Title

Eager `syntax-rules`

Author

Marc Nieper-Wißkirchen

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-148@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2016-12-31
- Draft #1 published: 2017-01-01
- Draft #2 published: 2017-07-03
- Draft #3 published: 2017-07-25
- Draft #4 published: 2017-08-03
- Finalized: 2017-08-08

Abstract

Writing powerful `syntax-rules` macros is hard because they do not compose well: The arguments of a macro expansion are not expanded. This SRFI defines an easy to comprehend high-level system for writing powerful, composable (or *eager*) macros, two of whose defining features are that its macro arguments are (in general) eagerly expanded and that it can be portably implemented in any Scheme implementation conforming to the R7RS.

Rationale

One defining aspect of the contemporary Scheme programming language is the provision of a high-level hygienic macro system, which is known as the `syntax-rules` macro system. While `syntax-rules` makes it easy and very natural to write simple macros — see, for example, the definition of the `and` form in the R7RS — writing more powerful macros, *e.g.* a portable hygienic pattern matcher, soon looks like black art to many Scheme users.

The main reason for this is that `syntax-rules`-macros do not compose well due to the head-first rewriting system on which they are based: The arguments of each macro use remain essentially unexpanded.

In order to discourage people from refraining from the beauty of `syntax-rules` by employing unhygienic low-level macro systems (or building on run-time evaluation when expansion at compile time would be beneficial to running time), this SRFI defines a custom macro transformer `em-syntax-rules`, which allows one to write eager macros, while retaining the hygiene and beauty of `syntax-rules`. This high-level system of eager macros is based on the idea of CK-macros, which has been popularized by Oleg Kiselyov.

There are at least three arguments in favor of standardizing a system of eager macros based on `syntax-rules`. Firstly, some standardization makes it simpler to understand and share each other's (macro) code. Secondly, by standardizing this system, future additions to the standard in the form of primitive eager macros (*e.g.* an eager macro keyword `em-number?` that can discern whether some syntax is a number literal or not) are made possible, and can be used to improve the expressiveness of `syntax-rules` while retaining hygiene. Thirdly, it makes very much sense to implement this proposal not as a portable library on top of `syntax-rules`, but natively by a Scheme implementation.

Examples

One should get a good feeling about how the macro facility presented in this SRFI is to be used by studying the implementations of the various predefined eager macros (see below), which are contained in a file of the sample implementation.

Factorial

Of course, an example involving the factorial must not be forgotten. The following example produces a vector literal whose elements are all 3! = 6 permutations of the list `(1 2 3)` at macro expansion time.

```
(em (em-list->vector (em-fact '(1 2 3))))
```

The `letrec`-binding construct

The definition of `letrec` as given in the appendix 7.3 of the R7RS can be rewritten as an eager macro as follows:

```
(define-syntax letrec
  (em-syntax-rules ()
    ((letrec ((var1 init1) ...) body ...)
     ((em-generate-temporaries '(var1 ...)) => '(temp1 ...))
      (em '(let ((var1 <undefined>) ...)
        (let ((temp1 init1) ...)
            (set! var1 temp1)
            ...
            body ...)))))))
```

The `do`-iteration construct

The definition of `do` as given in the appendix 7.3 of the R7RS can be rewritten as an eager macro as follows without "using a trick":

```
(define-syntax do
  (em-syntax-rules ()
    ((do ((var init . step*) ...)
         (test expr ...)
       command ...)
     (em `(letrec
          ((loop (lambda (var ...)
             (if test
               (begin
                 (if #f #f)
```

```
                  expr ...)
                (begin
                  command
                  ...
                  (loop ,(em-if (em-null? 'step*)
                                'var
                                (em-car 'step*))
                        ...))))))
        (loop init ...))))))
```

Pattern matcher

The following is a library implementing a very simple pattern patcher using the macro facility described in this SRFI:

```
(define-library (example simple-match)
  (export simple-match)
  (import (scheme base)
          (srfi 2)
          (srfi 148))
  (begin
    (define-syntax simple-match
      (em-syntax-rules ()
        ((simple-match expr (pattern . body) ...)
         (em
          `(call-with-current-continuation
            (lambda (return)
              (let ((e expr))
                (or (and-let*
                        ,(compile-pattern 'pattern 'e)
                      (call-with-values (lambda () . body) return))
                    ...
                    (error "does not match" expr)))))))))

    (define-syntax compile-pattern
      (em-syntax-rules ()
        ((compile-pattern '() 'e)
         '(((null? e))))
        ((compile-pattern '(pattern1 pattern2 ...) 'e)
         `(((not (null? e)))
           (e1 (car e))
           (e2 (cdr e))
           ,@(compile-pattern 'pattern1 'e1)
           ,@(compile-pattern '(pattern2 ...) 'e2)))
        ((compile-pattern 'x 'e)
         (em-if (em-symbol? 'x)
           '((x e))
           '(((equal? x e)))))))))
```

