Title

Hygienic macros.

Author

André van Tonder

Status

This SRFI is currently in *final* status. Here is an explanation of each status that a SRFI can hold. To provide input on this SRFI, please send email to `srfi-72@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2005-06-05
- Draft: 2005-06-14--2005-04-14
- Revised: 2005-07-01
- Revised: 2005-07-06
- Revised: 2005-08-04
- Revised: 2005-08-26
- Draft extension: 2005-09-20
- Revised: 2005-09-14
- Final: 2005-09-21
- Revised to fix errata:
  - 2020-02-11 (Fix broken links, etc.)

Index

Abstract

This SRFI describes a procedural macro proposal for Scheme with the following features:

- **Improved hygiene:**

  We argue that conventional hygiene algorithms may lead to accidental variable capture errors in procedural macros. We propose an improved algorithm that avoids these problems.

- **Reflective tower:**

  We specify a reflective tower of arbitrary height, and propose a refinement of lexical scoping that takes into account the phase of use of an identifier in determining its meaning.

- **Syntax-case:**

  In the current proposal, the `syntax-case` form is expressible as a macro in terms of a simpler set of primitives and is specified as library syntax.

- **Procedural interface:**

  The primitive interface for manipulating compound syntax objects consists of procedures rather than special forms. In particular, the traditional abstractions `car`, `cdr`, `cons`, `...` can be used on syntactic data.

- **Fast hygiene algorithm:**

  The reference implementation documents a fast imperative hygiene algorithm that is eager and linear in expression size.

- **Capturing identifiers:**

  A primitive `make-capturing-identifier` is provided for intentional variable capture and for building expansion-time fluid binding constructs.

Introduction

We start with a simple example:

```
(define-syntax (swap! a b)
   (quasisyntax
     (let ((temp ,a))
       (set! ,a ,b)
       (set! ,b temp))))
```

A syntax object is here constructed using the `quasisyntax` form. Syntax provided as part of the input expression can be inserted in the result using `unquote` or `unquote-splicing`. Macros written in this way are hygienic and referentially transparent.

The following example shows that we can use the procedures `car`, `cdr`, `null?`, `...`, on syntax objects. It also illustrates the use of the predicate `literal-identifier=?` for identifying literals in the input expression.

```
(define-syntax (my-cond c . cs)
  (if (literal-identifier=? (car c) (syntax else))
      (quasisyntax (begin ,@(cdr c)))
      (if (null? cs)
          (quasisyntax (if ,(car c) (begin ,@(cdr c))))
          (quasisyntax (if ,(car c)
                           (begin ,@(cdr c))
                           (my-cond ,@cs))))))
```

In the current proposal, the `syntax-case` form is expressible as a macro in terms of a simpler set of primitives, and is specified as library syntax. The `my-cond` macro can then also be expressed as:

```
(define-syntax my-cond
  (lambda (form)
    (syntax-case form (else)
      ((_ (else e1 ...) c1 ...) (syntax (begin e1 ...)))
      ((_ (e0 e1 ...))          (syntax (if e0 (begin e1 ...))))
      ((_ (e0 e1 ...) c1 ...)   (syntax (if e0
                                            (begin e1 ...)
                                            (my-cond c1 ...)))))))
```

Improved hygiene

In previous hygienic Scheme macro systems, accidental variable capture can take place in procedural macros, as the following example illustrates:

```
(let-syntax ((main (lambda (form)

                     (define (make-swap x y)
                       (quasisyntax
                        (let ((t ,x))
                          (set! ,x ,y)
                          (set! ,y t))))

                     (quasisyntax
                      (let ((s 1)
                            (t 2))
                        ,(make-swap (syntax s) (syntax t))
                        (list s t))))))
  (main))
           ==> (1 2) with conventional hygiene algorithm
               (2 1) with the proposal of this SRFI
```

This happens because the conventional hygiene algorithm regards all identifiers with the same name introduced during the entire duration of a macro invocation as identical.

This property makes it difficult to procedurally construct code fragments with fixed meaning for use in a different part of the program, since the meaning may always be accidentally corrupted at the use site. This violates the idea of referential transparency and may lead to fragility in large macros or where helper procedures are shared by macros.

