual notion of binding scopes that is able to support our extensions for cooperation among macros.

Previous work on expansion-passing style macros (Dybvig *et al.*, 1988) addresses the problem of expanding sub-forms in a macro use. In expansion-passing style, a macro receives two arguments: the term to transform and an expander function. The macro can call the function to expand sub-forms, and it can pass a modified expander function to be used for the sub-form expansion. Similarly, Common Lisp provides the functions `macroexpand` and `macroexpand-1`, as well as an expansion hook `*macroexpand-hook*`. Both of these mechanisms give macros the power to expand sub-forms, and they give a macro the ability to change the expander's behavior for the duration of the sub-form expansion. In contrast, `local-expand` always invokes the standard expander, allowing only the addition of new stopping conditions and an optional definition context. These restrictions make `local-expand` less powerful but more predictable than previous mechanisms for sub-form expansion. In addition, `local-expand` works with macros that respect lexical scope, whereas previous facilities were developed for scope-oblivious systems.

Continuation-passing style (CPS) also enables a kind of sub-form expansion for macros, as described by Hilsdale and Friedman (2000). Only macros explicitly written in CPS can participate in sub-form expansion, so such macros cannot easily reuse existing forms like `define`. Furthermore, since macros cannot verify that sub-forms follow the protocol, mistakes generally lead to mysterious error messages at best and bewildering behavior at worst.

Compile-time meta-programming in the style of Template Haskell (Sheard and Peyton Jones, 2002) supports the expansion of sub-forms within a macro transformer because macros are compile-time functions that can be called directly from other compile-time functions. Macros in the Template Haskell also respect lexical scope. Unlike Lisp and Scheme, however, uses of macros must be explicitly marked in the program source with a leading $, which creates different demands on the representation of syntax and the resolution of binding. For example, an identifier's role within a template as binder or not can be determined immediately, whereas the determination must be delayed within Scheme templates. The advantage of Scheme-style macros, and the target of our work, is to allow new syntactic forms that have the same status as built-in syntactic forms, thus supporting a tower of languages.

Other systems address the need for cooperation and communication of language extension at a different level. Both Ziggurat (Fisher and Shivers, 2008) and Silver (Van Wyk *et al.*, 2009) support static analysis in languages with extensible syntax. Expansion or "delegation" is automatically triggered by the system as necessary to support analyses, and expansion proceeds only far enough to produce a syntactic form that can be analyzed. Compared to Ziggurat and Silver, macro expansion in Racket is simpler and at a lower level; only expansion and binding information is available for a sub-term, and other information must be encoded in the expansion.

Language constructs based on fresh names (Gabbay and Pitts, 1999; Shinwell *et al.*, 2003) or the higher order abstract syntax (Pfenning and Elliott, 1988; Pfenning and Schürmann, 1999) address the problem of manipulating program fragments with bindings, but they have different operations than syntax objects. Programs using fresh-name features explicitly open and close term representations instead of automatically absorbing lexical information. With the higher order abstract syntax, binders and bindings are implicit instead of entities that can be manipulated explicitly. Syntax objects fit somewhere in between: lexical information is maintained automatically but can be manipulated more directly.

Acknowledgments

Thanks to Robert Ransom, Michael Sperber, Mitch Wand, Paul Stansifer, and the anonymous referees for many helpful comments and suggestions.

References

Bawden, A. & Rees, J (1988) Syntactic closures. In *Proceedings of the Lisp and Functional Programming*, July 25–27, Snowbird, UT, USA. Pittsburgh: ACM, pp. 86–95.

Clinger, William D. (1991) Hygienic macros through explicit renaming. *LISP Pointers* **4**(4), 25–28.

Clinger, C. & Rees, J. (1991) Macros that work. In *Proceedings of ACM Symposium on Principles of Programming Languages*. New York, NY: ACM, pp. 155–162.

Dybvig, R. Kent, Friedman, Daniel P. & Haynes, Christopher T. (1988) Expansion-passing style: A general macro mechanism. *LISP Symb. Comput.* **1**(1), 53–75.

Dybvig, R. Kent, Hieb, R. & Bruggeman, C. (1993) Syntactic abstraction in Scheme. *LISP Symb. Comput.* **5**(4), 295–326.

