THE MACROLOGICAL FASCICLE

Editor's introduction to this fascicle

Note: This introduction will not appear in the final report.

Lisp was the first language to incorporate macros operating on the syntax tree of a program, rather than the text of its source code; Scheme was the first language to add automatic hygiene to such a system. This first fascicle in the development of the seventh revision of the report on Scheme therefore builds on six decades of research and innovation in which Scheme and its predecessors have played an active role throughout. Indeed, the new features added to macros in this fascicle were all pioneered in implementations of Scheme.

Hygienic macros were first mentioned in a Scheme report in an optional appendix to the R4RS. Compared to R4RS, the macro system defined by this fascicle adds only what might be called refinements of the original concept:

- a library-based system of multiple phases of evaluation and expansion makes it possible to use one's own helper procedures in the definition of macro transformers (added in R6RS, extended by this report);
- the high-level `syntax-rules` system has been unified with the low-level manipulation of syntax objects by the introduction of the `syntax-case` system, which makes pattern variables accessible as bindings within Scheme code (added in R6RS);
- macro uses can be uses of identifiers outside the operator position of a combination, like variable names (added in R6RS);
- the high-level pattern matching system used by both `syntax-rules` and `syntax-case` can match some more types of patterns that were unavailable in R4RS (added in R6RS and R7RS small);
- identifiers may have properties associated with them, which allows macros to pass around additional information associated with identifiers to one another during expansion (added in this report);

- syntax parameters allow certain types of seemingly unhygienic macro to be implemented without actually breaking hygiene (added in this report).

The R6RS divided its treatment of macros between the main report (which treated the high-level `syntax-rules` system and the binding of syntax keywords to transformers) and the report on standard libraries (which actually defined what a transformer is, as well as syntax objects and `syntax-case`). Neither document contained a complete explanation of the core semantics of macros and expansion alone. In contrast, since the R7RS volume on standard libraries (internally called Batteries) is intended to be entirely implementable in terms of the Foundations, this fascicle explains the entire macro system of R7RS Large. The standard libraries report of R6RS also presented the `syntax-case` system as a high-level system, without providing enough lower-level features to allow it to be implemented as a user library; this fascicle, however, contains sufficient primitives in the sections on syntax objects and syntax transformation to allow the `syntax-case` form to be implemented as derived syntax. Therefore, it is perhaps better not to speak of the R7RS macro system as being based on `syntax-case` as R6RS's was, but rather based on syntax objects, as the low-level system in the appendix to the R4RS was.

Since extensions to Scheme macros beyond the `syntax-rules` system provided by the small language have been among the most controversial proposals made for changes in the Scheme language in the last two decades, I hope this fascicle will address these controversies by showing how the popular `syntax-rules` system is built up, from a theoretical hygiene model, through an implementation of the model on syntax objects processed by transformer procedures; and how finally, with the addition of the pattern matcher from `syntax-rules`, a specification of the entire `syntax-case` system is reached, with `syntax-rules` itself being a trivial transformation thereof. The tools provided in the first three chapters of this fascicle are sufficient to implement the high-level systems of R5RS and R6RS specified in chapters 4 and 5. A sample implementation which demonstrates this will be provided in due course.

What implementations need to do to support this fascicle

Implementations which do not support `syntax-case` as specified by the R6RS will need to adopt it, either by completely replacing their expanders or by adapting their existing ones. Experience from implementations which have already made the switch shows that the former is generally the easier approach in practice. A non-normative appendix to this fascicle shows how macros written for the explicit renaming system, the most common low-level implementation of macros besides `syntax-case`, can be accommodated by expanders written for `syntax-case`; alternatively, van Tonder (2006) implements a version of the `syntax-case` system which includes native support for a similar version of explicit renaming to that shown in the appendix.

This fascicle deprecates the R6RS's provisions for explicit phasing. Implementations which use explicit phasing and restrict all identifier bindings to the phase at which they were created will need to switch to implicit phasing (ignoring the phase declarations on imports specified by the R6RS) and allow uses of syntax defined in previous phases.

Compared to the versions of `syntax-case` and `syntax-rules` in the R6RS, this fascicle allows renaming the ellipsis (section 4.5) as well as using the ellipsis and underscore as pattern literals, as in the small language report's `syntax-rules`. Existing implementations of the pattern matcher will need to be extended to support these features.

This fascicle further extends the R6RS macro system with the addition of lexically-scoped identifier properties (section 2.5) and syntax parameters (section 2.3), both of which require support from the expander.

This fascicle also adds a number of procedures and syntax forms from the R4RS low-level macro system for which the R6RS offered no equivalent, including `quote-syntax` (equivalent to R6RS's `syntax` without pattern variable substitution). Some of these can be implemented portably in terms of the R6RS higher-level constructs.

The final division of the R7RS Large Foundations into libraries will be decided at a later stage. The library name `(r7rs-drafts macro-fascicle)` is assigned to a temporary library containing all bindings specified in this fascicle, intended for

experiments only. These bindings may change incompatibly if this fascicle is updated before the final report is issued. Production code should not depend on this library, and implementations should not support it any more once the final R7RS Large specification is ratified.

Changes in this fascicle compared to the source texts

The main source for the contents of this fascicle is the Yellow Ballot on macros and syntactic constructs held between October 2021 and February 2022 under the chairship of John Cowan. That ballot resulted in the adoption into R7RS Large of the R6RS Standard Libraries chapter on `syntax-case`, the R6RS `identifier-syntax` transformer specifier, and the SRFIs 139 (syntax parameters), 188 (`splicing-let-syntax` and `splicing-letrec-syntax`), and 213 (identifier properties), besides a number of other proposals which will be incorporated into future fascicles.

Compared to R7RS small and the source documents adopted under the Yellow Ballot, the following substantive changes and additions have been made:

- The R6RS hygiene model is expressed in different terms and in more detail.
- Low-level procedures and syntax forms originally from the R4RS appendix have been added to address criticism that, in R6RS, the high-level `syntax-case` pattern matcher offered the only tool to destructure syntax objects. Using these forms together with identifier properties, `syntax-case` itself can be implemented as derived syntax. In addition, the predicate `symbolic-identifier=?` from the R6RS examples is defined, because it is part of the operation of R7RS small `cond-expand`.
- The behaviour of identifier properties has been nearly completely respecified to be more explicit about how properties are attached to identifiers under export, import, and shadowing. The new specification matches the behaviour of existing implementations.
- An `identifier-defined?` procedure allows detecting whether a particular identifier is bound. Among other uses, this makes identifier properties more ergonomic to use in some cases, and allows macro transformers to report

more useful error messages when they operate on an identifier passed to them which is supposed to already be bound.

- The R6RS provisions for explicit phasing have been dropped because they proved unpopular with implementers and users alike. The behaviour of implicit phasing with regards to the evaluation contexts of macro transformers and the visibility of identifiers in different phases is specified. A future fascicle on the library system will complete the definition of implicit phasing and mark the syntactic provision of the R6RS for explicit phasing as deprecated.

- The expansion process defined by R6RS has been adapted for the new macro features in this fascicle. The current draft also shows what the expansion process would look like if mixing definitions and expressions in any order in any body were allowed, but a final decision on this has not yet been made. In the event this change is not made to bodies other than program and library bodies, reverting to the old semantics is a relatively small revision to this part of the text.

- It has been explicitly specified that, although `syntax-case` and `syntax-rules` use `free-identifier=?` to find instances of literals within their input forms, they use `bound-identifier=?` to find literals within patterns. This reflects the consensus of the Scheme community and the current practice of implementations, following difficulties with advanced macro-defining macros when `free-identifier=?` is used for both purposes (Clinger and Wand 2020, sec. 14.2, though note that they incorrectly refer to matching pattern literals in uses, rather than within patterns).

- The rules for when syntax objects are wrapped vs. unwrapped as a result of evaluating a `syntax` expression have been changed to more accurately reflect both the intention of the original R6RS authors (expressed in SRFI 93 but omitted from the final R6RS document) and what existing implementations actually do. (E.g. under the letter of the rules according to R6RS, an expression such as `#'(x ...)` was not actually guaranteed to evaluate to a proper list.)

- The form `custom-ellipsis` has been added, allowing the ellipsis renaming feature of R7RS small `syntax-rules` to be implemented in terms of `syntax-case`.

- `syntax-rules` is now specified mostly in terms of the semantics of `syntax-case` and `syntax`. In particular, syntax templates inherit the R6RS feature of allowing multiple ellipses after a single pattern variable, already a common extension to the `syntax-rules` system of R7RS small.

- A new declarative form `erroneous-syntax` allows more concisely defining macros which always signal an error when expanded. Syntax keywords with this property are expected to be increasingly common for use as syntax parameters and as the keys of identifier properties, besides their existing uses as auxiliary syntax keywords.

The R6RS condition system has also been implicitly adopted, following informal consensus of WG2 members.

THE MACROLOGICAL FASCICLE

Editorial conventions

The phrase ‘it is an error’ as traditionally used in Scheme reports has been retired in this fascicle. In its place we have adopted a three-way distinction between requirements on implementations, which should be familiar from the specifications of other languages including Common Lisp and C.

- ‘Undefined’ behaviour (and similar phrases such as ‘the behaviour is undefined …’) is the direct equivalent of ‘it is an error’ in the R7RS small and R6RS language reports. An implementation is allowed to behave in any way at all if instructed to evaluate code with undefined behaviour; however, implementers should be aware of the R6RS guarantee of ‘safety’, a version of which is expected to be applied to the libraries defined by the R7RS large language reports, including to situations which involve undefined behaviour.

- ‘Unspecified’ behaviour refers to situations in which the report allows implementations to choose one of a number of behaviours explicitly allowed by the report. Implementations are not required to choose the same option in all circumstances, nor are they required to document their choice.

- ‘Implementation-specified’ behaviour refers to situations in which the report allows implementations to choose between possible behaviours, but does require conforming implementations to document which behaviour they use.

As in the R7RS small language report, the phrase ‘an error is signalled’ continues to refer to a situation in which an exception is required to be raised.

The R7RS Large Foundations will incorporate a revised version of the R6RS condition types which is compatible with the error objects defined by the small language report, but this is not yet specified and will be included in a future fascicle. However, the phrase ‘it is a syntax violation’ or ‘a syntax violation is signalled’ means that an exception is raised with condition type `&syntax` as in the R6RS.

The phrase ‘domain error’ (or ‘it is a domain error’) refers to using a procedure with an argument value which is outside the specified range of possible values for

that argument. It is not yet specified what concrete action implementations should take in this case in the R7RS Large Foundations. Preliminarily, implementations are encouraged to signal an error with condition type `&assertion` as in the R6RS, but this is not a requirement. The circumstances, if any, in which a domain error is required to signal an error will likewise be established in a future fascicle.

The following variable names are used in the specifications of procedures to imply the type of the argument named by that variable:

Variable	Type
id	identifier
proc	procedure
stx	syntax object
symbol	symbol

[*Editorial note:* Cross reference each entry in the right hand column to where that type is defined in the text, where possible.]

It is a domain error if an argument to any procedure does not match the expected type, whether the expected type is implied by the use of a variable name listed in this table, or named explicitly in the specification text.

The key words ‘must’, ‘must not’, ‘required’, ‘shall’, ‘shall not’, ‘should’, ‘should not’, ‘recommended’, ‘may’, and ‘optional’ in this document, although not capitalized in this report, are to be interpreted as described in RFC 2119. (Bradner 1997)

Comments of the form ‘[*Editorial note: ...*]’ are editorial notes marking places requiring revision in the final report.

THE MACROLOGICAL FASCICLE

CHAPTER 1

Macros and hygiene

Scheme programs and libraries can define and use new derived expression types, called macros. Each instance of a macro is called a use of the macro. Macro uses can take similar forms to any expression type defined in this report which is associated with a binding to an identifier.