The problem can show up in seemingly straightforward macros, as the following example illustrates. The `let-in-order` form is a version of `let` with guaranteed left-to-right evaluation of bindings. Again, the conventional hygiene algorithm leads to accidental capture of instances of `t` introduced recursively:

```
(define-syntax let-in-order
  (lambda (form)
    (syntax-case form ()
      ((_ ((i e) ...) e0 e1 ...)
       (let f ((ies (syntax ((i e) ...)))
               (its (syntax ())))
         (syntax-case ies ()
```

```
             (()            (quasisyntax (let ,its e0 e1 ...)))
             (((i e) . ies) (quasisyntax
                             (let ((t e))
                               ,(f (syntax ies)
                                   (quasisyntax ((i t) ,@its))))))))))))

(let-in-order ((x 1)
               (y 2))
  (+ x y))                ==> 4 with conventional hygiene algorithm
                              3 with the proposal of this SRFI
```

These are exactly the kind of problems that hygiene was invented to solve, reappearing in a slightly different guise. To eliminate these problems, this SRFI proposes the following ***modified hygiene rule*** [2-4]:

> *A binding for an identifier can only capture a reference to another if both were present in the source or introduced during a single evaluation of a **syntax** or **quasisyntax** form, with the understanding that the evaluation of any nested, unquoted **syntax** or **quasisyntax** forms counts as part of the evaluation of an enclosing **quasisyntax**.*

In the first example, this rule guarantees that the two instances of `t` are distinct, since they occur in separate `quasisyntax` forms. In the second example, the instances of `t` are also distinct since they are introduced during different invocations of the `quasisyntax` form. With this proposal, the above macros are correct as they stand.

Reflective tower

A reflective tower consists of a sequence of disjoint environments, each determining a set of bindings in effect during a given phase of execution [11, 14].

In the following example, the second definition and the right hand side of the macro binding for `m` are evaluated in the expand-time environment, while the first definition and the result of expanding the `let-syntax` expression are evaluated in the runtime environment. These two environments constitute a two-level reflective tower.

```
(define x 1)

(begin-for-syntax (define x 2))

(let-syntax ((m (lambda (form)
                  (quasisyntax (list x ,x)))))
  (m))  ==> (1 2)
```

The second `x` in `(list x ,x)` is evaluated during expand-time, whereas the first `x` is evaluated during runtime. Each is looked up in the appropriate environment.

The necessity for the tower of environments can be understood as follows: In an interpreter, expand-time and runtime bindings have to be in memory simultaneously, since expansion and evaluation alternate. These bindings must be kept strictly separate to ensure the following property, of fundamental importance in a statically scoped language:

> *The meaning of a program must depend unambiguously on its textual representation.*

To see this, consider expanding and evaluating the above sequence of expressions one by one. If we were to allow the second definition to shadow the first in a unified environment, the answer would differ from the one obtained by running a precompiled version of the program, so that the meaning would be ambiguous.

The presence of the reflective tower, with the implied absence of inter-phase shadowing, is expressed by the following lexical scoping rule:

> *The meaning of an identifier is determined by lexically visible bindings in force during the phase of its use.*

This SRFI differs from comparable macro systems in uniformly applying this rule to both toplevel and local bindings. Compare the above example with the following:

```
(let ((x 1))
  (let-syntax ((m (lambda (form)
                    (let ((x 2))
                      (quasisyntax (list x ,x))))))
    (m)))  ==> (1 2)
```

Here the inner `let` establishes a binding in the expand-time environment. As before, this binding cannot shadow the outer binding, which exists in the runtime environment. Each `x` in `(list x ,x)` is looked up in the appropriate environment when evaluated.

The reflective tower can have an arbitrary number of levels. In the following example, in addition to the runtime and expand-time phases, the right hand side of the inner macro is evaluated in a meta-expand-time phase. There are therefore three environments in the reflective tower, a construction that can obviously be iterated to arbitrary level. By nesting `begin-for-syntax` commands, we may specify bindings at arbitrary phase. Each `x` in the expression `(list x ,x ,,x)` is used as a variable in a separate phase.

```
(define x 0)
(begin-for-syntax
  (define x 1)
  (begin-for-syntax
    (define x 2)))

(let-syntax ((foo (lambda (form)
                    (let-syntax ((bar (lambda (form)
                                        (quasisyntax
                                         (quasisyntax
                                          (list x ,x ,,x))))))
                      (bar)))))
    (foo))  ==> (0 1 2)
```

Escaping ellipses

We require the ellipsis in the template in `(syntax ...)` to be interpreted as an ordinary identifier, not an ellipsis literal. The following idiom can then be used to include ellipses in `syntax-case`-generated macros:

```
(let-syntax ((m (lambda (form)
                  (syntax-case form ()
                    ((_ x ...)
```

```
                    (with-syntax ((::: (syntax ...)))
                      (syntax
                       (let-syntax ((n (lambda (form)
                                         (syntax-case form ()
                                           ((_ x ... :::)
                                            (syntax `(x ... :::)))))))))
                         (n a b c d)))))))))
(m u v))
          ==> (a b c d)
```