The simple matcher can then be used in a program or at a REPL as follows:

```
(import (scheme base)
        (example simple-match))

(simple-match 10
 (10 'ten)
 ((11 x) x))     ⇒ 'ten
```

```
(simple-match '(11 eleven)
 (10 'ten)
 ((11 x) x))     ⇒ 'eleven
```

Specification

Overview

An eager macro is a (hygienic) macro whose keyword is bound to a transformer specified by an instance of `em-syntax-rules` instead of an instance of `syntax-rules`. A transformer spec using `em-syntax-rules` is very similar to the well-known `syntax-rules`, namely of one of the two forms:

```
(em-syntax-rules (<literal> ...)
  (<eager macro-rule ...>))
(em-syntax-rules <ellipsis> (<literal> ...)
  (<eager macro-rule ...>))
```

Each `<eager macro-rule>` is also very similar to a `<syntax rule>`, namely of one of the forms:

```
((<pattern> ...) <pattern binding spec> ... <template>)
((<pattern> ... <pattern> <ellipsis> <pattern> ...) <pattern binding spec> ... <template>)
```

When we forget about the `<pattern binding spec>`s for a moment, the difference between an eager macro and a `syntax-rules`-macro lies in how the input elements corresponding to the `<pattern>`s above are expanded before a macro use is expanded: `syntax-rules` macros never expand their arguments before evaluation. On the other hand, `em-syntax-rules` macros expand those input elements whose corresponding patterns are of the form `'<pattern>`. The output of an eager macro is handled as follows: If the template is of the form `'<template>`, the eager macro expands into `<template>` (with the pattern variables substituted); if the template is unquoted, the eager macro expands into the expansion of the template. This behavior is analogous to how evaluation of quoted datums is handled in the core Scheme language.

For example, the following expression evaluates to `42`:

```
(letrec-syntax
    ((foo (em-syntax-rules ()
            ((foo 'a) (bar 'a))))
     (bar (em-syntax-rules ()
            ((bar 'a) '(define a 42))))
     (baz (em-syntax-rules ()
            ((baz ignore 'a) 'a))))
  (foo (baz (not-expanded) 'a))
  a)
```

One more extension is available when compared with ordinary `syntax-rules` macros, namely the `<pattern binding spec>`s, which are of the form:

```
(<template> => <pattern>)
```

These work as follows: Before the final template in an eager macro use is expanded, the pattern variables in `<pattern>` are bound according to the expansion of `<template>` (which may contain pattern variables introduced earlier) and are available in subsequent templates.

The following example evaluates to the list `((1 x) (2 x) (3 x))`.

```
(letrec-syntax
    ((foo (em-syntax-rules ()
            ((foo 'x) ((bar 'x) => '(b c)) '(define x '(b (2 x) c)))))
     (bar (em-syntax-rules ()
            ((bar 'x) '((1 x) (3 x))))))
  (foo 'x)
  x)
```

Syntax

This SRFI extends 7.1 of the R7RS as follows:

```
<transformer spec>
  ⟶ (em-syntax-rules (<identifier> ...) <eager macro rule> ...)
  ⟶ (em-syntax-rules <ellipsis> (<identifier> ...) <eager macro rule> ...)

<eager macro rule>
  ⟶ (<top-level pattern> <pattern binding spec> ... <top-level template>)

<pattern binding spec>
  ⟶ (<top-level template> => <top-level pattern element>)

<top-level pattern>
  ⟶ (<identifier> <top-level pattern element> ...)
  ⟶ (<identifier> <top-level pattern element> ... <top-level pattern element>
     <ellipsis> <top-level pattern element>)

<top-level template>
  ⟶ <quoted template>
  ⟶ <template>

<top-level pattern element>
  ⟶ <quoted pattern>
  ⟶ <quasiquoted pattern>
  ⟶ <pattern>

<quoted pattern>
  ⟶ '<pattern>

<quasiquoted pattern>
  ⟶ `<pattern>

<quoted template>
  ⟶ '<template>

<macro use>
  ⟶ <eager macro use>

<eager macro use>
  ⟶ (<keyword> <eager macro datum> ...)
  ⟶ (em (<keyword> <eager macro datum> ...))
```

```
<eager macro datum>
  ⟶ <quoted datum>
  ⟶ <quasiquoted datum>
  ⟶ <eager macro use>
  ⟶ <datum>

<quoted datum>
  ⟶ '<datum>

<quasiquoted datum>
  ⟶ `<datum>

<slot or eager macro datum>
  ⟶ <>
  ⟶ <eager macro datum>
```

Auxiliary syntax

`=>`

`...`

`<>`

The bindings of these three identifiers coincide with the bindings of these identifiers as exported from `(scheme base)` and SRFI 26, respectively.

Note: SRFI 26 was finalized long before the current standard, R7RS, came into being. By the way literal identifiers in transformers are matched in R7RS and auxiliary syntax is handled in R7RS, the identifier `<>` *(and* `<...>`*) should really be bound by SRFI 26 when SRFI 26 is used in an R7RS system. (The sample implementation of this SRFI — see below — includes such an implementation of SRFI 26.)*

Semantics

Keywords bound to a transformer specified by an instance of `em-syntax-rules` define keywords for *eager macros*.