Felleisen, M., Findler, R. B. & Flatt, M. (2009) *Semantics Engineering with PLT Redex*. Cambridge, MA: MIT Press.

Fisher, D. & Shivers, O (2008) Building language towers with Ziggurat. *J. Funct. Program.* **18**(5–6), 707–780.

Flatt, M. (2002) Compilable and composable macros, you want it *when?* In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP 2002)*, Pittsburgh, PA, USA, October 4–6, pp. 72–83.

Gabbay, M. & Pitts, A. (1999) A new approach to abstract syntax involving binders. In *the Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS'99)*, Trento, Italy, pp. 341–363.

Gasbichler, M. (2006) *Fully-Parameterized Higher-Order Modules with Hygienic Macros*. PhD dissertation, Universität Tübingen, Tübingen, Germany.

Ghuloum, A. & Dybvig, R. Kent (2007) Portable syntax-case. Accessed April, 2012. Available at: https://www.cs.indiana.edu/chezscheme/syntax-case/

Herman, D. (2010) *A Theory of Typed Hygienic Macros*. PhD dissertation, Northeastern University, Boston, Massachusetts.

Hilsdale, E. & Friedman, D. P. (2000) Writing macros in continuation-passing style. *Proceedings of the Workshop on Scheme and Functional Programming*, Montreal, September 17.

Kohlbecker, E. Eugene, Friedman, Daniel P. Felleisen, M. & Duba, B. (1986) Hygienic macro expansion. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. New York, NY: ACM, pp. 151–181.

Kohlbecker, E. Eugene & Wand, M. (1987) Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Munich, Germany, January 2123, pp. 77–84.

Milner, R., Tofte, M. & Harper, R. (1990) *The Definition of Standard ML*. Cambridge, MA: MIT Press.

Pfenning, F. & Elliott, C. (1988) Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*. New York, NY: ACM, pp. 199–208.

Pfenning, F. & Schürmann, C. (1999) System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the International Conference on Automated Deduction*, Trento, Italy, pp. 202–206.

Sheard, T. & Peyton Jones, S. (2002) Template meta-programming for Haskell. In *Proceedings of Haskell Workshop 2002*, Pittsburgh, PA, USA, pp. 60–75.

Shinwell, Mark R., Pitts, A. M. & Gabbay, M. J. (2003) FreshML: Programming with binders made simple. In *Proceedings of the 8th ACM International Conference on Functional Programming*. New York, NY: ACM, pp. 263–274.

Sperber, M. (Ed.) (2009) Revised ^{6} report on the algorithmic language Scheme. *J. Funct. Program.* **19**(supplement), 1–301.

Van Tonder, A. (2007) R6RS libraries and macros. Accessed April, 2012. Available at: http://www.het.brown.edu/people/andre/macros/

Van Wyk, E., Bodin, D., Krishnan, L. & Gao, J. (2009) Silver: an extensible attribute grammar system. *Sci. Comput. Program.* **75**(1–2) doi: 10.1016/j.scico.2009.07.004

Waddell, O. & Dybvig, R. Kent (1999) Extending the scope of syntactic abstraction. In *Proceedings of ACM Symposium on the Principles of Programming Languages*. New York, NY: ACM, pp. 203–213.

Supplementary materials

For supplementary material for this article, please visit http:dx.doi.org/10.1017/S0956796812000093.

Appendix

$$
\begin{array}{rl}
ast & ::= var \mid \textbf{APP}(ast, ast, ...) \mid val \\
var & ::= \textbf{VAR}(name) \\
val & ::= \textbf{FUN}(var, ast) \mid atom \\
 & \quad \mid \textbf{LIST}(val, ...) \mid stx \mid \mathrm{DEFS}(\sigma) \\
atom & ::= sym \mid prim \mid tprim \mid .... \\
sym & ::= \text{'}name \\
prim & ::= \texttt{stx-e} \mid \texttt{mk-stx} \mid .... \\
tprim & ::= \texttt{new-defs} \mid \texttt{def-bind} \\
 & \quad \mid \texttt{lvalue} \mid \texttt{lexpand} \\
\sigma & ::= addr \mid \mathrm{NULL}
\end{array}
$$