Specification

The following primitive forms are provided:

define-syntax
let-syntax
letrec-syntax

identifier?
bound-identifier=?
free-identifier=?
literal-identifier=?

syntax
quasisyntax

datum->syntax-object
syntax-object->datum
make-capturing-identifier

begin-for-syntax
around-syntax

syntax-error

The following library forms are provided:

syntax-case
with-syntax
syntax-rules

Syntax objects:

A syntax object is a graph whose nodes are Scheme pairs or vectors and whose leaves are constants or identifiers. The following expressions evaluate to syntax objects:

```
'()
1
#f
'(1 2 3)
(cons (syntax x) (vector 1 2 3 (syntax y)))
(syntax (let ((x 1)) x))
(quasisyntax (let ((x 1)) ,(syntax x)))
```

Symbols may not appear in syntax objects:

```
'(let ((x 1)) x)  ==> not a syntax object
```

Reflective tower:

A reflective tower consists of a sequence of strictly disjoint environments, each determining a set of bindings in effect during a given phase of execution. By nesting `let-syntax` expressions in macro definitions, the number of phases, and therefore the height of the reflective tower, may be made arbitrarily large.

The environment at each level in the tower initially contains the standard bindings of the host Scheme, as well as the additional primitives described in this SRFI.

The meaning of an identifier is determined by lexically visible bindings during the phase of its use.

In the following expression, the occurrence of `(syntax x)` on the right hand side of `m` evaluates to an object during expand-time and is therefore not affected by the inner binding. The resulting identifier is used as a runtime variable in the expanded code, and therefore denotes the outer binding.

```
(let ((x 1))
  (let-syntax ((m (lambda (form)
                    (let ((x 2))
                      (syntax x)))))
    (m)))  ==> 1
```

In the following example, there are two instances of `x` introduced by the macro `n`. The first is used during expand-time, where it refers to the binding `2`, whereas the second is used at runtime, where it refers to the binding `1`.

```
(let ((x 1))
  (let-syntax ((m (lambda (form)
                    (let ((x 2))
                      (let-syntax ((n (lambda (form)
                                        (syntax
                                         (let ((y x))
                                           (quasisyntax (list x ,y)))))))
                        (n))))))
    (m))) ==> (1 2)
```

The primitive `begin-for-syntax`, described below, is provided for specifying computations at arbitrary reflective level.

A fundamental principle of block-structured languages is that the meaning of a program should depend only on its textual representation. In particular, implementations should respect the following principle:

> *One should obtain the same results whether one expands and evaluates a sequence of top-level forms one by one in an interpreter, or first expands and compiles the whole sequence and then evaluates it.*

Applying this principle to the following sequence then shows the necessity for maintaining a separate namespace, with separate lexical scoping, for each reflective level. In an interpreter,

the two bindings for `x` will be in memory at the same time. If we allowed one to shadow the other in a unified environment, the answer would differ from the one obtained by running a precompiled version.

```
(define x 1)

(begin-for-syntax (define x 2))

(let-syntax ((m (lambda (form)
                  (quasisyntax (list x ,x)))))
  (m))  ==> (1 2)
```

Order of expansion:

In a procedural macro system, the order of expansion is observable. We leave the order unspecified except for the following minimal requirements, chosen for their potential usefulness. Terms in brackets have the meanings defined in R5RS, section 7.1:

- Any `<expression>` encountered during expansion is expanded atomically.
- Any given expansion step is performed only once, including those occurring during expansion of the first `<expression>` in a `<body>`.
- Any `<sequence>` is expanded from left to right.
- Toplevel `<command or definition>`s in a `<program>` are expanded atomically from left to right.

syntax: `(define-syntax keyword exp)`
`(define-syntax (keyword . formals) exp1 exp ...)`

`Exp` is expanded and evaluated in the current top-level syntactic environment, and must evaluate to a procedure of type `syntax-object -> syntax-object`, also called a transformer. The top-level syntactic environment is extended by binding the identifier `keyword` to the resulting transformer.

The second variant is equivalent to

```
(define-syntax keyword
   (let ((transformer (lambda (dummy . formals) exp1 exp ...)))
     (lambda (form)
       (apply transformer form))))
```

syntax: `(let-syntax ((keyword exp) ...) exp* ...)`
`(letrec-syntax ((keyword exp) ...) exp* ...)`

We generalize R5RS, section (4.3.1), by allowing each expression `exp` to evaluate to an arbitrary transformer procedure.