Instances of an eager macro have one of the following forms:

```
(<keyword> <eager macro datum> ...)

(em (<keyword> <eager macro datum> ...))
```

Here, <keyword> is a keyword for an eager macro. The former form has to be used in definition contexts, the latter when the macro use expands into an expression (see also below).

The syntax and semantics of the `<eager macro rule>`s together with the (optional) specification of `<ellipsis>` and the `<literal>`s are as in the case of a `syntax-rules` transformer spec with the following exceptions and additions:

- A top-level pattern must be a list pattern without a dotted tail. (No expressive power is lost because at top-level, every pattern with a dotted tail can be rewritten, together with the corresponding template, into a proper list pattern by making use of the `<ellipsis>`).

- A `<pattern>` occurring in the list of top-level patterns can also be a *quoted pattern*, that is a pattern of the form `'<pattern>`. A quoted pattern does not count as a list pattern.
- The identifier `quote` may appear amongst the list of `<literal>`s. (As a quoted pattern does not count as a list pattern, the `quote` identifying quoted patterns is never considered a literal identifier.)
- It is an error if there are two rules such that there is a quoted pattern in one rule and a list pattern at the same position in the other rule.
- A quoted pattern `'<pattern>` matches an input element if and only if the input element is an eager macro use and the corresponding pattern `<pattern>` matches the result of the expansion of the eager macro use, or if it is a quoted datum (or a self-quoting syntax element; see below) and the pattern `<pattern>` matches the expression, or if it is a quasiquoted datum and the pattern `<pattern>` matches the expansion of the quasiquoted datum (see below). In other words, unquoted arguments (or unquoted elements in quasiquoted arguments) to quoted patterns that are eager macro uses are expanded before they are matched (*call-by-value* semantics, different from the *call-by-reference* semantics of ordinary `syntax-rules`-macros).
- A top-level `<template>` can also be a *quoted template*, that is a template of the form `'<template>`, or a *quasiquoted template*, that is a template of the form `` `<template> ``. A quoted template or quasiquoted template at top-level does not count as a list template.
- If the top-level template is a quoted template `'<template>`, the macro use is expanded as if it was a `syntax-rules` macro with `<template>` as the top-level template.

  If the top-level template is a quasiquoted template `` `<template> ``, the macro use is expanded as if it was a `syntax-rules` macro with `<template>` as the top-level template. However, unquoted template elements, that is template elements of the form `,<template element>`, inside the quasiquoted template are expanded first and are handled as if they are an unquoted top-level template of an eager macro (see below). `Unquote-splicing` and nested quasiquotations are handled analogously to quasiquotation in the core Scheme language.
- If the top-level template is an unquoted template `<template>`, the macro use is expanded as if it was a `syntax-rules` macro. It is an error if the expanded output is not an eager macro use (or a self-quoting syntax element; see below).
- Syntax elements that are not proper lists are self-quoting. Whenever they appear as input or output elements of eager macros where a quoted syntax element is expected, they are implicitly quoted.
- An eager macro use is only allowed in definition contexts, except when the macro use or top-level template of the macro is wrapped into `em` (see also below) and does expand in an expression.
- When `<pattern binding spec>`s appear in an eager macro rule that is matched, the `<pattern binding spec>`s are processed in left-to-right order before the final template of the rule is expanded: For each pattern binding spec `(<template> => <top-level pattern element>)`, the `<template>` is expanded according to the above rules and matched against the `<top-level pattern element>`. The pattern variables introduced by the `<top-level pattern element>` are available in subsequent templates. It is an error if

the `<top-level pattern element>` does not match or if a pattern variable is introduced more than once.

Eager macros

In this subsection, all predefined eager macros are listed. It is not specified whether these macros are able to deal with circular lists and vectors.

Index

General

`(em <eager macro use>)`

Expands into the expansion of <eager macro use> with the additional property that it is allowed in an expression context. It is an error if <eager macro use> does not expand into an expression.

Rationale: That eager macro uses that expand into expressions have to be wrapped in `em` forms to ensure that this SRFI can be portably implemented. If Scheme had a `let(rec)-syntax` form that spliced its body into the outer scope, this auxiliary `em` wouldn't be necessary.

Schemes that provide a splicing `let(rec)-syntax` form can implement `em` as a no-op so that it can be omitted and eager macro uses can appear equally in definition or expression context. It is recommended to actually require that `em` becomes a no-op in a future Scheme standard, which provides a splicing `let(rec)-form` and which includes the specification of this SRFI. In that case, `em` should become deprecated.