ξ ::= a mapping from *name* to *transform*
Σ ::= definition-context store, $\sigma \rightarrow (id \rightarrow sym)$
S ::= set of σ

$$
\begin{array}{rl}
stx & ::= \textbf{STX}(atom, ctx) \\
 & \quad \mid \textbf{STX}(\textbf{LIST}(stx, ...), ctx) \\
id & ::= \textbf{STX}(sym, ctx) \\
ctx & ::= \bullet \\
 & \quad \mid \mathrm{RENAME}(ctx, id, name, \sigma) \\
 & \quad \mid \mathrm{MARK}(ctx, mrk) \\
 & \quad \mid \mathrm{DEFS}(ctx, \sigma) \\
transform & ::= \mathrm{FUN} \mid \mathrm{LET\text{-}SYNTAX} \mid \mathrm{QUOTE} \\
 & \quad \mid (\mathrm{VAR}\ id) \mid val \mid \mathrm{STOP}
\end{array}
$$

name ::= a token such as `x`, `egg`, or `lambda`
addr, *mrk* ::= *name*

$$
\begin{array}{ll}
\delta(\texttt{stx-e}, \textbf{STX}(val, ctx)) & = val \\
\delta(\texttt{mk-stx}, atom, \textbf{STX}(val, ctx)) & = \textbf{STX}(atom, ctx) \\
\delta(\texttt{mk-stx}, \textbf{LIST}(stx, ...), \textbf{STX}(val, ctx)) & = \textbf{STX}(\textbf{LIST}(stx, ...), ctx) \\
...
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{eval}[\![\textbf{APP}(\textbf{FUN}(var, ast_{body}), ast_{arg}), \xi, mrk, \Sigma]\!] & = \mathsf{eval}[\![ast_{body}[var \leftarrow val], \xi_1, mrk, \Sigma_1]\!] \\
\quad \text{where } \langle val, \xi_1, \Sigma_1 \rangle = \mathsf{eval}[\![ast_{arg}, \xi, mrk, \Sigma]\!] & \\
\mathsf{eval}[\![\textbf{APP}(prim, ast_{arg}, ...), \xi, mrk, \Sigma]\!] & = \langle \delta(prim, val, ...), \xi_1, \Sigma_1 \rangle \\
\quad \text{where } \langle (val\ ...), \xi_1, \Sigma_1 \rangle = \mathsf{eval}^*[\![(), (ast_{arg}\ ...), \xi, mrk, \Sigma]\!] & \\
\mathsf{eval}[\![\textbf{APP}(ast_{op}, ast_{arg}, ...)]\!] & = \mathsf{eval}[\![\textbf{APP}(\mathsf{eval}[\![ast_{op}]\!], ast_{arg}, ...)]\!] \\
\mathsf{eval}[\![val, \xi, mrk, \Sigma]\!] & = \langle val, \xi, \Sigma \rangle \\
\mathsf{eval}[\![\textbf{APP}(\texttt{lvalue}, ast), \xi, mrk, \Sigma]\!] & = \langle \xi(\mathsf{resolve}[\![stx, \Sigma_1]\!]), \xi_1, \Sigma_1 \rangle \\
\quad \text{where } \langle stx, \xi_1, \Sigma_1 \rangle = \mathsf{eval}[\![ast, \xi, mrk, \Sigma]\!] & \\
\mathsf{eval}[\![\textbf{APP}(\texttt{lexpand}, ast_{expr}, ast_{stops}, ast_{defs}), \xi, mrk, \Sigma]\!] & = \langle \mathsf{mark}[\![\mathsf{defs}[\![stx, \sigma]\!], mrk]\!], \xi_3, \Sigma_4 \rangle \\
\quad \text{where } \langle stx_{expr}, \xi_1, \Sigma_1 \rangle = \mathsf{eval}[\![ast_{expr}, \xi, mrk, \Sigma]\!], & \\
\qquad \langle \textbf{LIST}(id_{stop}, ...), \xi_2, \Sigma_2 \rangle = \mathsf{eval}[\![ast_{stops}, \xi_1, mrk, \Sigma_1]\!], & \\