We also impose the requirement that `let[rec]-syntax` behave as a splicing form rather than introducing a new local scope. For example:

```
(let ((x 1))
   (let-syntax ((foo (syntax-rules ())))
     (define x 2))
   x)                ==> 2
```

procedure: **(identifier?** obj)

Returns `#t` if `obj` is an identifier, `#f` otherwise. The identifier type is disjoint from other Scheme primitive types described in R5RS, section (3.2).

procedure: **(bound-identifier=?** obj1 obj2)
(free-identifier=? obj1 obj2)
(literal-identifier=? obj1 obj2)

Identifiers are `free-identifier=?` if they would refer to the same lexical or toplevel binding if inserted as free identifiers in the result of the macro expansion. For this purpose, all identifiers that are not lexically bound are considered implicitly bound at the toplevel.

Identifiers are `literal-identifier=?` if they are `free-identifier=?` or if they both refer to toplevel bindings and have the same symbolic name. This primitive should be used to reliably identify literals (such as `else` in `cond`) even if they occur in a different module from the macro definition.

Identifiers are `bound-identifier=?` if a binding of one would capture references to the other in the scope of the binding.

Two identifiers with the same name are `bound-identifier=?` if both were present in the same toplevel expression in the original program text. Two identifiers will also be `bound-identifier=?` if they were produced from existing `bound-identifier=?` identifiers during a single evaluation of the same `syntax` or `quasisyntax` form, with the understanding that the evaluation of any nested, unquoted `syntax` or `quasisyntax` forms counts as part of the evaluation of an enclosing `quasisyntax`. In addition, `datum->syntax-object` may create identifiers that are `bound-identifier=?` to previously introduced identifiers.

These procedures return #f if either argument is not an identifier.

```
(free-identifier=?  (syntax x) (syntax x))       ==> #t
(bound-identifier=? (syntax x) (syntax x))       ==> #f

(let ((y (syntax (x . x))))
  (bound-identifier=? (car y)
                      (cdr y)))                   ==> #t

(quasisyntax ,(bound-identifier=? (syntax x)
                                  (syntax x)))   ==> #t

(let ((x 1))
  (let-syntax ((m (lambda (form)
                    (quasisyntax
                     (let ((x 2))
                       (let-syntax ((n (lambda (form)
                                         (free-identifier=? (cadr form)
                                                            (syntax x)))))
                         (n ,(cadr form))))))))
    (m x)))   ==> #f
```

syntax: (**syntax** datum)

Creates a new syntax object from `datum`, which must be a syntax object embedded in the input form, as follows: Constants contained in `datum` are unaffected, while identifiers are replaced by fresh identifiers that are different from all previously existing identifiers in the sense of `bound-identifier=?`. Two of the resulting identifiers will be `bound-identifier=?` only if they replaced existing `bound-identifier=?` identifiers in `datum` during a single evaluation of the `syntax` form.

These fresh identifiers remain `free-identifier=?` to the original identifiers. This means that a fresh identifier will denote the same binding as the original identifier in `datum` unless macro expansion places an occurrence of it in a binding position.

The core `syntax` form described here has no notion of pattern variable insertion, but is effectively rebound in the scope of `syntax-case` clauses to implement that feature (see below).

Examples:

```
(bound-identifier=? (syntax x) (syntax x))    ==> #f

(let ((y (syntax (x . x))))
  (bound-identifier=? (car y)
                      (cdr y)))               ==> #t


(syntax-object->datum (syntax (x ...)))       ==> (x ...)

(define (generate-temporaries list)
  (map (lambda (ignore) (syntax temp))
       list))
```

Note that `syntax` does not unify identifiers previously distinct in the sense of `bound-identifier=?` occurring in `datum` even if they have the same symbolic name:

```
(let ((x 1))
  (let-syntax
      ((foo (lambda (form)
              (quasisyntax
               (let-syntax
                   ((bar (lambda (_)
                           (syntax (let ((x 2)) ,(cadr form))))))
                 (bar))))))
    (foo x)))                  ==> 1
```

syntax: (**quasisyntax** template)

Constructs a new syntax object from `template`, parts of which may be unquoted using `unquote` or `unquote-splicing`. If no unquoted subexpressions appear at the same nesting level as the outermost `quasisyntax`, the result of evaluating `(quasisyntax template)` is equivalent to the result of evaluating `(syntax template)`. However, if unquoted expressions do appear, they are evaluated and inserted or spliced into the resulting structure according to the rules described for `quasiquote` in R5RS (4.2.6).