`(em-cut <slot or eager macro datum>`$_1$ `<slot or eager macro datum>`$_2$ `...)`

`(em-cut <slot or eager macro datum>`$_1$ `<slot or eager macro datum>`$_2$ `... <> <ellipsis>)`

Expands into an eager macro that accepts as many input elements as there are slots (elements of the form <>) amongst the `<eager macro datum>`s. When this eager macro is used, it expands into the expansion of `(<slot or eager macro datum>`$_1$ `<slot or eager macro datum>`$_2$`)` where the slots are replaced by its input elements in order. In case of the second syntax with `(...`

`<> <ellipsis>)`, the resulting eager macro takes a variable number of extra input elements that are spliced in place of the sequence `<> <ellipsis>`. (The semantics are analogous to the semantics of the `cut` macro of SRFI 26, except for the fact that `<...>` is replaced by the sequence `<> <ellipsis>`, which would not have been possible to implement when SRFI 26 was finalized). It is an error to use `em-cut` standalone, that is, not as an input or output element of an eager macro.

`(em-cute <slot or eager macro datum>`$_1$ `<slot or eager macro datum>`$_2$ `...)`

`(em-cute <slot or eager macro datum>`$_1$ `<slot or eager macro datum>`$_2$ `... <> <ellipsis>)`

Analogous to the `em-cut`-macro uses, except that the non-slot `<slot or eager macro datum>`s are expanded when the `em-cute`-macro use is expanded, not when the resulting eager macro is expanded. (The semantics are analogous to the semantics of the `cute` macro of SRFI 26). It is an error to use `em-cute` standalone, that is, not as an input or output element of an eager macro.

Rationale: While the `cut` and `cute` macros of SRFI 26 are not really essential for coding in Scheme because Scheme has closures, there is no such thing as closures of eager macros. However, `em-cut` and `em-cute` can be used to emulate closures.

`(em-constant <constant>)`

Expands into an eager macro that accepts an arbitrary number of input elements and that always expands into the expansion of <constant>. It is an error to use `em-constant` standalone, that is, not as an input or output element of an eager macro.

`(em-quote <eager macro datum>)`

Expands into the quotation of the expansion of the `<eager macro datum>`.

`(em-eval <eager macro datum>)`

Expands into the expansion of the expansion of <eager macro datum>.

`(em-apply <eager macro datum>`$_1$ `<eager macro datum>... <eager macro datum>`$_2$`)`

Semantically equivalent to:

```
(em-append (em-list <eager macro datum>1 <eager macro datum> ...) <eager macro datum>2)
```

It is an error if `<eager macro datum>`$_2$ does not expand into a list.

`(em-call <macro> <input> ...)`

Semantically equivalent to:

```
(em-apply <macro> <input> ... '())
```

`(em-error <message> <argument> ...)`

Expansion yields a syntax error whose message is the expansion of `<message>` and whose arguments are the expansions of the `<argument>`s.

`(em-gensym)`

Expands into a fresh identifier that is not `em-bound-identifier=?` (see below) to any already existing identifier.

`(em-generate-temporaries <list>)`

Expands into a list of pairwise different (in the sense of `bound-identifier=?`) identifiers whose length is the number of elements in the expansion of `<list>`. It is an error if `<list>` does not expand into a list.

Boolean logic

`(em-if <eager macro datum>`$_1$ `<eager macro datum>`$_2$ `<eager macro datum>`$_3$`)`

Expands into the expansion of `<eager macro datum>`$_2$ if `<eager macro datum>`$_1$ does not expand into `#f`, and into the expansion of `<eager macro datum>`$_3$ otherwise.

`(em-not <eager macro datum>)`

Expands into `#f` if `<eager macro datum>` does not expand into `#f`, and into `#t` otherwise.

`(em-or <eager macro datum> ...)`

Expands into the expansion of the first `<eager macro datum>` (in left-to-right order) that does not expand into `#f`, and does not expand any later `<eager macro datum>`s; expands into `#f` if all `<eager macro datum>`s expand into `#f`.

`(em-and <eager macro datum> ...)`

Expands into the expansion of the last `<eager macro datum>` (in left-to-right order) if all previous `<eager macro datum>`s do not expand into `#f`, and does not expand any `<eager macro datum>`s after the first that expands into `#f`; expands into `#f` otherwise.

`(em-null? <eager macro datum>)`

Expands into `#t` if `<eager macro datum>` expands into the empty list, and into `#f` otherwise.

`(em-pair? <eager macro datum>)`

Expands into `#t` if `<eager macro datum>` expands into a pair, and into `#f` otherwise.

`(em-list? <eager macro datum>)`

Expands into `#t` if `<eager macro datum>` expands into a (proper) list, and into `#f` otherwise.

`(em-boolean? <eager macro datum>)`

Expands into `#t` if `<eager macro datum>` expands into a boolean literal, and into `#f` otherwise.

`(em-vector? <eager macro datum>)`

Expands into `#t` if `<eager macro datum>` expands into a vector literal, and into `#f` otherwise.

`(em-symbol? <eager macro datum>)`

Expands into `#t` if `<eager macro datum>` expands into a symbol, and into `#f` otherwise.

`(em-bound-identifier=? <identifier> <input>)`

Expands into `#t` if `<identifier>` and `<input>` both expand into symbols that denote the same identifier (in the expansion context), and into `#f` otherwise. It is an error if `<identifier>` does not expand into a symbol.