Implementations of Scheme can evaluate such expression types because the definition of a macro binds an identifier, which is called the keyword or syntax keyword and which uniquely determines the expression type, to a procedure written in ordinary Scheme code which, when called with an argument representing the form of a macro use, can analyse that macro use and transcribe it into a more primitive expression. The Scheme implementation can then evaluate this new expression, possibly after transcribing additional macro uses within it into yet more primitive expressions. The procedures associated with macro definitions are called transformers. The process of transcribing macro uses using transformers within a program is called expansion, and the more primitive expression resulting from calling the transformer on the form of a macro use is called the expansion of that macro. The part of a Scheme implementation responsible for the process of expansion is thus called the expander.

This report defines two closely-related high-level systems for writing transformers (`syntax-case` and `syntax-rules`). Within these high-level systems, macros defined by the users are by default 'hygienic' and 'referentially transparent' and thus preserve Scheme's lexical scoping. (Kohlbecker 1986, Kohlbecker et al. 1986, Bawden and Rees 1988, Clinger and Rees 1991, Dybvig, Hieb and Bruggeman 1992)

- If a macro transformer inserts a binding for an identifier, the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers.

- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that surround the use of the macro.

Besides these high-level systems, this report also defines a low-level mechanism by which these properties are maintained, and which allows them to be deliberately broken in a controlled way. It is also possible to use these lower-level primitives within transformers which use the high-level `syntax-case` system, in order to write macros which deliberately break these properties.

In order to both maintain these properties by default and allow them to be broken when required, an implementation of Scheme contains a concrete implementation of a theoretical model of hygiene, which allows it to keep track of when and where an identifier was introduced, even if it is inserted elsewhere in a program by a macro expansion.

1.1. Defining hygiene

Barendregt (1984)'s hygiene condition for the lambda calculus is an informal notion that requires the free variables of an expression *N* that is to be substituted into another expression *M* not to be captured by bindings in *M* when such capture is not intended. Kohlbecker et al. (1986) propose a corresponding hygiene condition for macro expansion that applies in all situations where capturing is not explicit: 'Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step'. In the terminology of this report's model of hygiene, the generated identifiers are those introduced by a transformer rather than those present in the form passed to the transformer, and a macro transcription step corresponds to a single call by the expander to a transformer. Also, the hygiene condition applies to all introduced bindings rather than to introduced variable bindings alone.

This leaves open what happens to an introduced identifier that appears outside the scope of a binding introduced by the same call. Such an identifier refers to the lexical binding in effect where it appears inside the transformer body or one of the helpers it calls. This is essentially the referential transparency property described

by Clinger and Rees (1991). Thus, the hygiene condition can be restated as follows:

> A binding for an identifier introduced into the output of a transformer call from the expander must capture only references to the identifier introduced into the output of the same transformer call. A reference to an identifier introduced into the output of a transformer refers to the closest enclosing binding for the introduced identifier or, if it appears outside of any enclosing binding for the introduced identifier, the closest enclosing lexical binding where the identifier appears inside the transformer body or one of the helpers it calls.

Within the context of hygienic macro expansion, it is sometimes useful to write macros which deliberately violate this condition, either by introducing bindings for identifiers which capture the references of identifiers introduced outside of the same transformer call, or by introducing identifiers which refer to bindings present at the macro use site. Since uses of such non-hygienic macros may occur within the output of transformer calls for other macros, the transformers for non-hygienic macros must limit the extent of this capture to identifiers that were either introduced into the output of a specific transformer call, or which where introduced by and present in the original forms before expansion began. In other words, identifiers introduced in this way must be treated as if they had been implicitly present in a form generated by some specific macro transcription step, or in a form in the original source.

1.2. Modelling hygiene

> *Todo:* Express the hygiene model in notation as well as in prose. (Needs a better markup system for the spec than the current mess.)

Note: This section describes a possible operational semantics of one theoretical model of hygiene. Implementations are not required to adopt this exact model, as long as whichever model they use repects the hygiene condition defined above. The order in which implementations actually analyse macro uses within a library or program is defined in section 2.6.

In the process of macro expansion, Scheme code is represented as datums (as if a `quote` expression surrounding the code had been evaluated) which are wrapped in a hygienic context. This context propagates down to each individual symbol in the datum tree, each of which is ultimately wrapped. A single symbol wrapped with context is an identifier. However, at the beginning of expansion, an entire datum is wrapped in one context. As the analysis and expansion of macro uses progresses, the hygienic context is pushed down to new wraps on the datums contained within the original datum. This allows the expander to efficiently maintain and update the same hygienic context for a large tree of Scheme code at once, copying it only when required.

The hygienic context contained in a wrap consists of a history, which is a set of time-stamps, and a lexical environment, which keeps track of identifiers' lexical addresses. The history is used in the expansion process to keep track of when an identifier was introduced: the time-stamps within the history are of two types, recalling respectively the entry and exit of identifiers into and out of a transformer when it is called in a single macro transcription step. The lexical environment keeps track of where in the code identifiers were introduced and bound, including which binding each identifier has.

Note: In the R6RS, the history was referred to as the marks, and each time-stamp as one mark (for a time-stamp recording the completion of a macro transcription step) or an anti-mark (for a time-stamp recording the beginning of a macro transcription step). The lexical environment was referred to as a set of substitutions, and a lexical address as the label on one substitution. The model defined by this report is semantically identical, but the terminology has been changed (in the case of the term 'time-stamp', reverted to the original terminology of Kohlbecker) in the hope of improving the clarity of the definition to readers. Other authors have also referred to time-stamps as colours or renames, and to lexical addresses as bindings or binding names.

The expander can create a new wrap on a datum with an existing history and lexical environment. The expander can also unwrap a layer of the syntax tree, a process which consists of creating a new instance of the same type of datum contained within a wrap and filling it with new wraps, each with the contents of the corresponding parts of the original datum and a copy of the hygiene information

within the original wrap. Unwrapping successive layers of the syntax tree allows macro transformers to parse their input.

When the expander encounters a core binding construct such as `lambda` or `splicing-let-syntax`, it extends the hygiene context in its wrap by adding new lexical addresses for each of the identifiers which the construct binds. The expander also maintains a global binding store mapping lexical addresses to their expand-time values: if the core binding construct binds syntax, it also updates this binding store mapping the new lexical address to the evaluated transformer for the new syntax.

When the expander encounters a macro use, it looks up the transformer associated with the macro use's syntax keyword in the global binding store. It then adds a new time-stamp to the history of the wrap for the macro use, which records the beginning of a new macro transcription step, and calls the transformer with the macro use in its new wrap.

The transformer procedure is allowed to return a datum which is not wrapped in hygienic context, as long as all identifiers carry hygienic context — i.e., as long as no symbols occur in any non-wrapped subtree of the datum. In order to correctly record the end of the macro transcription step, therefore, the expander must add a time-stamp recording the end of the same macro transcription step to every wrap within the returned syntax object.

The effect of this process is that identifiers which were present anywhere in the input form of the macro use are recalled with both the time-stamp for the beginning of a transcription step and the time-stamp for the end of the step in their histories, while identifiers which were introduced by the transformer are recalled only with the time-stamp for the end of the transcription step. When histories are compared, time-stamps for the beginning and end of the same transcription step are both ignored, and both may be discarded. This allows identifiers introduced by the transformer call to be distinguished from identifiers in the input form, and more generally allows identifiers introduced by macros to be distinguished from macros within the original source code.

An identifier therefore has two kinds of name: the symbolic name, which is the symbol datum wrapped by the identifier, and the time-stamped name, which is the symbolic name plus the history. In most contexts, identifiers in Scheme treated as if their name were the time-stamped name, not the symbolic name.

THE MACROLOGICAL FASCICLE

CHAPTER 2

Syntax transformation

In order to evaluate a Scheme program, the macro uses within the program must be expanded into core forms which the implementation can directly further process itself. The set of core forms is implementation-dependent: an implementation may consider any syntax form defined by this report to be either a core form or a macro; the difference is not observable to users. Uses of both macros and core forms are represented as syntax objects.

Macro uses and core forms are both distinguished in forms to be processed by syntax keywords. These keywords occupy the same namespace as variables. That is, within the same scope, an identifier can be bound as a variable or keyword, or neither, but not both, and local bindings of either kind may shadow other bindings of either kind. In order to use a macro or a core form, the corresponding keyword must be imported into a program or library, or introduced within it by means of special macro definition and binding forms.

Many uses of macros defined by the user have the form:

(⟨keyword⟩ ⟨datum⟩ ...)

where ⟨keyword⟩ is an identifier which lexically refers to the binding established for the macro or core form. Macro uses can also take the form of improper lists, bare identifiers, or `set!` forms, where the second subform of the `set!` is the keyword:

(⟨keyword⟩ ⟨datum⟩ ... . ⟨datum⟩)

⟨keyword⟩

(`set!` ⟨keyword⟩ ⟨datum⟩)

In the latter case, the `set!` keyword must lexically refer to the same binding as the `set!` form defined in this report, and the binding of ⟨keyword⟩ must explicitly

allow this kind of macro use (see section 2.4).

Macros whose uses can take the form of bare identifiers are referred to as identifier macros.

2.1. Transformers

Syntax keywords are bound by user code to transformers.

Most transformers are ordinary Scheme procedures, called transformer procedures, which receive exactly one argument, a syntax object (see section 3) representing the form of a macro use, and return exactly one value, a syntax object representing the result of expanding the input macro use. The result of expanding the input form using the transformer procedure replaces the macro use in the place where it occurred.

Variable transformers (section 2.4) are another kind of transformer.

It is undefined behaviour to re-enter the dynamic extent of a call to a transformer by the expander after it has returned once.

2.2. Syntax definition and binding forms

Note: The examples in this section use the `syntax-rules` system to create transformers, which is defined in section 5.

`(define-syntax` ⟨keyword⟩ ⟨transformer expression⟩`)` syntax

`Define-syntax` binds syntax keywords in a manner analogous to how `define` binds variables. ⟨Transformer expression⟩ must be an expression that evaluates at expand time to a transformer. The ⟨keyword⟩ is then bound as a syntax keyword to this transformer during the process described in section 2.6. The created binding is visible throughout the body where `define-syntax` is used, unless shadowed by another binding construct within the body.

Examples:

```
(let ()
  (define even?
```

```
    (lambda (x)
      (or (= x 0) (odd? (- x 1)))))
  (define-syntax odd?
    (syntax-rules ()
      ((odd? x) (not (even? x)))))
  (even? 10))
```

⇒ `#t`

An implication of the left-to-right processing order (section 2.6) is that one definition can affect whether a subsequent form is also a definition.

```
(let ()
  (define-syntax bind-to-zero
    (syntax-rules ()
      ((bind-to-zero id) (define id 0))))
  (bind-to-zero x)
  x)
```

⇒ `0`

`(splicing-let-syntax` ⟨syntax bindings⟩ syntax
⟨definition or expression⟩ `...)`

Syntax: ⟨Syntax bindings⟩ has the form:

`(`(⟨keyword⟩ ⟨transformer expression⟩)` ...)`

⟨Transformer expression⟩ is as for `define-syntax`. It is a syntax violation if the same identifier (in the sense of `bound-identifier=?`) appears as the ⟨keyword⟩ of more than one of the ⟨syntax bindings⟩.

Semantics: The ⟨definition or expression⟩ forms are expanded in a syntactic environment containing the bindings of the syntactic environment of the `splicing-let-syntax` form with additional bindings created by associating each of the ⟨keyword⟩s as syntax keywords to transformers obtained by evaluating the corre-

sponding ⟨transformer expressions⟩. The evaluation of the ⟨transformer expression⟩s takes place within the lexical environment where the `splicing-let-syntax` form appears.

The ⟨definition or expression⟩ forms are treated as if wrapped in an implicit `begin`; thus definitions created as a result of expanding the forms have the same extent a definition which appeared in the place of the `splicing-let-syntax` would have.