To make nested `unquote-splicing` behave in a useful way, the R5RS-compatible extension to `quasiquote` in appendix B of the paper [10] is required mutatis mutandis for `quasisyntax`.

Identifiers introduced when evaluating the `quasisyntax` form are different from all previously existing identifiers in the sense of `bound-identifier=?`. Two of the resulting identifiers will be `bound-identifier=?` only if they replaced existing `bound-identifier=?` identifiers in `template` during a single evaluation of the `quasisyntax` form, with the understanding that the evaluation of any nested, unquoted `syntax` or `quasisyntax` forms counts as part of the evaluation of the enclosing `quasisyntax`.

These fresh identifiers remain `free-identifier=?` to the original identifiers. This means that a fresh identifier will denote the same binding as the original identifier in `datum` unless macro expansion places an occurrence of it in a binding position.

The core `quasisyntax` form described here has no notion of pattern variable insertion, but is effectively rebound in the scope of `syntax-case` clauses to implement that feature (see below).

Examples:

```
(bound-identifier=? (quasisyntax x)
                    (quasisyntax x))                    ==> #f

(quasisyntax ,(bound-identifier=? (quasisyntax x)
                                  (syntax x)))          ==> #t

(let-syntax ((f (lambda (form) (syntax (syntax x)))))
  (quasisyntax ,(bound-identifier=? (f) (f))))          ==> #f

(let-syntax ((m (lambda (_)
                  (quasisyntax
                   (let ((,(syntax x) 1)) ,(syntax x))))))
  (m))  ==> 1
```

In the above, notice that the rule for `bound-identifier=?` equivalence inside `quasisyntax` forms imply the equivalence:

```
(quasisyntax (let ((,(syntax x) 1)) ,(syntax x)))

    <-> (quasisyntax (let ((x 1)) x))
```

The traditional idiom for macro-generating macros containing nested `quasisyntax` forms works correctly:

```
(let-syntax ((m (lambda (form)
                  (let ((x (cadr form)))
                    (quasisyntax
                     (let-syntax ((n (lambda (_)
                                       (quasisyntax
                                        (let ((,(syntax ,x) 4)) ,(syntax ,x))))))
                       (n)))))))
  (m z))  ==> 4
```

The rule for `bound-identifier=?` equivalence inside `quasisyntax` is unique in ensuring that the above macro means exactly the same as the corresponding `syntax-case` macro:

```
(let-syntax ((m (lambda (form)
                  (syntax-case form ()
                    ((_ x) (syntax
                            (let-syntax ((n (lambda (_)
                                              (syntax (let ((x 4)) x)))))
                              (n)))))))))
  (m z))    ==> 4
```

procedure: **(datum->syntax-object** template-identifier obj)

Transforms `obj`, which must be a graph with pairs or vectors as nodes and with symbols or constants as leaves, to a syntax object as follows: Constants in `obj` are unaffected, while symbols appearing in `obj` are replaced by identifiers that behave under `bound-identifier=?`, `free-identifier=?` and `literal-identifier=?` the same as an identifier with the same symbolic name would behave if it had occurred together with `template-identifier` in the same source toplevel expression or was produced during the same evaluation of the `syntax` or `quasisyntax` expression producing `template-identifier`.

If `template-identifier` is a fluid identifier, the symbols in `obj` will also be converted to fluid identifiers.

```
(let-syntax ((m (lambda (_)
                  (let ((x (syntax x)))
                    (let ((x* (datum->syntax-object x 'x)))
                      (quasisyntax
                       (let ((,x 1)) ,x*)))))))
  (m))         ==> 1


(let ((x 1))
  (let-syntax ((m (lambda (form)
                    (quasisyntax
                     (let ((x 2))
                       (let-syntax ((n (lambda (form)
                                         (datum->syntax-object (cadr form) 'x))))
                         (n ,(cadr form)))))))))
    (m z)))    ==> 1
```