`(em-free-identifier=? <identifier>`$_1$ `<identifier>`$_1$`)`

Expands into `#t` if `<identifier>`$_1$ and `<identifier>`$_2$ both expand into symbols such that the identifiers denoted by them have the same meaning (in the expansion context), and into `#f` otherwise. It is an error if `<identifier>`$_1$ and `<identifier>`$_2$ do not expand into a symbol.

`(em-equal? <eager macro datum>`$_1$ `<eager macro datum>`$_2$`)`

Expands into `#t` if `<eager macro datum>`$_1$ and `<eager macro datum>`$_2$ are the same, and into `#f` otherwise. Pairs and vectors are compared recursively, and symbols are compared using `em-bound-identifier=?`.

List processing

Constructors

`(em-cons <input>`$_1$ `<input>`$_2$`)`

Expands into a pair whose car is the expansion of `<input>`$_1$ and whose cdr is the expansion of `<input>`$_2$.

`(em-cons* <element> ... <tail>)`

Expands into the expansion of `` `(,<element> ... . ,<tail>) ``

`(em-list <element> ...)`

Expands into a list whose elements are the expansions of the `<element>`s.

`(em-make-list <k> <element>)`

Expands into a list whose length is the length of the expansion of `<k>`. The elements of the list are the expansion of `<element>`. It is an error if `<k>` does not expand into a list.

Selectors

`(em-car <pair>)`

Expands into the car of the expansion of `<pair>`. It is an error if `<pair>` does not expand into a pair.

`(em-cdr <pair>)`

Expands into the cdr of the expansion of `<pair>`. It is an error if `<pair>` does not expand into a pair.

`(em-caar <pair>)`

`(em-cadr <pair>)`

`(em-cdar <pair>)`

```
(em-cddr <pair>)
```

Semantically equivalent to `(em-car (em-car <pair>))`, `(em-car (em-cdr <pair>))`, `(em-cdr (em-car <pair>))`, `(em-cdr (em-cdr <pair>))`.

```
(em-first <list> ...)
```

```
(em-second <list> ...)
```

```
(em-third <list> ...)
```

```
(em-fourth <list> ...)
```

```
(em-fifth <list> ...)
```

```
(em-sixth <list> ...)
```

```
(em-seventh <list> ...)
```

```
(em-eighth <list> ...)
```

```
(em-ninth <list> ...)
```

```
(em-tenth <list> ...)
```

Expands into the first, second, third, ... element of the expansion of `<list>`. It is an error if `<list>` does not expand into a list of sufficient length. (Analogous to `first`, ..., `tenth` from SRFI 1).

```
(em-list-tail <list> <k>)
```

```
(em-list-ref <list> <k>)
```

Expands into the *k*th element of the expansion of `<list>`, where *k* is the length of the expansion of `<k>`. It is an error if `<list>` and `<k>` do not expand into lists.

```
(em-take <list> <k>)
```

Expands into the sublist consisting of the first *k* elements (in left-to-right order) of the expansion of `<list>`, where *k* is the length of the expansion of `<k>`. It is an error if `<list>` and `<k>` do not expand into lists. (Analogous to `take` from SRFI 1.)

```
(em-drop <list> <k>)
```

Expands into the sublist of the expansion of `<list>` obtained by omitting as many elements on the left as the expansion of `<k>` has elements. It is an error if `<list>` and `<k>` do not expand into lists. (`Em-drop` is an alternative name to `list-tail`, analogously to `drop` from SRFI 1.)

```
(em-take-right <list> <k>)
```

Expands into the sublist consisting of the last *k* elements (in left-to-right order) of the expansion of `<list>`, where *k* is the length of the expansion of `<k>`. It is an error if `<list>` and `<k>` do not expand into lists. (Analogous to `take-right` from SRFI 1.)

```
(em-drop-right <list> <k>)
```

Expands into the sublist consisting of all but the last *k* elements (in left-to-right order) of the expansion of `<list>`, where *k* is the length of the expansion of `<k>`. It is an error if `<list>` and `<k>` do not expand into lists. (Analogous to `drop-right` from SRFI 1.)

`(em-last <pair>)`

Expands into the last element of the expansion of `<pair>`. It is an error if `<pair>` does not expand into a (possibly dotted) list. (Analogous to `last` from SRFI 1.)

`(em-last-pair <pair>)`

Expands into the last pair of the expansion of `<pair>`. It is an error if `<pair>` does not expand into a (possibly dotted) list. (Analogous to `last-pair` from SRFI 1.)

Miscellaneous

`(em-append <list> ...)`

Expands into a (possibly improper) list that is the concatenation of the expansions of the `<list>`s. It is an error if any of the `<list>`s but the last does not expand into a proper list.

`(em-reverse <list>)`

Expands into a list whose elements are the elements of the expansion of `<list>` in reverse order. It is an error if `<list>` does not expand into a list.