Example:

```
(let ((x 21))
  (splicing-let-syntax
      ((def (syntax-rules ()
              ((def stuff ...) (define stuff ...)))))
    (def foo 42))
  foo)
```

⇒ 42

Note: This form was called `let-syntax` in R6RS and had the additional restriction that the forms had to be either definitions or expressions, but not both.

`(splicing-letrec-syntax` ⟨syntax bindings⟩ syntax ⟨definition or expression⟩ `...)`

Syntax: Same as for `splicing-let-syntax`.

Semantics: The ⟨definition or expression⟩ forms are expanded in a syntactic environment containing the bindings of the syntactic environment of the `splicing-letrec-syntax` form with additional bindings created by associating each of the ⟨keyword⟩s as syntax keywords to transformers obtained by evaluating the corresponding ⟨transformer expressions⟩. The evaluation of the ⟨transformer expression⟩s takes place within a lexical environment which contains the bindings of the ⟨keyword⟩s themselves, so the transformers can transcribe forms into uses of the macros introduced by the `splicing-letrec-syntax` form. It is undefined behav-

iour if the evaluation of any of the ⟨transformer expressions⟩ requires knowledge of the actual transformer bound to one of the ⟨keyword⟩s.

As for `splicing-let-syntax`, the ⟨definition or expression⟩ forms are treated as if wrapped in an implicit `begin` and can expand into definitions visible outside of the `splicing-letrec-syntax` form itself.

Note: This form was called `letrec-syntax` in R6RS and had similar restrictions on its contents to that report's `let-syntax`, as described above.

`(let-syntax` ⟨syntax bindings⟩ ⟨body⟩`)` syntax

Syntax: The ⟨syntax bindings⟩ are the same as for `splicing-let-syntax` and `splicing-letrec-syntax`.

Semantics: The syntactic environment in the location of the `let-syntax` expression is extended by new syntax keyword bindings in the manner of `splicing-let-syntax` and the ⟨body⟩ expanded within that environment. `Let-syntax` differs from `splicing-let-syntax` in that it creates a new lexical body which is not spliced into a surrounding body: definitions within the ⟨body⟩ are not visible outside of the extent of the ⟨body⟩ itself.

Example: Compare this example with the example under `splicing-let-syntax`.

```
(let ((x 21))
  (let-syntax
      ((def (syntax-rules ()
              ((def stuff ...) (define stuff ...)))))
    (def foo 42))
  foo)
```

⇒ 21

Implementation:

```
(define-syntax let-syntax
  (syntax-rules ()
```

```
((_ bindings body_0 body_1 ...)
 (splicing-let-syntax bindings
   (let () body_0 body_1 ...))))
```

Note: This form is the same as the `let-syntax` in the small language report, but not the same as `let-syntax` from the R6RS (see the remark under `splicing-let-syntax`). The `(scheme base)` library must export the same binding [*Editorial note:* as whatever large language library this ends up in.]

`(letrec-syntax` ⟨syntax bindings⟩ ⟨body⟩`)` syntax

Syntax: The ⟨syntax bindings⟩ are the same as for `splicing-let-syntax` and `splicing-letrec-syntax`.

Semantics: The syntactic environment in the location of the `letrec-syntax` expression is extended by new syntax keyword bindings in the manner of `splicing-letrec-syntax` and the ⟨body⟩ expanded within that environment. `Letrec-syntax` differs from `splicing-letrec-syntax` in that, like `let-syntax`, it creates a new lexical body which is not spliced into a surrounding body.

Example:

```
(letrec-syntax
    ((xor
      (syntax-rules ()
        ((_) #f)
        ((_ e)
         (if e #t #f))
        ((_ e_1 e_2 ...)
         (let ((temp e_1))
           (if temp
               (not (or e_2 ...))
               (xor e_2 ...)))))))
```

```
  (values (xor #t #f #f)
          (xor #t #t #f)))
```

⇒ `#t #f`

Implementation:

```
(define-syntax letrec-syntax
  (syntax-rules ()
    ((_ bindings body_0 body_1 ...)
     (splicing-letrec-syntax bindings
       (let () body_0 body_1 ...)))))
```

Note: This form is the same as the `letrec-syntax` in the small language report, but not the same as `letrec-syntax` from the R6RS (see the remark under `splicing-let-syntax`). The `(scheme base)` library must export the same binding [*Editorial note:* as whatever large language library this ends up in.]

2.3. Syntax parameters

Syntax parameters are a minor variation on ordinary syntax keyword bindings. They provide a mechanism for rebinding a macro definition within the dynamic extent of a macro expansion.

Among other uses, this provides a convenient solution to one of the most common types of unhygienic macro: those that reintroduce the same unhygienic binding each time the macro is used. With syntax parameters, instead of introducing the binding unhygienically each time, one instead creates a single binding for the keyword, which is adjusted when the keyword is supposed to have a different meaning. As no new bindings are introduced, hygiene is preserved. Using a syntax parameter also provides the advantage that the identifier for the binding can be renamed when it is imported, if the macro user so wishes.

`(define-syntax-parameter` ⟨keyword⟩ ⟨transformer expression⟩`)` syntax

Binds ⟨keyword⟩ as a parameterizable syntax keyword, using the transformer created by evaluating the ⟨transformer expression⟩ at expand time as the default

transformer. When ⟨keyword⟩ is used outside the context of a `syntax-parameterize` body, the result is equivalent to if that ⟨keyword⟩ had been defined using `define-syntax`.

`Define-syntax-parameter` is similar to `define-syntax`, except the created binding is marked as parameterizable.

`(syntax-parameterize ((`⟨keyword⟩ ⟨transformer expression⟩`) ...)` syntax ⟨body⟩`)`

Adjusts the ⟨keyword⟩s to use the transformer obtained by evaluating the corresponding ⟨transformer expression⟩s when the keywords are used within the expansion of the ⟨body⟩. It is a syntax violation if any of the ⟨keyword⟩s refer to bindings that are not parameterizable syntax keyword bindings.

`Syntax-parameterize` differs from `let-syntax` in that the binding is not shadowed, but adjusted, and so uses of the ⟨keyword⟩s in the expansion of ⟨body⟩ use the new transformers.

Example: The following example defines a form `lambda^` which automatically makes an early-return procedure called `return` available within its body.

```
(define-syntax-parameter return
  (erroneous-syntax "return used outside of lambda^"))

(define-syntax lambda^
  (syntax-rules ()
    ((lambda^ formals body_0 body_1 ...)
     (lambda formals
       (call-with-current-continuation
        (lambda (escape)
          (syntax-parameterize
              ((return (identifier-syntax escape)))
            body_0 body_1 ...)))))))
```

> *Todo:* This example will probably need changing to use delimited control operators, once it is decided what form those will take in the Foundations.

2.4. Variable transformers

Variable transformers are another kind of transformer besides transformer procedures. A variable transformer is a simple container for a procedure, created by calling `make-variable-transformer` on that procedure. Binding a syntax keyword to a variable transformer declares to the expander that the procedure contained within it also expects to process macro uses of the form `(set!` ⟨keyword⟩ ⟨datum⟩`)`. An attempt to expand a macro use of this form whose transformer is not a variable transformer is a syntax violation.

`(make-variable-transformer` *proc*`)` procedure

Wraps the procedure *proc* in a variable transformer and returns it.

When a syntax keyword is bound to the result of invoking `make-variable-transformer` on a transformer procedure, that transformer procedure is invoked for all macro uses with that keyword, including when the keyword is the left-hand side of a `set!` expression, which would otherwise be a syntax violation.

Rationale: If `set!` worked as described for all macro transformer procedures, many macros would mistakenly process `set!` forms as if they were macro uses with the keyword in the operator position, and could actually produce a result if their usual syntax happened to be of the approximate form `(`⟨keyword⟩ ⟨identifier⟩ ⟨expression⟩`)`. The result of that expansion might then turn out to be valid Scheme code, creating unexpected behaviour in a program whose cause might be difficult to discover. All macros would have to check explicitly for the comparatively rare `set!` case to guard against it. By centralizing this check within the macro expander, requiring transformers which actually expect to process `set!` forms to explicitly declare this fact, this kind of programming error becomes impossible.

Example:

```
(define-syntax used-as
  (make-variable-transformer
   (lambda (stx)
     (cond ((identifier? stx)
            (quote-syntax (quote reference)))
           ((free-identifier=? (car (unwrap-syntax stx)) #'set!)
            `(,(quote-syntax cons)
              ,(quote-syntax (quote assignment))
              (,(quote-syntax quote)
               ,(cdr (unwrap-syntax
                      (cdr (unwrap-syntax stx)))))))
           (else
            `(,(quote-syntax cons) ,(quote-syntax (quote combination))
                                   (,(quote-syntax quote)
                                   ,(cdr (unwrap-syntax stx)))))))))
```

```
used-as
⇒ reference

(set! used-as x)
⇒ (assignment x)

(used-as y)
⇒ (combination y)
```

2.5. Identifier properties

During expansion, a set of properties can be associated with each identifier in a Scheme program. This allows arbitrary information to be associated with identifiers, which can be used by macro transformers to inform their treatment of particular identifiers. For example, the sample implementation of the `syntax-case` pattern matcher included with this report uses identifier properties to implement pattern variables.

Each property defined on an identifier associates a key (which must also be an identifier) with a value (which may be any object). When an identifier binding is created by definition or by a local binding construct, it is associated with a new, empty set of identifier properties. If the identifier bound shadows one from a containing lexical context, the identifier properties on the shadowed identifier effectively become hidden within the lexical extent of the new binding, in the same way its binding is hidden.

When an identifier property is defined on an identifier, the property belongs only to the lexical scope in which that property is defined. The property itself may shadow properties created on the same identifier and with the same key in containing lexical contexts.

When an identifier is imported from a library, it brings with it a copy of the set of identifier properties that were defined on it in that library. Additional identifier properties may be defined on it, and properties from the original library may be redefined within the context in which the identifier was imported, without those definitions or redefinitions being visible in the original library. If the identifier was imported into a library which subsequently re-exports it, the re-exported version has the identifier properties as they were (re-)defined in the library which re-exports it. If the same binding is then imported into another context from both the original and the re-exporting library, or from multiple re-exporting libraries which each defined their own properties on the identifier, the identifier in that context has a set of properties which is the union of the properties from all the libraries it is imported from. If two properties with the same key are imported on the same identifier, and the values of the properties are not the same in the sense of `eqv?`, it is an import error.

Note: Though identifier properties are superficially similar to a classical Lisp feature known as symbol property lists, the two are quite different, even though they can sometimes be used for the same purposes. A symbol property list is typically held globally, unlike identifier properties, which are lexically scoped to where they were defined.

> *Todo:* Define the interaction of identifier properties with phasing.

`(define-property ⟨identifier⟩ ⟨key⟩ ⟨expression⟩)` syntax

Syntax: Both ⟨identifier⟩ and ⟨key⟩ must be bound identifiers.

Semantics: The ⟨expression⟩ is evaluated at expand time to produce a single value, and an identifier property is defined on the ⟨identifier⟩ associating the ⟨key⟩ with this value.

Operationally, when the expander encounters a `define-property` form, it creates a new lexical address within the lexical environment for a tuple of the ⟨identifier⟩ and the lexical address for the binding of the ⟨key⟩. It then stores the result of evaluating the ⟨expression⟩ in its global binding store under the new address.

`(identifier-property` *id key*`)` procedure
`(identifier-property` *id key default*`)` procedure

Returns the identifier property associated with the identifier *id* whose key has the same binding as *key*. If there is no such property, it returns *default*, or `#f` if no *default* argument was provided. If either *id* or *key* is not bound, a syntax violation is signalled.

The `identifier-property` procedure can only be called within the dynamic extent of a call by the expander to a transformer. If it is called in other situations, it is unspecified whether the procedure will work as intended, or act as if *id* or *key* or the property requested is unbound, or will signal an error [*Editorial note:* an assertion violation].

Operationally, `identifier-property` first finds the lexical addresses a_{id} and a_{key} of *id* and *key* respectively, then finds the lexical address a_{prop} in the lexical environment of *id* for the tuple of *id* and these lexical addresses. Finally, it looks up the address a_{prop} in the global binding store and returns the value associated with it.