\qquad \langle \mathrm{DEFS}(\sigma), \xi_3, \Sigma_3 \rangle = \mathsf{eval}[\![ast_{defs}, \xi_2, mrk, \Sigma_2]\!], & \\
\qquad \xi_{stops} = \mathsf{nostops}[\![\xi_3]\!] + \{\mathsf{resolve}[\![id_{stop}, \Sigma_3]\!] \rightarrow \mathrm{STOP}\}\ ..., & \\
\qquad stx_{new} = \mathsf{defs}[\![\mathsf{mark}[\![stx_{expr}, mrk]\!], \sigma]\!], \langle stx, \Sigma_4 \rangle = \mathsf{expand}[\![stx_{new}, \xi_{stops}, \Sigma_3]\!] & \\
\mathsf{eval}[\![\textbf{APP}(\texttt{new-defs}), \xi, mrk, \Sigma]\!] & = \langle \mathrm{DEFS}(\sigma), \xi, \Sigma + \{\sigma \rightarrow \varnothing\} \rangle \\
\quad \text{where } \sigma = \text{fresh} & \\
\mathsf{eval}[\![\textbf{APP}(\texttt{def-bind}, ast_{defs}, ast_{id}), \xi, mrk, \Sigma]\!] & = \langle 0, \xi_2 + \{name_{new} \rightarrow (\mathrm{VAR}\ id_{new})\}, \Sigma_3 \rangle \\
\quad \text{where } \langle \mathrm{DEFS}(\sigma), \xi_1, \Sigma_1 \rangle = \mathsf{eval}[\![ast_{defs}, \xi, mrk, \Sigma]\!], \langle id, \xi_2, \Sigma_2 \rangle = \mathsf{eval}[\![ast_{id}, \xi_1, mrk, \Sigma_1]\!], & \\
\qquad name_{new} = \text{fresh}, id_{new} = \mathsf{rename}[\![id, id, name_{new}]\!], & \\
\qquad \Sigma_3 = \Sigma_2 + \{\sigma \rightarrow \Sigma_2(\sigma) + \{\mathsf{mark}[\![id, mrk]\!] \rightarrow name_{new}\}\} & \\
\mathsf{eval}[\![\textbf{APP}(\texttt{def-bind}, ast_{defs}, ast_{id}, ast_{stx}), \xi, mrk, \Sigma]\!] & = \langle 0, \xi_4 + \{name_{new} \rightarrow val\}, \Sigma_5 \rangle \\
\quad \text{where } \langle \mathrm{DEFS}(\sigma), \xi_1, \Sigma_1 \rangle = \mathsf{eval}[\![ast_{defs}, \xi, mrk, \Sigma]\!], \langle id, \xi_2, \Sigma_2 \rangle = \mathsf{eval}[\![ast_{id}, \xi_1, mrk, \Sigma_1]\!], & \\
\qquad \langle stx, \xi_3, \Sigma_3 \rangle = \mathsf{eval}[\![ast_{stx}, \xi_2, mrk, \Sigma_2]\!], & \\
\qquad \langle val, \xi_4, \Sigma_4 \rangle = \mathsf{eval}[\![\mathsf{parse}[\![\mathsf{defs}[\![\mathsf{mark}[\![stx, mrk]\!], \sigma]\!], \Sigma_3]\!], \xi_3, mrk, \Sigma_3]\!], & \\
\qquad name_{new} = \text{fresh}, id_{new} = \mathsf{rename}[\![id, id, name_{new}]\!], & \\
\qquad \Sigma_5 = \Sigma_4 + \{\sigma \rightarrow \Sigma_4(\sigma) + \{\mathsf{mark}[\![id, mrk]\!] \rightarrow name_{new}\}\} &
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{eval}^*[\![(val\ ...), (), \xi, mrk, \Sigma]\!] & = \langle (val\ ...), \xi, \Sigma \rangle \\
\mathsf{eval}^*[\![(val\ ...), (ast_0\ ast_1\ ...), \xi, mrk, \Sigma]\!] & = \mathsf{eval}^*[\![(val\ ...\ val_0), (ast_1\ ...), \xi_1, mrk, \Sigma_1]\!] \\
\quad \text{where } \langle val_0, \xi_1, \Sigma_1 \rangle = \mathsf{eval}[\![ast_0, \xi, mrk, \Sigma]\!] &
\end{array}
$$