Folding, unfolding and mapping

`(em-fold <proc> <nil> <list> ...)`

Expands into the expansion `<nil>` if at least one of the `<list>`s expands into the empty list. Expands into the expansion of `(em-fold <proc> (em-call <proc> (em-car <list>) ... <nil>) (em-cdr <list>) ...)` otherwise. It is an error if the `<list>`s do not expand into lists. (Analogous to `fold` from SRFI 1.)

`(em-fold-right <proc> <nil> <list> ...)`

Expands into the expansion `<nil>` if at least one of the `<list>`s expands into the empty list. Expands into the expansion of `(em-call <proc> (em-car <list>) ... (em-fold-right <proc> <nil> (em-cdr <list>) ...))` otherwise. It is an error if the `<list>`s do not expand into lists. (Analogous to `fold-right` from SRFI 1.)

`(em-unfold <stop?> <mapper> <successor> <seed> <tail-mapper>)`

`(em-unfold <stop?> <mapper> <successor> <seed>)`

Expands into the expansion of `(<tail-mapper> <seed>)` if (`<stop?> <seed>`) does not expand into `#f`. Expands into the expansion of `(em-cons (<mapper> <seed>) (em-unfold <stop?> <mapper> <successor> (<successor> <seed>) <tail-mapper>)` otherwise. If `<tail-mapper>` is omitted, it defaults to `(em-constant '())`. (Analogous to `unfold` from SRFI 1.)

`(em-unfold-right <stop?> <mapper> <successor> <seed> <tail>)`

`(em-unfold-right <stop?> <mapper> <successor> <seed>)`

Expands into the expansion of `<tail>` if `(<stop?> <seed>)` does not expand into `#f`. Expands into the expansion of `(em-unfold-right <stop?> <mapper> <successor> (<successor> <seed>) (em-cons (<mapper> <seed>) <tail>))` otherwise. If `<tail>` is omitted, it defaults to `'()`. (Analogous to `unfold-right` from SRFI 1.)

`(em-map <proc> <list> ...)`

Expands into the empty list if at least one the `<list>`s expands into the empty list. Expands into the expansion of `(em-cons (<proc> <head> ...) (em-map <proc> <tail> ...))` if the `<list>`s expand into `(<head> . <tail>)`.

`(em-append-map <proc> <list> ...)`

Expands into the empty list if at least one the `<list>`s expands into the empty list. Expands into the expansion of `(em-append (<proc> <head> ...) (em-map <proc> <tail> ...))` if the `<list>`s expand into `(<head> . <tail>)`. (Analogous to `append-map` from SRFI 1.)

Filtering

`(em-filter <pred> <list> ...)`

Expands into a list consisting of those elements *element* of the expansion of `<list>`, in order, for which `(em-call <pred> element)` does not expand into `#f`. It is an error if `<list>` does not expand into a list. (Analogous to `filter` from SRFI 1.)

`(em-remove <pred> <list> ...)`

Expands into a list consisting of those elements *element* of the expansion of `<list>`, in order, for which `(em-call <pred> element)` expands into `#f`. It is an error if `<list>` does not expand into a list. (Analogous to `remove` from SRFI 1.)

Searching

`(em-find <pred> <list>)`

Expands into the first element *element* of the expansion of `<list>` for which `(em-call <pred> element)` does not expand into `#f`. Expands into `#f` if there is no such *element*. It is an error if `<list>` does not expand into a list. (Analogous to `find` from SRFI 1.)

`(em-find-tail <pred> <list>)`

Expands into the first pair *pair* of the expansion of `<list>` for which `(em-call <pred> (em-car pair))` does not expand into `#f`. Expands into `#f` if there is no such *pair*. It is an error if `<list>` does not expand into a list. (Analogous to `find-tail` from SRFI 1.)

`(em-take-while <pred> <list>)`

Expands into the longest initial sublist consisting of those *element*s of the expansion of `<list>` for which `(em-call <pred> element)` does not expand into `#f`. It is an error if `<list>` does not expand into a list. (Analogous to `take-while` from SRFI 1.)

`(em-drop-while <pred> <list>)`

Expands into the first pair *pair* of the expansion of `<list>` for which `(em-call <pred> (em-car pair))` expands into `#f`. Expands into `#f` if there is no such *pair*. It is an error if `<list>` does not expand into a list. (Analogous to `drop-while` from SRFI 1.)

`(em-any <pred> <list> ...)`

Expands into `#f` if at least one of the `<list>`s expands into the empty list. Expands into the expansion of `(em-or (em-call <pred> (em-car <list>) ...) (em-any (em-cdr <list>) ...))` otherwise. It is an error if `<list>` does not expand into a list. (Analogous to `em-any` from SRFI 1.)

```
(em-every <pred> <list> ...)
```

Expands into `#f` if at least one of the `<list>`s expands into the empty list. Expands into the expansion of `(em-and (em-call <pred> (em-car <list>) ...) (em-any <pred> (em-cdr <list>) ...))` otherwise. It is an error if `<list>` does not expand into a list. (Analogous to `em-any` from SRFI 1.)

```
(em-member <obj> <list> <compare>)
```

```
(em-member <obj> <list>)
```

Expands into the expansion of `(em-find-tail (em-cut <compare> <obj> <>) <list>)`. If the input `<compare>` is omitted, it defaults to `em-equal?`.