Note: Two identifiers which share the same binding will not necessarily have the same identifier properties: `free-identifier=?` is used to match identifier keys but not the identifiers themselves in the binding store when looking up identifiers. This can occur when an identi-

fier property's value is shadowed, or when a binding is imported into multiple libraries or under multiple names, as in the following example.

```
(import (scheme base)
        (rename (only (scheme base) cons) (cons make-pair)))

(define-syntax renamed?
  (erroneous-syntax "only an identifier property key"))
(define-property make-pair renamed? #t)

(define-syntax both-renamed?
  (lambda (stx)
    (and (identifier-property #'cons #'renamed?)
         (identifier-property #'make-pair #'renamed?))))

(values (free-identifier=? #'cons #'make-pair)
        (both-renamed?))
```

⇒ #t #f

2.6. Expansion process

In order to expand a body (whether library, program, or other body), the expander processes the initial forms within from left to right. How the expander processes each form encountered depends upon the kind of form.

[*Editorial note:* The following has been formulated based on the equivalent expansion process defined by the R6RS, but assuming that R7RS will relax the restriction on the order of definitions and expressions in all bodies. R7RS already relaxed the restriction in library bodies compared to R6RS. If the restriction is not relaxed within regular bodies, only a small adjustment to the text, reverting to R6RS semantics for those bodies, is required.]

[*Editorial note:* This process does not define semantics compatible with those prescribed for program bodies by the small language. Those semantics will be specified in a future fascicle.]

macro use

The expander invokes the associated transformer to transform the macro use, then recursively performs whichever of these actions are appropriate for the resulting form.

`define-syntax` or `define-syntax-parameter` form

The expander expands and evaluates the right-hand-side expression and binds the keyword to the resulting transformer.

`define` form

The expander records the fact that the defined identifier is a variable but defers expansion of the right-hand-side expression until after all of the forms in the body have been processed.

`define-property` form

The expander expands and evaluates the value expression and creates or replaces a property for the key on the given identifier, associating it with the resulting value.

`begin` form

The expander splices the subforms into the list of body forms it is processing.

`splicing-let-syntax` or `splicing-letrec-syntax` form

The expander splices the inner body forms into the list of (outer) body forms it is processing, arranging for the keywords bound by the `splicing-let-syntax` and `splicing-letrec-syntax` to be visible only in the inner body forms.

expression, i.e., nondefinition

The expander defers the expansion of the expression until after all the forms in the body have been processed.

Once the rightmost form in the body has been processed, the expander makes a second pass over the forms deferred as the right-hand sides of variable definitions or as nondefinitions.

Note that this algorithm does not directly reprocess any form. It requires a single left-to-right pass over the definitions followed by a single pass (in any order) over the body expressions and deferred right-hand sides.

The behaviour is undefined if any definition in the sequence of forms to define any identifier whose binding is used to determine the meaning of the undeferred portions of the definition, or of any definition that precedes it in the sequence of forms. Similarly, the behaviour is undefined if the evaluation of any form in the sequence of forms uses or assigns the value of a defined variable whose definition is to the right of that form. For example, the behaviour of each of the following examples is undefined:

```
(define define 3)
```

```
(begin (define begin list))
```

```
(display (+ x 16))
(define x 32)
```

[*Editorial note:* Last example should be within a `(let () ...)` if the relaxation is accepted for all bodies.]

The behaviour of the following example is not undefined, because the body of the internal `increase` procedure will not be evaluated by a call to it until after the `value` variable it closes over has been defined:

```
(define (make-counter)
  (define (increase)
    (set! value (+ value 1))
    value)
  (define value 0)
  increase)
```

2.7. Phases of evaluation and macro expansion

The algorithm for processing forms in bodies outlined above requires the expressions creating macro transformers to be evaluated before evaluation of the Scheme program as a whole can proceed. The environment in which such evaluation takes place is defined by dividing the evaluation of Scheme programs into phases. Each phase is identified by a non-negative integer, and the number of the phase in which evaluation is currently taking place at any time is denoted ϕ. If a macro definition appears in phase n code, then its right-hand-side expression is evaluated in phase $n + 1$. The expansion and evaluation of Scheme forms after all syntax keywords have been defined takes place at phase 0; thus, at the top level of a body, the expansion and evaluation of the right-hand sides of all `define-syntax` forms and the transformer expressions of `splicing-let-syntax` and `splicing-letrec-syntax` bindings takes place at phase 1.

The environment at each phase is defined as follows. The environment at the earliest phase of evaluation contains all bindings which the program or library has imported from other libraries. All of these bindings, whether they are variables or syntax keywords, are available at all phases of evaluation. All syntactic bindings created in the course of expansion are likewise available at all phases of evaluation within the scopes in which they are visible. Variable bindings are available only in the phase in which they are created. It is undefined behaviour to either attempt to access the binding of, or to rebind an identifier which is a variable defined in a different phase.

Note: The possibility provided by the R6RS for explicit control of the availability of imported bindings at particular phases in import specs has been removed, because it proved unpopular with implementers and users.

THE MACROLOGICAL FASCICLE

CHAPTER 3

Syntax objects

Syntax objects are the means by which the hygiene model (section 1.2) is implemented in the Scheme language. They are the means by which macros written by users can obtain information about the forms used to invoke them at the site of macro use, and by which they produce their output.

A syntax object may be wrapped, as described in the hygiene model. It may also be unwrapped, fully or partially, i.e., consist of list and vector structure with wrapped syntax objects or nonsymbol values at the leaves. More formally, a syntax object is:

- a pair of syntax objects;
- a vector of syntax objects;
- a datum which is not a symbol, nor a pair, nor a vector; or
- a wrapped syntax object.

A syntax object may contain circular structures created by datum labels or by use of the `datum->syntax` procedure (section 3.2). An implementation may also consider other, non-datum values to be syntax objects, but the meaning and behaviour of such values when included in the output of transformers is not defined by this report.

The distinction between the terms ‘syntax object’ and ‘wrapped syntax object’ is important. For example, when invoked by the expander, a transformer procedure must accept a wrapped syntax object but may return any syntax object, including an unwrapped syntax object. Wrapped syntax objects are distinct from other types of values.

3.1. Identifiers

Syntax objects representing identifiers are always wrapped. A symbol which is not wrapped is never a valid syntax object.

`(identifier?` *obj*`)` procedure

Returns `#t` if *obj* is an identifier, i.e., a syntax object representing an identifier, and `#f` otherwise.

Examples:

```
(identifier? #'x)
⇒ #t

(identifier? 'x)
⇒ #f

(identifier? #'(x))
⇒ #f
```

`(identifier-defined?` *id*`)` procedure

Returns `#t` if the given *id* has a binding associated with it, or `#f` otherwise.

Operationally, `identifier-defined?` returns `#t` if the given identifier has a lexical address associated with it within its lexical environment, and `#f` otherwise.

Rationale: While it is possible to detect whether a particular identifier is bound or not using identifier properties (section 2.5), it is somewhat cumbersome to have to catch and deal with the exception raised by an attempted reference to a property on an unbound identifier. The `identifier-defined?` procedure also does not depend on the environment maintained by the expander, and can therefore be used outside of the dynamic extent of a call to a macro transformer.

In general, a test to determine whether an identifier is bound or not is useful to improve error reporting in macros which depend on some identifier named within them having been bound outside of the macro use.

Examples:

```
(identifier-defined? #'identifier-defined?)
⇒ #t
```

Assuming no identifier `x` is defined:

```
(identifier-defined? #'x)
⇒ #f

(let ((x 1)) (identifier-defined? #'x))
⇒ #t
```

`(generate-identifier)` procedure

`(generate-identifier` *symbol*`)` procedure

Returns a new identifier. The optional argument *symbol* specifies the symbolic name of the resulting identifier. The returned identifier is guaranteed not to be `bound-identifier=?` to any existing identifier. If the optional *symbol* argument is not given it should also not be `symbolic-identifier=?` to any existing identifier.

Operationally, `(generate-identifier` *symbol*`)` returns a new wrapped syntax object wrapping the *symbol* with a history containing a time-stamp for the end of a fictive macro transcription step.

`(generate-temporaries` *list-stx*`)` procedure

list-stx must be a list or syntax object representing a list-structured form.

Returns a list of generated identifiers as long as the input list *list-stx*. Each generated identifier is subject to the same requirements as imposed on `generate-identifier` when called without an argument.

Operationally, `generate-temporaries` first converts *list-stx* to a proper list, unwrapping the successive cdrs of any wrapped pairs, then calls `map` on the resulting list, generating a new identifier with `(generate-identifier)` for each item.

`(bound-identifier=?` id_1 id_2`)` procedure

Returns `#t` if a binding for one *id* would capture a reference to the other in the output of the transformer, assuming that the reference appears within the scope of the binding, and `#f` otherwise. In general, two identifiers are `bound-identifier=?` only if both are present in the original program or both are introduced by the same transformer application (perhaps implicitly — see `datum->syntax`).

Operationally, `bound-identifier=?` returns `#t` if id_1 and id_2 both have the same symbolic names and the same histories (discarding time-stamps for the beginning and end of the same macro transcription step), and `#f` otherwise.

Examples:

```
(bound-identifier=? #'x #'x)
⇒ #t

(bound-identifier=? #'x #'y)
⇒ #f

(bound-identifier=? (generate-identifier 'x)
                    (generate-identifier 'x))
```

```
⇒ #f
```

`(symbolic-identifier=?` id_1 id_2`)` procedure

Returns `#t` if the two *id*s have the same symbolic name, and `#f` otherwise.

Rationale: An example definition for this procedure was given in the R6RS, but the procedure was not actually made part of any library. Since it is part of the operation of `free-identifier=?` and is needed to implement the small language's `cond-expand`, it is provided here.

Examples:

```
(symbolic-identifier=? (generate-identifier 'x)
                       (generate-identifier 'x))
```

```
⇒ #t

(symbolic-identifier=? #'x #'y)
⇒ #f
```

Implementation:

```
(define (symbolic-identifier=? id_1 id_2)
  (symbol=? (syntax->datum id_1)
            (syntax->datum id_2)))
```

(`free-identifier=?` id_1 id_2) procedure

Returns `#t` if the bindings for the two *id*s would refer to the same lexical binding if inserted as free identifiers in the output of the transformer. If either of the *id*s is not lexically bound, return `#t` if they are `symbolic-identifier=?`. Otherwise, return `#f`.

Operationally, `free-identifier=?` returns `#t` if id_1 and id_2 map to the same lexical address within their respective lexical environments, or if neither maps to any lexical address and their symbolic names are the same, and `#f` otherwise.

`Free-identifier=?` can be used within transformers to find uses of auxiliary syntax keywords.

Examples:

```
(import (scheme base)
        (rename (scheme base)
                (else otherwise)))
(free-identifier=? #'else #'otherwise)
⇒ #t

(free-identifier=? #'else #'=>)
⇒ #f
```

The following examples show that unbound identifiers compare the same if they have the same symbolic names. The examples assume that no identifier `x` is defined.

```
(free-identifier=? (generate-identifier 'x)
                   (generate-identifier 'x))
```

⇒ `#t`

```
(free-identifier=? #'x (generate-identifier 'x))
```

⇒ `#t`

```
(let ((x 1))
  (free-identifier=? #'x (generate-identifier 'x)))
```

⇒ `#f`

3.2. Wrapped syntax objects

`(quote-syntax` ⟨syntactic datum⟩`)` syntax

Syntax: The ⟨syntactic datum⟩ is either an identifier, or a datum which is neither an identifier nor a list nor a vector, or one of the following.

`(`⟨syntactic datum⟩ `...)`

`(`⟨syntactic datum⟩ `... .` ⟨syntactic datum⟩`)`

`#(`⟨syntactic datum⟩ `...)`

Semantics: `quote-syntax` is the syntactic analogue of `quote`. It evaluates to a syntax object representation of the ⟨syntactic datum⟩ which retains hygiene information for the identifiers contained in the ⟨syntactic datum⟩. The result of evaluating a `quote-syntax` expression is suitable for inclusion in the expansion of a macro use.