$\text{expand}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{lam}, id_{arg}, stx_{body}), ctx), \xi, \Sigma]\!] = \langle \mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{lam}, id_{new}, stx_{expbody}), ctx), \Sigma_1\rangle$
where $\text{F}\textsc{un} = \xi(\text{resolve}[\![id_{lam}, \Sigma]\!])$, $name_{new}$ = fresh, $id_{new} = \text{rename}[\![id_{arg}, id_{arg}, name_{new}]\!]$,
$stx_{newbody} = \text{rename}[\![stx_{body}, id_{arg}, name_{new}]\!]$, $\xi_{new} = \xi + \{name_{new} \rightarrow (\text{V}\textsc{ar}\ id_{new})\}$,
$\langle stx_{expbody}, \Sigma_1\rangle = \text{expand}[\![stx_{newbody}, \xi_{new}, \Sigma]\!]$

$\text{expand}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{quote}, stx), ctx), \xi, \Sigma]\!] = \langle \mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{quote}, stx), ctx), \Sigma\rangle$
where $\text{Q}\textsc{uote} = \xi(\text{resolve}[\![id_{quote}, \Sigma]\!])$

$\text{expand}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{ls}, id_{mac}, stx_{rhs}, stx_{body}), ctx), \xi, \Sigma]\!] = \text{expand}[\![stx_{newbody}, \xi + \{name_{new} \rightarrow val\}, \Sigma_1]\!]$
where $\text{L}\textsc{et-}\text{S}\textsc{yntax} = \xi(\text{resolve}[\![id_{ls}, \Sigma]\!])$, $name_{new}$ = fresh,
$\langle val, \xi_1, \Sigma_1\rangle = \text{eval}[\![\text{parse}[\![stx_{rhs}, \Sigma]\!], \xi, \texttt{no-mrk}, \Sigma]\!]$,
$stx_{newbody} = \text{rename}[\![stx_{body}, id_{mac}, name_{new}]\!]$

$\text{expand}[\![stx_{macapp}, \xi, \Sigma]\!] = \text{expand}[\![\text{mark}[\![stx_{exp}, mrk_{new}]\!], \xi, \Sigma_1]\!]$
where $\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{mac}, stx_{arg}, ...), ctx) = stx_{macapp}$, $val = \xi(\text{resolve}[\![id_{mac}, \Sigma]\!])$, mrk_{new} = fresh,
$\langle stx_{exp}, \xi_1, \Sigma_1\rangle = \text{eval}[\![\mathbf{A}\textbf{PP}(val, \text{mark}[\![stx_{macapp}, mrk_{new}]\!]), \xi, mrk_{new}, \Sigma]\!]$

$\text{expand}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{stop}, stx, ...), ctx), \xi, \Sigma]\!] = \langle \mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{stop}, stx, ...), ctx), \Sigma\rangle$
where $\text{S}\textsc{top} = \xi(\text{resolve}[\![id_{stop}, \Sigma]\!])$

$\text{expand}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(stx_{rtor}, stx_{rnd}, ...), ctx), \xi, \Sigma]\!] = \langle \mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(stx_{exprtor}, stx_{exprnd}, ...), ctx), \Sigma_1\rangle$
where $\langle (stx_{exprtor}\ stx_{exprnd}\ ...), \Sigma_1\rangle = \text{expand*}[\![(), (stx_{rtor}\ stx_{rnd}\ ...), \xi, \Sigma]\!]$

$\text{expand}[\![id, \xi, \Sigma]\!] = \langle id_{new}, \Sigma\rangle$
where $(\text{V}\textsc{ar}\ id_{new}) = \xi(\text{resolve}[\![id, \Sigma]\!])$

$\text{expand*}[\![(stx_{done}\ ...), (), \xi, \Sigma]\!] = \langle (stx_{done}\ ...), \Sigma\rangle$