`Datum->syntax-object` provides a hygienic mechanism for inserting bindings that intentionally capture existing references. Since composing such macros is a subtle affair, with various incorrect examples appearing in the literature, we present a worked-out example, courtesy of [2]:

```
(define-syntax if-it
  (lambda (x)
    (syntax-case x ()
      ((k e1 e2 e3)
       (with-syntax ((it (datum->syntax-object (syntax k) 'it)))
         (syntax (let ((it e1))
                   (if it e2 e3))))))))

(define-syntax when-it
```

```
    (lambda (x)
      (syntax-case x ()
        ((k e1 e2)
         (with-syntax ((it* (datum->syntax-object (syntax k) 'it)))
           (syntax (if-it e1
                          (let ((it* it)) e2)
                          (if #f #f))))))))

  (define-syntax my-or
    (lambda (x)
      (syntax-case x ()
        ((k e1 e2)
         (syntax (if-it e1 it e2))))))

  (if-it 2 it 3)     ==> 2
  (when-it 42 it)    ==> 42
  (my-or 2 3)        ==> 2
  (my-or #f it)      ==> Error: undefined identifier: it


  (let ((it 1)) (if-it 42 it #f))     ==> 42
  (let ((it 1)) (when-it 42 it))      ==> 42
  (let ((it 1)) (my-or 42 it))        ==> 42
  (let ((it 1)) (my-or #f it))        ==> 1
  (let ((if-it 1)) (when-it 42 it))   ==> 42
```

Notice how `my-or` purposely does not expose `it` to the user. On the other hand, the definition of `when-it` explicitly re-exports `it` to the use site, while preserving referential transparency in the last example.

procedure: (**syntax-object->datum** syntax-object)

Transforms a syntax object to a new graph with identifiers replaced by their symbolic names.

procedure: (**make-capturing-identifier** template-identifier symbol)

This procedure returns a fresh identifier that is `free-identifier=?` to `(datum->syntax-object template-identifier symbol)`. The new identifier is not `bound-identifier=?` to any existing identifiers. If the identifier is inserted as a bound identifier in a binding form, the binding will capture any identifiers in the scope of the binding that are `free-identifier=?` to it.

This primitive provides an alternative to `datum->syntax-object` for intentional capture. Since the capture mechanism is now based on `free-identifier=?` equivalence rather than `bound-identifier=?` equivalence, the implementation and the semantics are subtly different. Consider:

```
  (define-syntax if-it
    (lambda (x)
      (syntax-case x ()
        ((k e1 e2 e3)
         (with-syntax ((it (make-capturing-identifier (syntax here) 'it)))
           (syntax (let ((it e1))
                     (if it e2 e3))))))))

  (define-syntax when-it
```

```
    (lambda (x)
      (syntax-case x ()
        ((k e1 e2)
         (syntax (if-it e1 e2 (if #f #f)))))))

  (define-syntax my-or
    (lambda (x)
      (syntax-case x ()
        ((k e1 e2)
         (syntax (let ((thunk (lambda () e2)))
                   (if-it e1 it (thunk))))))))

  (if-it 2 it 3)      ==> 2
  (when-it 42 it)     ==> 42
  (my-or 2 3)         ==> 2
  (my-or #f it)       ==> undefined identifier: it


  (let ((it 1)) (if-it 42 it #f))      ==> 1
  (let ((it 1)) (when-it 42 it))       ==> 1
  (let ((it 1)) (my-or 42 it))         ==> 42
  (let ((it 1)) (my-or #f it))         ==> 1
  (let ((if-it 1)) (when-it 42 it))    ==> 42
```

Notice that `when-it` here is simpler than the above solution using `datum->syntax-object`, since captures do not have to be explicitly propagated. However, it now takes a little more work to prevent propagation of captures, as the `my-or` macro shows. Also, in two cases the answer is different, with explicit bindings here taking precedence over implicit bindings. This behaviour is the same as that of the MzScheme solution suggested in [13], but can be changed by modifying the first argument to `make-capturing-identifier`.

This primitive is useful for implementing expand-time fluid binding forms. The following example illustrates how one may use it to implement the Chez Scheme `fluid-let-syntax` form [6, 7]:

```
  (define-syntax fluid-let-syntax
    (lambda (form)
      (syntax-case form ()
        ((_ ((i e) ...) e1 e2 ...)
         (with-syntax (((fi ...)
                        (map (lambda (i)
                               (make-capturing-identifier i
                                                          (syntax-object->datum i)))
                             (syntax (i ...)))))
           (syntax
            (let-syntax ((fi e) ...) e1 e2 ...)))))))


  (let ((f (lambda (x) (+ x 1))))
    (let-syntax ((g (syntax-rules ()
                      ((_ x) (f x)))))
      (let-syntax ((f (syntax-rules ()
                        ((_ x) x))))
        (g 1))))   ==> 2
```

```
(let ((f (lambda (x) (+ x 1))))
  (let-syntax ((g (syntax-rules ()
                    ((_ x) (f x)))))
    (fluid-let-syntax ((f (syntax-rules ()
                            ((_ x) x))))
      (g 1))))   ==> 1
```

syntax: (**begin-for-syntax** form ...)