Examples:

```
(symbol? (quote-syntax x))
```

```
⇒ #f

(identifier? (unwrap-syntax (quote-syntax x)))
⇒ #t

(let-syntax ((car (lambda (x) (quote-syntax car))))
  ((car) '(0)))
```

```
⇒ 0

(let-syntax
    ((quote-foo
      (lambda (stx)
        (quote-syntax (quote foo)))))
  (let ((quote (lambda (x) 'bar)))
    (quote-foo)))
```

```
⇒ foo
```

Note: This form was called `syntax` in the R4RS. In the R6RS, the form called `syntax` was extended with additional functionality; that is also the version which appears under that name in this report (section 4.3).

`(unwrap-syntax` *stx*`)` procedure

Unwraps the immediate datum structure from the syntax object *stx*, leaving nested syntax structure (if any) in place, without stripping any syntactic information from identifiers.

Operationally, if *stx* is an identifier or is not a wrapped syntax object, then it is returned unchanged. Otherwise `unwrap-syntax` converts the outermost structure of *stx* into a data object, returning a pair whose car and cdr are syntax objects, a vector whose elements are syntax objects, or a Scheme value which is neither an identifier, a pair, nor a vector. Syntax objects within the pairs or vectors returned by `unwrap-syntax` retain their original hygiene information.

Examples:

```
(identifier? (unwrap-syntax (quote-syntax x)))
⇒ #t

(identifier? (cdr (unwrap-syntax (quote-syntax (x . y)))))
⇒ #t

(identifier? (cdr (unwrap-syntax (quote-syntax (x y z)))))
⇒ #f
```

`(syntax->datum` *stx*`)` procedure

Strips all syntactic information from the syntax object *stx* and returns the corresponding Scheme datum. Identifiers stripped in this manner are converted to their symbolic names.

The result of `syntax->datum` must not be and must not contain any wrapped syntax objects. If a datum wrapped within *stx* contains cycles, these must be present (re-created in a copy, if necessary) within the returned datum.

Rationale: By processing the result of calling `syntax->datum` on a syntax object in parallel with unwrapping the original syntax object step by step, transformers which need to handle cyclical structures specially can detect and process such structures as appropriate.

This procedure irrevocably deletes hygiene information from identifiers: `syntax->datum` and `datum->syntax` cannot, in general, round-trip cleanly.

Examples:

```
(symbol? (syntax->datum #'x))
⇒ #t

(syntax->datum (quote-syntax (quote #1=(a . #1#))))
⇒ (quote #1=(a . #1#))
```

Note: This procedure, which can operate on entire expressions and not just individual identifiers, replaces the procedure `identifier->symbol` of the R4RS.

`(datum->syntax` *context-id datum*`)` procedure

context-id must be an identifier and *datum* should be a datum value.

Returns a syntax object representation of *datum* that contains the same contextual information as *context-id*, with the effect that the syntax object behaves as if it were introduced into the code when *context-id* was introduced.

Operationally, `datum->syntax` creates a new wrapped syntax object which wraps *datum* and which copies its hygiene information from *context-id*.

Note: This procedure, which can operate on entire expressions and not just individual symbols, replaces the procedure `construct-identifier` of the R4RS.

> *Todo:* What if *datum* already contains wrapped syntax objects? Should they be unchanged in the output?

Example: The following macro makes an early return procedure available in its body under the name `return`, without this name having to be explicitly given to the macro.

```
(define-syntax with-return
  (lambda (stx)
    (let ((return-id
           (syntax->datum (car (unwrap-syntax stx)) 'return))
          (body (cdr (unwrap-syntax stx))))
      `(,(quote-syntax call-with-current-continuation)
        (,(quote-syntax lambda) (,return-id) . ,body)))))
```

```
(define (find-odd ls)
  (with-return
    (for-each
     (lambda (n) (if (odd? n) (return n)))
     ls)))
```

```
(find-odd '(6 2 8 3 1 8))
⇒ 3
```

The following example shows how the name `return` is introduced into the code at the same time the keyword `with-return` in the macro use was introduced: it is available within the code introduced by the expansion of a hygienic `syntax-rules` macro, and not to the user of that macro, preserving the referential transparency of hygienic macros which make use of non-hygienic macros in their implementation.

```
(define-syntax suppress-exceptions
  (syntax-rules ()
    ((_ body_0 body_1 ...)
     (with-return
       (with-exception-handler
           (lambda (e) (return #f))
         (lambda () body_0 body_1 ...))))))
```

```
(suppress-exceptions (raise 'oops))
⇒ #f

(let ((return (lambda (ignored) #t)))
  (suppress-exceptions
    (return #f)))
⇒ #t
```

Todo: This example will probably need changing to use delimited control operators, once it is decided what form those will take in the Foundations.

THE MACROLOGICAL FASCICLE

CHAPTER 4

The syntax-case system

The syntax-case system provides support for writing low-level macros in a high-level style.

4.1. Pattern variables

Pattern variables are the unifying concept of both the syntax-case system and the closely related syntax-rules system, which is defined in section 5. They provide support for accessing the terminal symbols of a basic parser which operates on Scheme forms.

Pattern variables are a type of binding exactly like variables and syntax keywords. They occupy the same namespace as variables and syntax keywords and can shadow, and be shadowed by, bindings of them; the same name cannot refer to both a pattern variable and another type of binding within the same scope. The value of pattern variables cannot be changed after they have been bound.

Unlike normal variables, pattern variables can be bound to a sequence of multiple values, or any nesting of sequences of multiple values. The number of levels of nesting is determined statically by the pattern which names the pattern variable for binding. When the values are actually assigned to such a pattern variable at run time, each sequence may ultimately be empty or contain only one value.

4.2. Parsing input

The centrepiece of the syntax-case macro system is the eponymous pattern-based parser, the fundamental form for parsing macro uses, and the syntax form, the fundamental form for constructing syntax objects. Syntax-case binds pattern variables after parsing a form, and syntax is used to access their values.

(syntax-case ⟨expression⟩ (⟨pattern literal⟩ . . .) syntax
⟨syntax-case clause⟩ . . .)

(`syntax-case` ⟨custom ellipsis clause⟩ syntax
⟨expression⟩ (⟨pattern literal⟩ ...)
⟨syntax-case clause⟩ ...)

_ auxiliary syntax

... auxiliary syntax

Syntax: Each ⟨pattern literal⟩ must be an identifier. Each ⟨syntax-case clause⟩ must take one of the following two forms:

(⟨pattern⟩ ⟨output expression⟩)

(⟨pattern⟩ ⟨fender⟩ ⟨output expression⟩)

⟨Fender⟩ and ⟨output expression⟩ must be expressions.

A ⟨pattern⟩ is an identifier, a constant, or one of the following.

(⟨pattern⟩ ...)

(⟨pattern⟩ ⟨pattern⟩ ... . ⟨pattern⟩)

(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ...)

(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ... . ⟨pattern⟩)

#(⟨pattern⟩ ...)

#(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ ...)

⟨Custom ellipsis clause⟩, if present, is an instance of `custom-ellipsis` (section 4.5); ⟨ellipsis⟩ within a ⟨pattern⟩ refers to the auxiliary syntax keyword ... unless overridden by such a clause.

Semantics: A `syntax-case` expression first evaluates ⟨expression⟩ to obtain a syntax object. This input syntax object is matched against the ⟨pattern⟩s contained in the ⟨syntax-case clause⟩s from left to right.

An identifier appearing within a ⟨pattern⟩ can be an underscore (_), a literal identifier listed in the list of ⟨pattern literal⟩s, or the ⟨ellipsis⟩. All other identifiers appearing within a ⟨pattern⟩ are pattern variables.

Pattern variables match arbitrary input elements and are used to refer to elements of the input in the template. It is a syntax violation if the same pattern variable (in the sense of `bound-identifier=?`) appears more than once in a ⟨pattern⟩.

Underscores also match arbitrary input elements but are not pattern variables and so cannot be used to refer to those elements. If an underscore appears in the ⟨pattern literal⟩s list, then that takes precedence and underscores in the ⟨pattern⟩s match as literals. Multiple underscores can appear in a ⟨pattern⟩.

Identifiers that appear in `(`⟨pattern literal⟩ `...)` are interpreted as literal identifiers to be matched against corresponding elements of the input. An identifier within a ⟨pattern⟩ is treated as a literal identifier if and only if it is `bound-identifier=?` to an identifier within `(`⟨pattern literal⟩ `...)`. An element in the input matches a literal identifier in the pattern if and only if the two identifiers are the same in the sense of `free-identifier=?`.

A subpattern followed by ⟨ellipsis⟩ can match zero or more elements of the input, unless ⟨ellipsis⟩ appears in the ⟨pattern literal⟩s, in which case it is matched as a literal.

More formally, an input expression E matches a pattern P if and only if:

- P is an underscore (_); or
- P is a non-literal identifier; or
- P is a literal identifier and E is `free-identifier=?` to it; or
- P is a list $(P_1 \ \ldots \ P_n)$ and E is a list of n elements that match P_1 through P_n respectively; or

- P is an improper list $(P_1\ P_2\ \ldots\ P_n\ .\ P_{n+1})$ and E is a list or improper list of n or more elements that match P_1 through P_n, respectively, and whose nth tail matches P_{n+1}; or

- P is of the form $(P_1\ \ldots\ P_k\ P_e\ \langle\text{ellipsis}\rangle\ P_{m+1}\ \ldots\ P_n)$ where E is a proper list of n elements, the first k of which match P_1 through P_k, respectively, whose next $m-k$ elements each match P_e, whose remaining $n-m$ elements match P_{m+1} through P_n; or

- P is of the form $(P_1\ \ldots\ P_k\ P_e\ \langle\text{ellipsis}\rangle\ P_{m+1}\ \ldots\ P_n\ .\ P_x)$ where E is a list or improper list of n elements, the first k of which match P_1 through P_k, whose next $m-k$ elements each match P_e, whose remaining $n-m$ elements match P_{m+1} through P_n, and whose nth and final cdr matches P_x; or

- P is a vector of the form $\#(P_1\ \ldots\ P_n)$ and E is a vector of n elements that match P_1 through P_n; or

- P is of the form $\#(P_1\ \ldots\ P_k\ P_e\ \langle\text{ellipsis}\rangle\ P_{m+1}\ \ldots\ P_n)$ where E is a vector of n elements the first k of which match P_1 through P_k, whose next $m-k$ elements each match P_e, and whose remaining $n-m$ elements match P_{m+1} through P_n; or

- P is a constant and E is equal to P in the sense of the `equal?` procedure.

When the ⟨pattern⟩ of a given ⟨syntax-case clause⟩ matches the input syntax object, and the ⟨syntax-case clause⟩ contains a ⟨fender⟩ expression, the expression is evaluated to act as an additional constraint on acceptance of a clause. If the result of the evaluation is `#f`, the clause as a whole does not match, and pattern matching resumes on the next clause to the right. It is a syntax violation if the input syntax object does not match any of the clauses.

If the ⟨pattern⟩ of the clause matches and there is no ⟨fender⟩ expression, or the evaluation of the ⟨fender⟩ expression returned a true value, the ⟨output expression⟩ is evaluated and its value returned as the value of the `syntax-case` expression. If the `syntax-case` form is in tail context, each ⟨output expression⟩ is also in tail position.

Pattern variables contained within a clause's ⟨pattern⟩ are bound within the clause's ⟨fender⟩ (if present) and ⟨output expression⟩ to the corresponding pieces of the input form which they matched. Pattern variables contained within subpatterns followed by ⟨ellipsis⟩ are marked as holding sequences of multiple values according to the numbers of levels of nested levels of such subpatterns they are within; the results of destructuring those the input form according to the pattern become the values of those pattern variables.