$\text{expand*}[\![(stx_{done}\ ...), (stx_0\ stx_1\ ...), \xi, \Sigma]\!] = \text{expand*}[\![(stx_{done}\ ...\ stx_{done0}), (stx_1\ ...), \xi, \Sigma_1]\!]$
where $\langle stx_{done0}, \Sigma_1\rangle = \text{expand}[\![stx_0, \xi, \Sigma]\!]$

$\text{parse}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{lambda}, id_{arg}, stx_{body}), ctx), \Sigma]\!] = \mathbf{F}\textbf{UN}(\mathbf{V}\textbf{AR}(\text{resolve}[\![id_{arg}, \Sigma]\!]), \text{parse}[\![stx_{body}, \Sigma]\!])$
where $\texttt{lambda} = \text{resolve}[\![id_{lambda}, \Sigma]\!]$

$\text{parse}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{quote}, stx), ctx), \Sigma]\!] = \text{strip}[\![stx]\!]$
where $\texttt{quote} = \text{resolve}[\![id_{quote}, \Sigma]\!]$

$\text{parse}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(id_{syntax}, stx), ctx), \Sigma]\!] = stx$
where $\texttt{syntax} = \text{resolve}[\![id_{syntax}, \Sigma]\!]$

$\text{parse}[\![\mathbf{S}\textbf{TX}(\mathbf{L}\textbf{IST}(stx_{rator}, stx_{rand}, ...), ctx), \Sigma]\!] = \mathbf{A}\textbf{PP}(\text{parse}[\![stx_{rator}, \Sigma]\!], \text{parse}[\![stx_{rand}, \Sigma]\!], ...)$

$\text{parse}[\![id, \Sigma]\!] = \mathbf{V}\textbf{AR}(\text{resolve}[\![id, \Sigma]\!])$

$$\mathsf{resolve}[\![stx, \Sigma]\!] = \mathsf{resolve^*}[\![stx, \Sigma, \emptyset, \emptyset]\!]$$