Association lists

```
(em-assoc <key> <alist> <compare>)
```

```
(em-assoc <key> <alist>)
```

Expands into the first element *element* of the expansion of `<alist>` for which `(em-call <compare> <key> (em-car element))` does not expand into `#f`. Expands into `#f` if there is no such *pair*. If the input `<compare>` is omitted, it defaults to `em-equal?`. It is an error if `<alist>` does not expand into an association list, a list of pairs.

```
(em-alist-delete <key> <alist> <compare>)
```

```
(em-alist-delete <key> <alist>)
```

Expands into an association list consisting of those *association*s of the expansion of `<alist>` in order such that `(em-call <compare> <key> (em-car element))` expands into `#f`. If the input `<compare>` is omitted, it defaults to `em-equal?`. It is an error if `<alist>` does not expand into an association list, a list of pairs. (Analogous to `em-alist-delete` from SRFI 1.)

Set operations

The following operations are analogous to the set operations on lists from SRFI 1.

```
(em-set<= <compare> <list> ...)
```

Expands into `#t` if the elements of the expansion of the `<list>`s form an increasing tower of sets, where $element_1$ and $element_2$ are considered equal if `(em-call <compare>` $element_1$ $element_2$`)` does not expand into `#f`. It is an error if any of the `<list>`s do not expand into a list.

```
(em-set= <compare> <list>1 <list> ...)
```

Expands into `#t` if the elements of the expansion of the `<list>`s each form the same set as the elements of the expansion of `<list>`$_1$, where $element_1$ and $element_2$ are considered equal if `(em-`

`call <compare> element1 element2)` does not expand into `#f`. It is an error if `<list>1` or any of the `<list>`s do not expand into a list.

`(em-set-adjoin <compare> <list> <element> ...)`

Expands into a list consisting of the elements of the expansion of `<list>` and the expansion of those `<element>` added successively to the front, that are not already elements of the list to which they are added. Here, *element1* and *element2* are considered equal if `(em-compare element1 element2)` does not expand into `#f`. It is an error if `<list>` does not expand into a list.

`(em-set-union <compare> <list> ...)`

Expands into a list constructured by adjoining (as in `em-set-adjoin`) the elements of the expansions of the `<list>`s to the empty list. It is an error if the `<list>`s do not expand into a list.

`(em-set-intersection <compare> <list>1 <list> ...)`

Expands into the sublist of the expansion of `<list>1` (as in `em-filter`) consisting of those elements that also elements in the expansions of the `<list>`s, where *element1* and *element2* are considered equal if `(em-call <compare> element1 element2)` does not expand into `#f`. It is an error if `<list>1` or any of the `<list>`s do not expand into a list.

`(em-set-difference <compare> <list>1 <list> ...)`

Expands into the sublist of the expansion of `<list>1` (as in `em-filter`) consisting of those elements that are not elements in any of the expansions of the `<list>`s, where *element1* and *element2* are considered equal if `(em-call <compare> element1 element2)` does not expand into `#f`. It is an error if `<list>1` or any of the `<list>`s do not expand into a list.

`(em-set-xor <compare> <list>1 <list>2)`

Expands into a list consisting of those elements of the expansion of `<list>1` that are not in the expansion of `<list>2` and those elements of `<list>2` that are not in the expansion of `<list>1`, where *element1* and *element2* are considered equal if `(em-call <compare> element1 element2)` does not expand into `#f`. It is an error if `<list>1` or the `<list>2`s do not expand into a list.

Vector processing

`(em-vector <element> ...)`

Expands into a vector whose elements are the expansions of the `<element>`s.

`(em-list->vector <list>)`

Expands into a vector whose elements are the elements of the expansion of `<list>`s. It is an error if `<list>` does not expand into a (proper) list.

`(em-vector->list <vector>)`

Expands into a list whose elements are the elements of the expansion of `<vector>`s. It is an error if `<vector>` does not expand into a vector.

`(em-vector-map <proc> <vector> ...)`

Expands into the empty list if at least one the `<vectors>`s expands into the empty vector. Expands into the expansion of `` `#(,(<proc> *head* ...) ,@(em-vector->list (em-vector-map <proc> *tail* ...))) `` if the `<vector>`s expand into vectors of the form `` `#u(,*head* ,@*tail*) ``.

`(em-vector-ref <vector> <k>)`

Expands into the *k*th element of the expansion of `<vector>`, where *k* is the length of the expansion of `<k>`. It is an error if `<vector>` does not expand into a vector, or `<k>` do not expand into a list.

Combinatorics

`(em-0)`

`(em-1)`

`(em-2)`

`(em-3)`

`(em-4)`

`(em-5)`

`(em-6)`

`(em-7)`

`(em-8)`

`(em-9)`

`(em-10)`

Expands into a list of zero, one, two, ... ten pairwise different elements.