Note: R6RS made any attempt to use the ellipsis or underscore as literals a syntax violation, and did not provide any means of renaming the ellipsis.

4.3. Generating expansions

`(syntax` ⟨template⟩`)` syntax

`(syntax` ⟨custom ellipsis clause⟩ ⟨template⟩`)` syntax

`#'`⟨template⟩ syntax

`...` auxiliary syntax

Syntax: `(syntax` ⟨template⟩`)` can be abbreviated as `#'`⟨template⟩. The two notations are equivalent in all respects.

A ⟨template⟩ is an identifier, a pattern datum, or one of the following.

`(`⟨subtemplate⟩ `...)`

`(`⟨subtemplate⟩ `... .` ⟨template⟩`)`

`#(`⟨subtemplate⟩ `...)`

`(`⟨ellipsis⟩ ⟨template⟩`)`

A ⟨subtemplate⟩ is a ⟨template⟩ followed by zero or more instances of ⟨ellipsis⟩.

⟨Custom ellipsis clause⟩, if present, is an instance of `custom-ellipsis` (section 4.5); ⟨ellipsis⟩ within a ⟨template⟩ refers to the auxiliary syntax keyword `...` unless overridden by such a clause.

It is a syntax violation if the ⟨template⟩ contains circular references.

Semantics: A `syntax` expression is similar to a `quote-syntax` expression, except that the values of pattern variables appearing within ⟨template⟩ are inserted into the ⟨template⟩ by copying the template, and the result of evaluating a `syntax` expression is a syntax object which is only partially wrapped, as described below.

A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. Pattern variables that occur in subpatterns followed by one or more ellipses may occur only in subtemplates that are followed by (at least) as many ellipses. These pattern variables are replaced in the output by the input subforms to which they are bound, distributed as specified. If a pattern variable is followed by more ellipses in the subtemplate than in the associated subpattern, the input form is replicated for the outermost excess ellipses as necessary. [*Editorial note:* Can the meaning of 'replicated for the outermost excess ellipses' be made clearer?] The subtemplate must contain at least one pattern variable from a subpattern followed by an ellipsis, and for at least one such pattern variable, the subtemplate must be followed by exactly as many ellipses as the subpattern in which the pattern variable appears; otherwise, it is a syntax violation.

A template of the form (⟨ellipsis⟩ ⟨template⟩) is equivalent to ⟨template⟩, except that the effect of the ellipsis within the template is suppressed and it is treated like any other ordinary identifier. In particular, the template (⟨ellipsis⟩ ⟨ellipsis⟩) produces a single ellipsis. This allows macro uses to expand into forms containing ellipses.

The result of evaluating a `syntax` expression is a copy of the ⟨template⟩ which is wrapped or unwrapped according to the following rules.

- The copy of a template which is a proper or improper list consists of unwrapped pairs as far as the rightmost subtemplate which contains a pattern variable. The cars of the pairs in the copy of the list are wrapped if they would be wrapped by applying these rules to the cars in the subtemplates recursively. If the last subtemplate in a proper list contains a pattern variable, then all pairs which form part of the list and the empty list in the final cdr are unwrapped. If the template is an improper list and the final cdr is a pattern variable, then all pairs which form part of the improper list are un-

wrapped and the final cdr is replaced by the value of the pattern variable in the copy.

- The copy of a template which is a vector is unwrapped if any of its subtemplates contains at least one pattern variable.
- The copy of any other template may be wrapped.

The values of the pattern variables are not copied when substituted into the template, and are thus wrapped or unwrapped to the same degree as when they were bound. Other datums and identifiers that are not pattern variables or ellipses are copied directly into the output, maintaining the contextual information associated with them.

`(quasisyntax` ⟨quasi-template⟩`)`	syntax
`#`` ⟨quasi-template⟩	syntax
`(quasisyntax` ⟨custom ellipsis clause⟩ ⟨quasi-template⟩`)`	syntax
`#`` ⟨quasi-template⟩	syntax
`(unsyntax` ⟨expression⟩ `...)`	auxiliary syntax
`#,` ⟨expression⟩	auxiliary syntax
`(unsyntax-splicing` ⟨expression⟩ `...)`	auxiliary syntax
`#,@` ⟨expression⟩	auxiliary syntax
`...`	auxiliary syntax

Syntax: `(quasisyntax` ⟨quasi-template⟩`)` can be abbreviated as `#`` ⟨quasi-template⟩, `(unsyntax` ⟨expression⟩`)` as `#,` ⟨expression⟩, and `(unsyntax-splicing` ⟨expression⟩`)` as `#,@` ⟨expression⟩. The notations are equivalent in all respects.

A ⟨quasi-template⟩ is either a ⟨template⟩, an instance of `quasisyntax`, `unsyntax`, or `unsyntax-splicing`, or a list or vector containing further ⟨quasi-template⟩s. Uses of `unsyntax` and `unsyntax-splicing` are valid only within ⟨quasi-template⟩s.

⟨Custom ellipsis clause⟩, if present, is an instance of `custom-ellipsis` (section 4.5); ⟨ellipsis⟩ within a ⟨template⟩ refers to the auxiliary syntax keyword

. . . unless overridden by such a clause.

The behaviour is undefined if the ⟨quasi-template⟩ contains circular references outside of a context within an ⟨expression⟩ where they are allowed.

Semantics: The `quasisyntax` form is similar to `syntax`, but it allows parts of its template to be evaluated, in a manner similar to the operation of `quasiquote`. `Unsyntax` and `unsyntax-splicing` are the `quasisyntax` analogues of `unquote` and `unquote-splicing`.

Rationale: While `unquote` and `unquote-splicing` could be re-used in `quasisyntax` for the purpose of escaping out of the quoted environment, that would make generating macro output including a `quasiquote` expression unnecessarily tricky.

Within the ⟨quasi-template⟩, the ⟨expression⟩s of `unsyntax` and `unsyntax-splicing` forms are evaluated; everything else is treated as ordinary template material, as with `syntax`. The value of each `unsyntax` subform is inserted into the output in place of the unsyntax form, while the value of each `unsyntax-splicing` subform is spliced into the surrounding list or vector structure.

A `quasisyntax` expression may be nested, with each `quasisyntax` introducing a new level of syntax quotation and each `unsyntax` or `unsyntax-splicing` taking away a level of syntax quotation. An expression nested within *n* `quasisyntax` expressions must be within *n* `unsyntax` or `unsyntax-splicing` expressions to be evaluated.

All uses of `unsyntax-splicing`, and uses of `unsyntax` or `unsyntax-splicing` with zero or more than one subform, are valid only within lists or vectors. Each use of `unsyntax` or `unsyntax-splicing` with zero subforms results in no elements being inserted into the list or vector: the `unsyntax` or `unsyntax-splicing` is treated as if it were not there. Each use of `unsyntax` or `unsyntax-splicing` with more than one subform is equivalent to the same number of individual `unsyntax` or `unsyntax-splicing` forms, each with one of the subforms, in the same order.

Rationale: Uses of `unsyntax` and `unsyntax-splicing` with zero or more than one subform enable certain idioms, such as `#,@#,@`. This has the effect of a doubly indirect splicing when

used within a doubly nested and doubly evaluated `quasisyntax` expression.

4.4. Binding other pattern variables within procedural macros

`(with-syntax ((`⟨pattern⟩ ⟨expression⟩`) ...)` syntax
⟨body⟩`)`

`(with-syntax` ⟨custom ellipsis clause⟩ syntax
`((`⟨pattern⟩ ⟨expression⟩`) ...)`
⟨body⟩`)`

The `with-syntax` form is the fundamental pattern variable binding form.

Syntax: Each ⟨pattern⟩ is identical in form to a `syntax-case` pattern.

⟨Custom ellipsis clause⟩, if present, is an instance of `custom-ellipsis` (section 4.5); ⟨ellipsis⟩ within a ⟨pattern⟩ refers to the auxiliary syntax keyword `...` unless overridden by such a clause.

Semantics: The value of each ⟨expression⟩ is computed and destructured according to the corresponding ⟨pattern⟩, and pattern variables within the ⟨pattern⟩ are bound as if by `syntax-case` to the corresponding portions of the value within ⟨body⟩. It is a syntax violation if the result of evaluating an ⟨expression⟩ does not match the corresponding ⟨pattern⟩.

Implementation:

```
(define-syntax with-syntax
  (lambda (stx)
    (syntax-case stx ()
      ((_ ((pattern expression) ...) body_0 body_1 ...)
       #'(syntax-case (list expression ...) ()
           ((pattern ...) (let () body_0 body_1 ...)))))))
```

4.5. Writing macros which generate other macros

`(custom-ellipsis ⟨custom ellipsis⟩)` auxiliary syntax

Syntax: ⟨Custom ellipsis⟩ must be an identifier.

Semantics: When a `custom-ellipsis` form is the first subform of a `syntax-case`, `syntax`, `quasisyntax`, or `with-syntax` form, instances of ⟨ellipsis⟩ within the syntax of the ⟨pattern⟩, ⟨template⟩, or ⟨quasi-template⟩ of the respective form refer not to the auxiliary syntax keyword `...`, but to any identifier which is `bound-identifier=?` to the ⟨custom ellipsis⟩ identifier.

4.6. Examples

Many simpler macros can be written using `syntax-rules` (see section 5) and trivially converted into `syntax-case`. This is useful, for example, when changing code by using `syntax-case` to add additional functionality or error checking to a macro whose original definition was in `syntax-rules`. The following example shows how the `swap!` example of `syntax-rules` (section 5) can first be rewritten to use `syntax-case`.

```
(define-syntax swap!
  (syntax-rules ()
    ((_ a b)
     (let ((temp a))
       (set! a b)
       (set! b temp)))))
```

```
≡ (define-syntax swap!
    (lambda (stx)
      (syntax-case stx ()
        ((_ a b)
         #'(let ((temp a))
             (set! a b)
             (set! b temp))))))
```

The definition can then be improved using a fender clause to improve error reporting in the case that either of the arguments to `swap!` is not an identifier. With the above definition, `(swap! (car x) (car y))` would result in a syntax violation being signalled which claims that `set!` had been used incorrectly, even though there is no `set!` explicitly used in the code.

```
(define-syntax swap!
  (lambda (stx)
    (syntax-case stx ()
      ((_ a b)
       (and (identifier? #'a)
            (identifier? #'b))
       #'(let ((temp a))
           (set! a b)
           (set! b temp))))))
```

With this definition, the syntax violation signalled by `(swap! (car x) (car y))` will correctly report that `swap!` was used incorrectly.