This form may only occur in toplevel contexts. Evaluates the forms `form ...` at macro-expansion time from left to right in an environment one reflective level higher than the level at which expansion is being carried out. The return value is unspecified, and the forms are not evaluated again at runtime.

```
(define x 1)
(begin-for-syntax (define x 2))

(let-syntax ((m (lambda (form)
                  (quasisyntax (list x ,x)))))
  (m))  ==> (1 2)
```

By nesting `begin-for-syntax`, we may introduce toplevel bindings and execute computations at arbitrary phase and arbitrary level in the reflective tower.

```
(begin-for-syntax
  (define x 1)
  (begin-for-syntax
    (define x 2)))

(let ((x 0))
  (let-syntax ((foo (lambda (form)
                      (let-syntax ((bar (lambda (form)
                                          (quasisyntax
                                           (quasisyntax
                                            (list x ,x ,,x))))))
                        (bar)))))
    (foo))) ==> (0 1 2)
```

syntax: (**around-syntax** before-exp form after-exp)

When an `around-syntax` form is expanded, the expression `before-exp` is first evaluated, the form `form` is then expanded fully, and finally the expression `after-exp` is evaluated. The result of the expansion is the fully expanded `form`.

Both `before-exp` and `after-exp` are evaluated in an environment one reflective level higher than the level at which expansion is being carried out.

This primitive allows the macro writer to manage information used to control expansion of subexpressions. An example of its use is in the reference `syntax-case` implementation, where `around-syntax` is used to manage the pattern variable environment that controls the expansion of `syntax` templates.

```
(begin-for-syntax (define env (list (syntax a))))

(let-syntax ((foo (lambda (form)
                    (quasisyntax ',env))))
```

```
        (list
         (around-syntax (set! env (cons (syntax b) env))
                        (foo)
                        (set! env (cdr env)))
         (foo)))

             ==> ((b a) (a))
```

procedure: (**syntax-error** obj ...)

Raises a syntax error. The objects `obj ...` are displayed, available source-object correlation information is displayed or provided to debugging tools, and the expander is stopped.

library syntax: (**syntax-case** exp (literal ...) clause ...)

```
clause := (pattern output-expression)
          (pattern fender output-expression)
```

The `syntax-case` form can be written as a macro in terms of the primitives specified above.

Identifiers in a pattern that are not `bound-identifier=?` to any of the identifiers `(literal ...)` are called pattern variables. These belong to the same namespace as ordinary variables and can shadow, or be shadowed by, bindings of the latter. Each pattern is identical to a `syntax-rules` pattern (R5RS 4.3.2), and is matched against the input expression `exp`, which must evaluate to a syntax object, according to the rules of R5RS, section (4.3.2), except that the first position in a pattern is not ignored. When an identifier in a pattern is `bound-identifier=?` to an identifier in `(literal ...)`, it will be matched against identifiers in the input using `literal-identifier=?`. If a pattern is matched but a fender expression is present and evaluates to `#f`, evaluation proceeds to the next clause.

In the `fender` and `output-expression` of each clause, the `(syntax template)` and `(quasisyntax template)` forms are effectively rebound so that pattern variables in `pattern`, and visible pattern variables in nesting `syntax-case` forms, will be replaced in `template` by the subforms they matched in the input. For this purpose, the template in `(syntax template)` is treated identically to a `syntax-rules` template (R5RS 4.3.2). Subtemplates of `quasisyntax` templates that do not contain unquoted expressions are treated identically to `syntax` templates.

The rules for `bound-identifier=?` equivalence of fresh identifiers replacing identifiers in templates that do not refer to pattern variables remain as specified in the sections describing the primitives `syntax` and `quasisyntax` above.