$$\begin{aligned}
&\mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, \bullet), \Sigma, S_{spine}, S_{branch}]\!] &&= name\\
&\mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, \text{M}\scriptstyle{\text{ARK}}\displaystyle(ctx, mrk)), \Sigma, S_{spine}, S_{branch}]\!] &&= \mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, ctx),\\
& &&\qquad \Sigma, S_{spine}, S_{branch}]\!]\\
&\mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, \text{RENAME}(ctx, id_{orig}, name_{new}, \sigma)), \Sigma, S_{spine}, S_{branch}]\!] &&= name_{new}\\
&\quad \text{where } name_1 = \mathsf{resolve^*}[\![id_{orig}, \Sigma, S_{branch}, S_{branch}]\!],\\
&\qquad name_1 = \mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, ctx), \Sigma, \{\sigma\} \cup S_{spine}, S_{branch}]\!],\\
&\qquad \mathsf{marksof}[\![id_{orig}, \Sigma, name_1]\!] = \mathsf{marksof}[\![\textbf{S}\text{TX}(\text{'}name, ctx), \Sigma, name_1]\!]\\
&\mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, \text{RENAME}(ctx, id_{orig}, name_2, \sigma)), \Sigma, S_{spine}, S_{branch}]\!] &&= \mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, ctx),\\
& &&\qquad \Sigma, S_{spine}, S_{branch}]\!]\\
&\mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, \text{DEFS}(ctx, \sigma)), \Sigma, S_{spine}, S_{branch}]\!] &&= \mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, ctx),\\
& &&\qquad \Sigma, S_{spine}, S_{branch}]\!]\\
&\quad \text{where } \sigma \in S_{spine}\\
&\mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, \text{DEFS}(ctx, \sigma)), \Sigma, S_{spine}, S_{branch}]\!] &&= \mathsf{resolve^*}[\![\textbf{S}\text{TX}(\text{'}name, ctx_{new}),\\
& &&\qquad \Sigma, S_{spine}, \{\sigma\} \cup S_{branch}]\!]\\
&\quad \text{where } \{id \to name_{new} \ldots\} = \Sigma(\sigma),\ ctx_{new} = \mathsf{renames}[\![\sigma, ((id\ name_{new}) \ldots), ctx]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{renames}[\![\sigma, (), ctx]\!] &&= ctx\\
&\mathsf{renames}[\![\sigma, ((id\ name)\ (id_2\ name_2) \ldots), ctx]\!] &&= \mathsf{renames}[\![\sigma, ((id_2\ name_2) \ldots),\\
& &&\qquad \text{RENAME}(ctx, id, name, \sigma)]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{mark}[\![\textbf{S}\text{TX}(atom, ctx), mrk]\!] &&= \textbf{S}\text{TX}(atom, \text{MARK}(ctx, mrk))\\
&\mathsf{mark}[\![\textbf{S}\text{TX}(\textbf{LIST}(stx, \ldots), ctx), mrk]\!] &&= \textbf{S}\text{TX}(\textbf{LIST}(\mathsf{mark}[\![stx, mrk]\!], \ldots), \text{MARK}(ctx, mrk))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{rename}[\![\textbf{S}\text{TX}(atom, ctx), id, name]\!] &&= \textbf{S}\text{TX}(atom, \text{RENAME}(ctx, id, name, \text{NULL}))\\
&\mathsf{rename}[\![\textbf{S}\text{TX}(\textbf{LIST}(stx, \ldots), ctx), id, name]\!] &&= \textbf{S}\text{TX}(\textbf{LIST}(\mathsf{rename}[\![stx, id, name]\!], \ldots),\\
& &&\qquad \text{RENAME}(ctx, id, name, \text{NULL}))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{defs}[\![\textbf{S}\text{TX}(atom, ctx), \sigma]\!] &&= \textbf{S}\text{TX}(atom, \text{DEFS}(ctx, \sigma))\\
&\mathsf{defs}[\![\textbf{S}\text{TX}(\textbf{LIST}(stx, \ldots), ctx), \sigma]\!] &&= \textbf{S}\text{TX}(\textbf{LIST}(\mathsf{defs}[\![stx, \sigma]\!], \ldots), \text{DEFS}(ctx, \sigma))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{marksof}[\![\textbf{S}\text{TX}(val, \bullet), \Sigma, name]\!] &&= ()\\
&\mathsf{marksof}[\![\textbf{S}\text{TX}(val, \text{MARK}(ctx, mrk)), \Sigma, name]\!] &&= mrk \oplus \mathsf{marksof}[\![\textbf{S}\text{TX}(val, ctx), \Sigma, name]\!]\\
&\mathsf{marksof}[\![\textbf{S}\text{TX}(val, \text{RENAME}(ctx, id, name_2, \sigma)), \Sigma, name]\!] &&= \mathsf{marksof}[\![\textbf{S}\text{TX}(val, ctx), \Sigma, name]\!]\\
&\mathsf{marksof}[\![\textbf{S}\text{TX}(val, \text{DEFS}(ctx, \sigma)), \Sigma, name]\!] &&= ()\\
&\quad \text{where } name \in \text{rng}(\Sigma(\sigma))\\
&\mathsf{marksof}[\![\textbf{S}\text{TX}(val, \text{DEFS}(ctx, \sigma)), \Sigma, name]\!] &&= \mathsf{marksof}[\![\textbf{S}\text{TX}(val, ctx), \Sigma, name]\!]
\end{aligned}$$

$$\begin{aligned}
&mrk_1 \oplus (mrk_1\ mrk_2 \ldots) &&= (mrk_2 \ldots)\\
&mrk_1 \oplus (mrk_2 \ldots) &&= (mrk_1\ mrk_2 \ldots)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{strip}[\![\textbf{S}\text{TX}(atom, ctx)]\!] &&= atom\\
&\mathsf{strip}[\![\textbf{S}\text{TX}(\textbf{LIST}(stx, \ldots), ctx)]\!] &&= \textbf{LIST}(\mathsf{strip}[\![stx]\!], \ldots)
\end{aligned}$$

$$\begin{aligned}
\mathsf{nostops}[\![\xi]\!] = \{var \to transform \mid\ &\xi(var) = transform\\
&\text{and } transform \neq \text{STOP}\}
\end{aligned}$$