The following example also shows how `syntax-case` can be used to improve error reporting from macros by writing explicit error checking code. It defines a variant of `case` which checks that all datums in a clause belong to types that can portably be used in `case`: that is, their behaviour under `eqv?` never depends on their location in the store, which for other types is dependent on the Scheme implementation. This kind of error checking is not possible in `syntax-rules`, which cannot in general detect the type of any subform as a datum. (This version of `case` also does not provide an `else` clause, instead signalling an error if no specific clause matches.)

```
(define-syntax my-case
  (let ((eqv-undefined?
         (lambda (x-stx)
           (let ((x (syntax->datum x-stx)))
             (not (or (boolean? x) (symbol? x) (number? x)
```

```
                    (char? x) (null? x)))))))
    (lambda (stx)
      (syntax-case stx ()
        ((_ key ((datum ...) expr_0 expr_1 ...) ...)
         (cond ((find eqv-undefined? #'(datum ... ...))
                => (lambda (bad-datum)
                     (syntax-violation
                      'my-case
                      "use of datum in my-case is not portable"
                      stx bad-datum)))
               (else
                #'(case key
                    ((datum ...) expr_0 expr_1 ...) ...
                    (else
                     (error "key did not match any my-case datum"
                            key)))))))))
```

Macros written using `syntax-case` can also bind an implicit identifier, which cannot be done with `syntax-rules`. The `with-return` example from section 3.2 can be reformulated in terms of `syntax-case` as follows. The two definitions are equivalent except that the first one uses `quasisyntax` and the second `with-syntax`.

```
(define-syntax with-return
  (syntax-case stx ()
    ((k body_0 body_1 ...)
     (let ((return-id (datum->syntax #'k 'return)))
       #`(call-with-current-continuation
          (lambda (#,return-id)
            body_0 body_1 ...))))))
```

```
(define-syntax with-return
  (syntax-case stx ()
```

```
    ((k body_0 body_1 ...)
     (with-syntax ((return (datum->syntax #'k 'return)))
       #'(call-with-current-continuation
          (lambda (return)
            body_0 body_1 ...))))))
```

Syntax-case can also be used in the definition of identifier macros. The used-as example from section 2.4 can be reformulated in terms of syntax-case as follows.

```
(define-syntax used-as
  (make-variable-transformer
   (lambda (stx)
     (syntax-case stx (set!)
       (id
        (identifier? #'id)
        #'(quote reference))
       ((set! _ value)
        #'(quote (assignment value)))
       ((_ . operands)
        #'(quote (combination . operands)))))))
```

Identifier macros written using syntax-case can be used to optimize expensive procedure calls at expand time, while still providing the functionality of a first-class procedure. The following wrapper around concatenate turns uses into the more efficient append-map when its argument is known to be a call to the map procedure.

```
(define-syntax fast-concatenate
  (lambda (stx)
    (syntax-case stx (map)
      ((_ (map f ls_0 ls_1 ...))
       #'(append-map f ls_0 ls_1 ...))
      ((_ ls)
```

```
        #'(concatenate ls))
       (id
        (identifier? #'id)
        #'concatenate))))

(fast-concatenate (map make-list '(1 2 3) '(a b c)))
⇒ (a b b c c c)

(fast-concatenate (list '(bh b p) '(dh d t)))
⇒ (bh b p dh d t)

(apply fast-concatenate '(((gh g k) (g*h g* k*))))
⇒ (gh g k g*h g* k*)
```

Users should note, however, that many implementations of Scheme include sophisticated compilers which are able to recognize procedure calls which can be safely evaluated before run time, and which can usually optimize such cases more effectively than any macro definition. Explicit use of macros like this should usually be limited to instances where optimization cannot be done by a compiler. This typically includes cases in which the procedure uses side effects within its definition, or (as in the above example) where an optimization is possible when some information about arguments' values is known at expand time, but the values are otherwise not known until run time. Note also that the above example does not prevent the compiler from later additionally performing this optimization on the resulting `append-map` call when all its arguments are known at compile time.

THE MACROLOGICAL FASCICLE

CHAPTER 5

The `syntax-rules` system

The `syntax-rules` system can be used to write the simplest macros somewhat more concisely than is possible in the `syntax-case` system, with which it is closely related. A macro implemented in the `syntax-rules` system cannot perform arbitrary Scheme evaluation during its expansion. It also has no means of identifier capture, though this can sometimes be simulated using syntax parameters (see section 2.3).

`(syntax-rules (`⟨pattern literal⟩ `...)` syntax
⟨syntax rule⟩ `...)`

`(syntax-rules` ⟨custom ellipsis⟩ `(`⟨pattern literal⟩ `...)` syntax
⟨syntax rule⟩ `...)`

`_` auxiliary syntax

`...` auxiliary syntax

Syntax: Each ⟨pattern literal⟩ must be an identifier. If a ⟨custom ellipsis⟩ is provided, it must be an identifier. Each ⟨syntax rule⟩ has the form

(⟨rule pattern⟩ ⟨template⟩)

A ⟨template⟩ has the same form as in the definition of `syntax`. A ⟨rule pattern⟩ must have one of the following four forms:

(⟨identifier⟩ ⟨pattern⟩ `...`)

(⟨identifier⟩ ⟨pattern⟩ `...` `.` ⟨pattern⟩)

(⟨identifier⟩ ⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...`)

(⟨identifier⟩ ⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...` `.` ⟨pattern⟩)

⟨Pattern⟩ has the same form as in the definition of `syntax-case`.

⟨Custom ellipsis⟩, if provided, must be an identifier. Within a ⟨rule pattern⟩, ⟨pattern⟩, and ⟨template⟩, ⟨ellipsis⟩ refers to an identifier which is `bound-identifier=?` to this custom ellipsis identifier, if it is provided, or to the auxiliary syntax keyword `...` otherwise.

Semantics: An instance of `syntax-rules` evaluates to a transformer procedure which operates according to a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by `syntax-rules` is matched against the patterns contained in the ⟨syntax rule⟩s, beginning with the leftmost ⟨syntax rule⟩. When a match is found, the macro use is transcribed hygienically according to the template. It is a syntax violation when no match is found.

The ⟨identifier⟩ at the beginning of a ⟨rule pattern⟩ is not involved in the matching and is considered neither a pattern variable nor a literal identifier. Thus, ⟨rule pattern⟩s are like `syntax-case` patterns, but are restricted to matching uses of macros which are not identifier macros.

Excluding the initial identifier, which is treated as if it were the auxiliary syntax keyword `_`, a macro use defined using `syntax-rules` is transcribed according to the template of the matching ⟨syntax rule⟩ as if the pattern had been matched using `syntax-case` with the same given list of ⟨pattern literal⟩s, and a `syntax` expression containing the ⟨template⟩ were the only content of that `syntax-case` clause's output expression.

Examples:

The following macro destructively swaps the values associated with the identifiers it is given.

```
(define-syntax swap!
  (syntax-rules ()
    ((_ a b)
     (let ((temp a))
```

```
      (set! a b)
      (set! b temp)))))
```

Because hygiene is automatically maintained, the use of `temp` as an identifier internal to the expansion does not conflict with any existing binding called `temp` which exists in the place where the macro is used:

```
(define temp 37)
(define fever-temp 38)
(swap! fever-temp temp)
(values temp fever-temp)
```

⇒ 38 37

Further, local redefinitions or re-bindings of `let` or `set!` at the place the macro is used do not affect the meaning of the macro expansion:

```
(let-syntax
    ((let (erroneous-syntax "let is not allowed here")))
  (swap! x y))
```

⇒ swaps the values of `x` and `y` without raising an error

Because the identifiers introduced by each macro transcription step receive a unique time-stamp, a recursively-expanding `syntax-rules` macro can generate an arbitrary number of distinct identifiers. The following example uses this together with the guaranteed evaluation order of `let*` to define a macro which expands into a normal Scheme procedure call, but guarantees that the procedure and its operands will be evaluated in left-to-right order.

```
(define-syntax call*
  (syntax-rules ()
    ((_ args ...)
     (call*-aux (args ...) ()))))

(define-syntax call*-aux
```

```
  (syntax-rules ()
    ((_ (expr . more-exprs) (exprs-w/gen-ids ...))
     (call*-aux more-exprs (exprs-w/gen-ids ... (gen-id expr))))
    ((_ () ((gen-id expr) ...))
     (let* ((gen-id expr) ...)
       (gen-id ...)))))
```

If the body of the `let` expression in the following example were simply `(cons (read source) (read source))`, an implementation of Scheme would be allowed to return `((then this) . this-first)`. Using this `call*` macro guarantees the intended result.

```
(let ((source (open-input-string "this-first (then this)")))
  (call* cons (read source)
              (read source)))
```

⇒ `(this-first then this)`

Implementation:

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ ell (lit ...) ((k . p) t) ...)
       (and (identifier? #'ell)
            (every identifier? #'(lit ... k ...)))
       #'(lambda (x)
           (syntax-case (custom-ellipsis ell) x (lit ...)
             ((_ . p) (syntax (custom-ellipsis ell) t)) ...)))
      ((_ (lit ...) ((k . p) t) ...)
       (every identifier? #'(lit ... k ...))
       #'(lambda (x)
           (syntax-case x (lit ...)
             ((_ . p) #'t) ...))))))
```

Note: The `syntax-rules` of this report is a compatible extension to that of the small language, and the `(scheme base)` library must export the same binding [*Editorial note:* as whichever large language library this ends up in]. Compared to the small language version, the ability to match macro uses of the form (⟨keyword⟩ ⟨datum⟩ ... . ⟨datum⟩), to use multiple ellipses after a subtemplate, and to use ellipsized subtemplates which include pattern variables with mismatching levels of ellipsis nesting in the pattern have been added.

Compared to the R6RS version of `syntax-rules`, the ability to rename the ellipsis and to use the ellipsis and underscore as literals have been added.

Compared to the R5RS version of `syntax-rules`, the pattern language has been extended to allow further patterns after an ⟨ellipsis⟩, and the special identifier _ to match any input form without creating a pattern variable and the ability to rename the ellipsis have been added. As in the small language version, the R5RS version also did not allow macro uses of the form (⟨keyword⟩ ⟨datum⟩ ... . ⟨datum⟩), nor multiple ellipses after a subtemplate, nor ellipsized subtemplates with pattern variables at different levels of nesting in the pattern.

`(identifier-syntax` ⟨template⟩`)` syntax

`(identifier-syntax` syntax

`(`⟨identifier$_1$⟩ ⟨template$_1$⟩`)`

`((set!` ⟨identifier$_2$⟩ ⟨pattern⟩`)` ⟨template$_2$⟩`))`

`set!` auxiliary syntax

`Identifier-syntax` is a purely template-based form for creating transformers for identifier macros, in the same way that `syntax-rules` is a purely template-based form for creating transformers for macros that are not identifier macros.

Syntax: The ⟨pattern⟩ must be as for `syntax-case`, and the ⟨template⟩s must be as for `syntax` and `syntax-rules`.

Note: The `set!` referred to here as auxiliary syntax has the same binding as the `set!` keyword used as non-auxiliary syntax.

Semantics: `Identifier-syntax` evaluates to a macro transformer.

In the first form of `identifier-syntax`, every instance of the syntax keyword bound to the returned transformer is replaced by the ⟨template⟩ within the scope

of the keyword. It is a syntax violation to use `set!` on syntax keywords associated with transformers created with this form of `identifier-syntax`.

The second, more general, form of `identifier-syntax` permits the transformer to determine what happens when `set!` is used. In this case, uses of the syntax keyword itself are replaced by ⟨template$_1$⟩, and uses of `set!` with the syntax keyword are replaced by ⟨template$_2$⟩.

Pattern variables within the ⟨template⟩s are substituted as in `syntax-rules`; for this purpose, the ⟨identifier⟩s and ⟨pattern⟩ are treated as patterns matched against the relevant parts of the macro uses.

Note: Use of ⟨pattern⟩s more complex than a single pattern variable in the `set!` clause of the second form of `identifier-syntax` is inadvisable because the resulting macros will likely surprise users when they cannot be used to `set!` the identifier to the value of another, existing variable.

Todo: Should the ⟨pattern⟩ be restricted to one identifier? [*Editorial note:* Issue 142.]

Todo: Should `identifier-syntax` support `custom-ellipsis` or otherwise ellipsis renaming?

Example: `Identifier-syntax` could be used to define constants which cannot be mutated anywhere.

```
(define-syntax define-constant
  (syntax-rules ()
    ((_ name value)
     (begin
       (define constant-value value)
       (define-syntax name (identifier-syntax constant-value))))))
```

Using `define-constant` in the manner of `define` creates an identifier binding which cannot be `set!` in any context.

```
(define-constant π 3.1415927)
(set! π #e3.2)
```

⇒ syntax violation

Conversely, another possible use of `identifier-syntax` is to create the illusion that a library exports a variable which can be mutated.