`(em= <list>`$_1$ `<list> ...)`

Expands into `#t` if `<list>`$_1$ and each of the `<list>`s expand into lists of the same length, and into `#f` otherwise. It is an error if `<list>`$_1$ or any of the `<list>`s do not expand into lists.

`(em< <list> ...)`

Expands into `#t` if the `<list>`s expand into lists of strictly increasing length, and into `#f` otherwise. It is an error if any of the `<list>`s do not expand into lists.

`(em<= <list> ...)`

Expands into `#t` if the `<list>`s expand into lists of (not necessarily strictly) increasing length, and into `#f` otherwise. It is an error if any of the `<list>`s do not expand into lists.

`(em> <list> ...)`

Expands into `#t` if the `<list>`s expand into lists of strictly decreasing length, and into `#f` otherwise. It is an error if any of the `<list>`s do not expand into lists.

`(em>= <list> ...)`

Expands into `#t` if the `<list>`s expand into lists of (not necessarily strictly) decreasing length, and into `#f` otherwise. It is an error if any of the `<list>`s do not expand into lists.

`(em-zero? <list>)`

Expands into `#t` if `<list>` expands into the empty list, and into `#f` otherwise. It is an error if `<list>` does not expand into a list.

`(em-even? <list>)`

Expands into `#t` if `<list>` expands into the list with an even number of elements, and into `#f` otherwise. It is an error if `<list>` does not expand into a list.

`(em-odd? <list>)`

Expands into `#t` if `<list>` expands into the list with an odd number of elements, and into `#f` otherwise. It is an error if `<list>` does not expand into a list.

`(em+ <list> ...)`

Expands into a list whose elements are the elements of the expansions of the `<list>`s in left-to-right order. It is an error if the `<list>`s do not expand into a list.

`(em- <list>`$_1$ `<list> ...)`

Expands into a list whose elements are all but the last *k* elements of the expansion of the `<list>`$_1$, where *k* is total number of elements of the expansion of the `<list>`s. It is an error if the `<list>`s do not expand into lists. It is likewise an error if `<list>`$_1$ does not expand into a list or in a list with less then *k* elements.

`(em* <list> ...)`

Expands into a list whose elements are the elements of the cartesian product of the expansions of the `<list>`s in lexicographic left-to-right order. It is an error if the `<list>`s do not expand into lists.

`(em-quotient <list> <k>)`

Expands into a list of each *k*th element in left-to-right order of the expansion of `<list>`, where *k* is the number of elements of the expansion of `<k>`. It is an error if `<list>` or `<k>` do not expand into lists.

`(em-remainder <list> <k>)`

Expands into a smallest tail of the expansion of `<list>` such that the number of elements before the tail is divisible by *k*, where *k* is the number of elements of the expansion of `<k>`. It is an error if `<list>` or `<k>` do not expand into lists.

`(em-fact <list>)`

Expands into a list of all permutations of the expansion of `<list>` in lexicographic left-to-right order. It is an error if `<list>` does not expand into a list.

`(em-binom <list> <k>)`

Expands into a list of all *k*-element (ordered) sublists in lexicographic left-to-right order of the expansion of `<list>`, where *k* is the number of elements of the expansion of `<k>`. It is an error if `<list>` or `<k>` do not expand into lists.

Implementation

This SRFI comes with a complete portable sample implementation of this specification as an R7RS library. The sample implementation is based on SRFI 147, of which an implementation is included in this SRFI. Therefore, when using the sample implementation, one has to import `(srfi 147)` as well (as long as the Scheme system does not support SRFI 147 natively).

An implementation of SRFI 26, from which the identifier `<>` is imported and re-exported, is also included.

The test suite also depends on SRFI 2 and SRFI 64, for which suitable implementations are included as well.

A Scheme system wishing to support this SRFI can either bundle the provided sample implementation with its system libraries, or implement the interface natively, mostly for the sake of expansion speed and for much improved error reporting.

Acknowledgements

This SRFI is based on the ideas on CK-macros outlined by Oleg Kiselyov in *Applicative syntax-rules: macros that compose better*.

The CK abstract machine, on which the macro system is based, has been described by Matthias Felleisen and Daniel P. Friedman in *Control operators, the SECD machine, and the lambda-calculus*.

There is a simple implementation of Oleg Kiselyov's idea by John Croisant as a Chicken egg.

SRFI 53 by Andre van Tonder was an earlier try to "provide a portable framework for writing complex high-level macros that perform nontrivial computations during expansion". At that time, the expressiveness of Scheme (in the form of the R5RS) wasn't sufficient enough to build a usable framework of that sort on top of `syntax-rules` (see this message).

Finally, I would like to thank William Clinger whose experiments with his Larceny implementation of Scheme showed that a previous version of the sample implementation of this SRFI (and of SRFI 147) relied on an unpropitious assumption on the semantics of a corner case of the R7RS macro system.

Copyright

Editor: Arthur A. Gleckler