An ellipsis in a template that is not preceded by an identifier is not interpreted as an ellipsis literal. This allows the following idiom for generating macros containing ellipses:

```
(let-syntax ((m (lambda (form)
                  (syntax-case form ()
                    ((_ x ...)
                     (with-syntax (((::: (syntax ...)))
                       (syntax
                        (let-syntax ((n (lambda (form)
                                          (syntax-case form ()
```

```
                                                ((_ x ... :::)
                                                 (syntax `(x ... :::)))))))))
                                  (n a b c d)))))))))
          (m u v))
                    ==> (a b c d)
```

library syntax: (**with-syntax** template)

As in [6, 7], `with-syntax` expands to an instance of `syntax-case`:

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      ((_ ((p e0) ...) e1 e2 ...)
       (syntax (syntax-case (list e0 ...) ()
                 ((p ...) (begin e1 e2 ...))))))))
```

library syntax: (**syntax-rules** template)

See R5RS, section (4.3.2). Definable in terms of `syntax-case` as [6. 7]:

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (i ...) ((keyword . pattern) template) ...)
       (syntax (lambda (x)
                 (syntax-case x (i ...)
                   ((dummy . pattern) (syntax template))
                   ...)))))))
```

Reader extensions

The reader extensions `#'e` for `(syntax e)` and #`e for `(quasisyntax e)` are recommended.

Implementation

To date, three implementations are available: The reference implementation, an implementation for the CHICKEN Scheme compiler, and a PLT DrScheme-integrated version with source location tracking and syntax highlighting.

The reference implementation uses the forms and procedures specified in R5RS. It does not require R5RS macros or any other existing macro system. In addition, it uses an interaction-environment, no-argument variant of `eval`, which is available in most Scheme systems.

The reference implementation is available here (`.tgz`). It has been successfully run on at least Chez, CHICKEN, Gambit and MzScheme. The implementation was strongly influenced by the explicit renaming system [8, 11]. It uses a fast imperative hygiene algorithm, based on a shallow binding paradigm, that is eager and linear in expression size.

The proposal of this SRFI has been implemented as a language extension for the CHICKEN Scheme compiler, available here.

A PLT DrScheme-integrated version that implements source-object correlation tracking and provides correct syntax highlighting for both expansion-time and runtime errors is also available here (`.tgz`).

Source-object correlation

The specification requires compound syntax objects to be represented as ordinary Scheme lists or vectors. This means that we cannot store source location information for these in the syntax object itself.

Given this representation, a method to track source information was worked out by Dybvig and Hieb [2]: The expander simply maintains a record of the source information for each list and each (occurrence of each) identifier in some external data structure, e.g., a hash table. This would require an extra wrapper for each identifier occurrence to give it its own identity.

Acknowledgments

Special thanks to Kent Dybvig, Matthew Flatt and Felix Winkelmann for helpful comments.

References

[1] André van Tonder - Portable macros and modules
http://www.het.brown.edu/people/andre/macros/index.htm
[2] R. Kent Dybvig - Private communication.
[3] Marcin 'Qrczak' Kowalczyk - Message on comp.lang.scheme:
https://groups.google.com/forum/#!msg/comp.lang.scheme/9x-1PRs6dMk/29tRp0xeB7cJ
[4] Ben Rudiak-Gould - Message on comp.lang.scheme:
https://groups.google.com/forum/#!msg/comp.lang.scheme/9x-1PRs6dMk/jig8hSd2DBgJ
[5] Matthew Flatt - Composable and Compilable Macros You Want it When?
https://www.cs.utah.edu/plt/publications/macromod.pdf
[6] R. Kent Dybvig - Schez Scheme user's guide:
https://cisco.github.io/ChezScheme/#docs
[7] Robert Hieb, R. Kent Dybvig and Carl Bruggeman - Syntactic Abstraction in Scheme. R. Kent Dybvig - Writing hygienic macros in syntax-case
https://doc.lagout.org/programmation/Lisp/Scheme/Programming%20With%20Hygienic%20Macros%20-%20R%20K%20Dybvig.pdf
[8] William D. Clinger - Hygienic macros through explicit renaming.
https://dl.acm.org/doi/10.1145/1317265.1317269 or
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.5184
[9] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen and Bruce F. Duba - Hygienic macro expansion
https://web.cs.ucdavis.edu/~devanbu/teaching/260/kohlbecker.pdf
[10] Alan Bawden - Quasiquotation in Lisp
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.1290

[11] Richard Kelsey and Jonathan Rees - The Scheme 48 implementation
http://s48.org/
[12] Robert Hieb, R. Kent Dybvig - A compatible low-level macro facility Revised(4) Report on the Algorithmic Language Scheme (appendix)
http://people.csail.mit.edu/jaffer/r4rs.pdf
[13] Matthew Flatt - Introducing an Identifier into the Lexical Context of a Macro Call
https://lists.racket-lang.org/users/archive/2004-October/006893.html
[14] Christian Queinnec - Macroexpansion Reflective Tower
https://christian.queinnec.org/PDF/towexp.pdf

Copyright

Author: André van Tonder
Editor: Francisco Solsona