```
(library (magic-library)
  (export magic-variable)
  (import #;(todo))

  (define magic-variable-contents (cons 'initial-value '()))
  (define-syntax magic-variable
    (identifier-syntax
      (_ (car magic-variable-contents))
      ((set! _ val) (set-car! magic-variable-contents val)))))
```

Programs and libraries which import `(magic-library)` can seemingly see and set the value of an identifier called `magic-variable`. Its value is always the same in all contexts in which it is imported. In reality, the car of a pair is holding the value, rather than any binding directly.

Implementation:

```
(define-syntax identifier-syntax
  (lambda (x)
    (syntax-case x (set!)
      ((_ e)
       #'(lambda (x)
           (syntax-case x ()
             (id (identifier? #'id) #'e)
```

```
          ((_ x (... ...)) #'(e x (... ...))))))
    ((_ (id exp1) ((set! var val) exp2))
     (and (identifier? #'id) (identifier? #'var))
     #'(make-variable-transformer
        (lambda (x)
          (syntax-case x (set!)
            ((set! var val) #'exp2]
            ((id x (... ...)) #'(exp1 x (... ...)))
            (id (identifier? #'id) #'exp1)))))))
```

5.1. Indicating erroneous macro uses

`(syntax-error` ⟨message⟩ ⟨irritant⟩ `...)` syntax

Syntax: ⟨Message⟩ must be a string literal. The ⟨irritant⟩s may be any Scheme datums.

Semantics: Any attempt to expand a `syntax-error` form results in a syntax violation being signalled. The condition value raised is a compound condition with at least three components: a `&syntax-violation` condition whose *form* and *subform* fields are unspecified, but which should both be set to `#f` if no useful value can be provided; a `&message` condition whose field is set to the given ⟨message⟩; and an `&irritants` condition whose field is set to a list of the ⟨irritant⟩s as syntax objects. An implementation may also include additional components within the compound condition.

This can be used as a `syntax-rules` ⟨template⟩ for a ⟨rule pattern⟩ that is an invalid use of the macro, which can provide more descriptive error messages. An implementation of `syntax-rules` can recognize when an instance of `syntax-error` is the only content of a ⟨template⟩ and use this to provide more helpful information for the fields of the `&syntax-violation` condition component.

Example:

```
(define-syntax simple-let
  (syntax-rules ()
    ((_ ((x . y) val) body1 body2 ...)
     (syntax-error "expected an identifier" (x . y)))
    ((_ (name val) body1 body2 ...)
     ((lambda (name) body1 body2 ...) val))))
```

Implementation:

```
(define-syntax syntax-error
  (lambda (stx)
    (syntax-case stx ()
      ((_ message irritant ...)
       (string? (syntax->datum #'message))
       (raise
        (condition
         (make-syntax-violation #f #f)
         (make-message-condition (syntax->datum #'message))
         (make-irritants-condition #'(irritant ...)))))))))
```

`(erroneous-syntax)` syntax

`(erroneous-syntax` ⟨message⟩`)` syntax

Syntax: ⟨Message⟩, if given, must be a string literal.

Semantics: An instance of `erroneous-syntax` evaluates to a macro transformer which always signals a syntax violation when invoked. If a ⟨message⟩ is provided, it will become the value of the message field of the syntax violation; if no message is provided, a default message (which may vary depending on the exact form of the macro use) should indicate that the keyword cannot be used in this context.

This can be used to define auxiliary syntax keywords which can never be correct macro uses on their own, as well as syntax parameters with no meaningful default transformers, and keys for identifier properties.

Examples: An implementation of Scheme might contain this somewhere for use by the `cond` and `case` forms:

```
(define-syntax => (erroneous-syntax))
(define-syntax else (erroneous-syntax))
```

A keyword used only as a key for identifier properties might be defined thus:

```
(define-syntax documentation
  (erroneous-syntax "The documentation keyword is used only as an \
identifier property key"))
```

THE MACROLOGICAL FASCICLE

APPENDIX A

Other macro systems

This section of the report is non-normative and aims to demonstrate how macros written with some other Scheme macro systems can be accommodated by a macro system based on syntax objects.

A.1. Unhygienic macros

Traditional Lisp macros offer no protection against accidental identifier capture and do not ensure that free variables in the expansion have the same references as at the point where the macro was defined. The argument to this `lisp-transformer` form is a procedure which receives the macro use as a completely unwrapped datum.

Implementation:

```
(define (lisp-transformer transformer)
  (lambda (stx)
    (syntax-case stx ()
      ((use-ctx . rest)
       (datum->syntax #'use-ctx
                      (transformer
                       (syntax->datum stx)))))))
```

A.2. Explicit renaming macros

The explicit renaming system was introduced by Clinger (1991) as an alternative means of implementing `syntax-rules`. As a low-level macro system, its weaknesses are its relative verbosity when used to implement a fully hygienic macro, and its inability to control the context of identifier capture, meaning it cannot comply with the requirements on introducing capturing references in section 1.1. The version of the `er-macro-transformer` form defined here can avoid this prob-

lem by allowing selective use of the `datum->syntax` form to capture identifiers, instead of using symbols as in the original definition of `er-macro-transformer`.

A small number of explicit renaming macros may require adjustment to work correctly under this version of `er-macro-transformer`. Some adjustment may be required because typical explicit renaming macros use `eq?`, `eqv?`, or `symbol=?` to check whether binding one identifier would shadow another — in other words, to provide the functionality of `bound-identifier=?`. This use was not defined in Clinger's original paper and is a later addition to the vernacular of explicit renaming systems.

In the `syntax-case` system, two identifiers with this property are not typically the same in the sense of `eq?` or `eqv?`. If the macro implementation uses `symbol=?` to perform this check, a domain error will occur on any such attempt to compare identifiers; such macros already do not work on many implementations of explicit renaming, however, where identifiers are wrapped in syntactic closures (Bawden and Rees 1988, Hanson 1991). Likewise, using `symbol?` to check whether part of a macro's input expression is an identifier will not work in this implementation, but such macros were also already broken in explicit renaming systems implemented on top of syntactic closures. In fact, there is no way to do this which already works across different explicit renaming-based expanders.

As can be seen, this implementation's weaknesses mainly find their ultimate root cause in the under-specified nature of the explicit renaming system itself. Another weakness is that if a use of a macro defined with this implementation contains a circular literal value, the `unwrap` procedure will diverge.

Implementation:

```
(define (unwrap stx)
  (syntax-case stx ()
    ((a . b) (cons (unwrap #'a) (unwrap #'b)))
    (#(a ...) (vector-map unwrap #'#(a ...)))
    (id (identifier? #'id) #'id)
    (_ (syntax->datum stx))))
```

```
(define (rewrap ctx expr)
  (let rewrap* ((e e))
    (cond
     ((pair? e) (cons (rewrap* (car e)) (rewrap* (cdr e))))
     ((vector? e) (vector-map rewrap* e))
     ((identifier? e) e)
     (else (datum->syntax ctx e)))))

(define (make-compare ctx)
  (lambda (x y)
    (free-identifier=? (rewrap ctx x) (rewrap ctx y))))

(define (make-rename ctx)
  (lambda (x)
    (datum->syntax ctx x)))

(define-syntax er-macro-transformer
  (lambda (stx)
    (syntax-case stx ()
      ((k proc-expr)
       #'(let ((proc proc-expr))
           (lambda (stx)
             (syntax-case stx ()
               ((m . _)
                (rewrap #'m
                        (proc (unwrap stx)
                              (make-rename #'k)
                              (make-compare #'m)))))))))))
```

A.3. Implicit renaming macros

Implicit renaming macros are a variant of explicit renaming macros with one difference. Instead of providing a `rename` procedure for referring to or inserting identifiers hygienically, and taking bare symbols in the output as the names of identifiers to be captured in the context of the macro keyword, the situation is inverted: bare symbols in the output are renamed hygienically and an `inject` procedure is used to capture identifiers in the context of the keyword at the macro use site. As above, in this implementation `datum->syntax` can be used to more safely perform this capture in contexts other than that of the macro use keyword.

The same caveats around the assumption that symbols are used to represent identifiers mentioned above in the section about explicit renaming macros apply here. The `unwrap`, `rewrap`, `make-compare`, and `make-rename` procedures are the same as for the explicit renaming implementation.

Implementation:

```
(define-syntax ir-macro-transformer
  (lambda (stx)
    (syntax-case stx ()
      ((k proc-expr)
       #'(let ((proc proc-expr))
           (lambda (stx)
             (syntax-case stx ()
               ((m . _)
                (rewrap #'k
                        (proc (unwrap stx)
                              (make-rename #'m)
                              (make-compare #'m)))))))))))
```

THE MACROLOGICAL FASCICLE

Acknowledgements

The following people responded to the Yellow Ballot:

- Alex Shinn
- Amirouche Boubekki
- Artem Chernyak
- Arthur A. Gleckler
- Chris Vine
- Daphne Preston-Kendal
- Dimitris Vyzovitis
- Dmitry Moskowski
- Duy Nguyen
- Emmanuel Medernach
- Gabriel B. Sant'Anna
- Graham Watt
- Jaime Fournier
- Jani Juhani Sinervo
- Jeremy Steward
- John Cowan
- Justin Ethier
- Linas Vepstas
- 'Lulu'
- Marc Nieper-Wißkirchen
- Marc-André Bélanger
- Mark Hughes
- Martin Rodgers
- Nicholas Carlson
- Ondřej Majerech
- Ross Angle
- Roy Mu
- Sam Phillips
- Shiro Kawai

- Takashi Kato
- Taylan Kammer
- Tim Van den Langenbergh
- Vijay Marupudi
- Vincent Manis
- Vladimir Nikishkin
- Wolfgang Corcoran-Mathe

Syntax parameters were introduced by Barzilay, Culpepper and Flatt (2011).

Marc Nieper-Wißkirchen contributed the sample implementation of explicit renaming in terms of syntax objects.

The following sources were also used:

- The Revised^4, Revised^6, and Revised^7 (small language) Reports on Scheme
- The Guile manual
- The Racket Reference
- The Chez Scheme 9 User Guide

THE MACROLOGICAL FASCICLE

Barendregt, Henk P., 'Introduction to the Lambda Calculus', *Nieuw Archief Voor Wiskunde*, 4/2 (1984), 337–72

Barzilay, Eli, Culpepper, Ryan, and Flatt, Matthew, 'Keeping It Clean with Syntax Parameters', 2011 <http://www.schemeworkshop.org/2011/papers/Barzilay2011.pdf> [accessed 30 August 2023]

Bawden, Alan, and Rees, Jonathan, 'Syntactic Closures', in *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88 (New York, NY, USA, 1988), 86–95 <https://doi.org/10.1145/62678.62687>

Bradner, Scott O., 'Key words for use in RFCs to Indicate Requirement Levels', Request for Comments (1997) <https://www.rfc-editor.org/info/rfc2119>

Clinger, William, 'Hygienic Macros Through Explicit Renaming', *SIGPLAN Lisp Pointers*, 4/4 (1991), 25–28 <https://doi.org/10.1145/1317265.1317269>

Clinger, William D., and Wand, Mitchell, 'Hygienic Macro Technology', *Proceedings of the ACM on Programming Languages*, 4/HOPL (2020) <https://doi.org/10.1145/3386330>

Clinger, William, and Rees, Jonathan, 'Macros That Work', in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91 (New York, NY, USA, 1991), 155–62 <https://doi.org/10.1145/99583.99607>

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

Flatt, Matthew, 'Binding as Sets of Scopes', in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16 (New York, NY, USA, 2016), 705–17 <https://doi.org/10.1145/2837614.2837620>

Hanson, Chris, 'A Syntactic Closures Macro Facility', *SIGPLAN Lisp Pointers*, 4/4 (1991), 9–16 <https://doi.org/10.1145/1317265.1317267>

Kohlbecker, Eugene E., Jr., 'Syntactic Extensions in the Programming Language LISP' (unpublished PhD thesis, Indiana University, 1986)

Kohlbecker, Eugene E., Jr., Friedman, Daniel P., Felleisen, Matthias, and Duba, Bruce, 'Hygienic Macro Expansion', in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86 (New York, NY, USA, 1986), 151–61 <https://doi.org/10.1145/319838.319859>

van Tonder, André, 'R6RS Libraries and Macros', 2006 <http://www.het.brown.edu/people/andre/macros/index.html> [accessed 28 September 2009]