Title

Basic `syntax-rules` template extensions

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-149@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2017-01-01
- 60-day deadline: 2017-03-02
- Draft #1 published: 2017-01-01
- Draft #2 published: 2017-01-02
- Draft #3 published: 2017-05-02
- Draft #4 published: 2017-07-01
- Finalized: 2017-07-08
- Post-finalization note added: 2017-08-07

Post-finalization note

After finalization of this SRFI, a discussion (starting here on the mailing list) revealed that the behaviour of this SRFI when a pattern variable in a subtemplate is followed by more instances of `<ellipsis>` than the subpattern in which it occurs differs from how implementations of the R6RS handle this case. While this SRFI specifies that the input subform is repeated for the innermost excess instances of `<ellipsis>`, the implementations of the R6RS repeat for the outermost excess instances.

There is no obvious reason to prefer one convention over the other as there are interesting macros that work with the former convention and not with the latter convention, and vice-versa.

However, this means that an implementation supporting the R6RS semantics of `syntax-rules` can never properly support this SRFI because this SRFI requires that `(srfi 149)` and `(scheme base)` export the same `syntax-rules`.

Therefore, the author of this SRFI now recommends that implementers ignore this extra requirement so that `(srfi 149)` may export a version of `syntax-rules` compatible with this SRFI and `(scheme base)` may export another. Instead, it is recommended that implementations provide a feature identifier `srfi-149-compatible` in case the implementation of `syntax-rules` in `(scheme base)` has the semantics as specified in this SRFI.

Abstract

The rules for valid `<template>`s of `<syntax rules>` are slightly softened to allow for more than one consecutive `<ellipsis>` in subtemplates, and to allow pattern variables in subtemplates to be followed by more instances of the identifier `<ellipsis>` than they are followed in the subpattern in which they occur.

Rationale

The pattern language for `syntax-rules`-macros as defined in 4.3.2 of the R7RS, impose two unnatural restrictions on the validity of templates in syntax rules.

The first restriction is that an instance of the identifier `<ellipsis>` cannot — by the formal syntax — immediately follow a `<template element>`, which already ends in an instance of the identifier `<ellipsis>`.

For example, the following expression is formally an error by the R7RS:

```
(let-syntax
    ((my-append
      (syntax-rules ()
        ((my-append (a ...) ...) '(a ... ...)))))
  (my-append (1 2 3) (4 5 6)))
```

With the extension of the pattern language as described in this SRFI — which is already supported by many Scheme systems like Chibi-Scheme and Kawa — the example expression ceases to be an error and instead evaluates into the list `(1 2 3 4 5 6)`.

The second unnatural restriction of the pattern language as described in the R7RS is that a difference is made between pattern variables that occur in subpatterns followed by no instance of the identifier `<ellipsis>` and those that occur in subpatterns followed by one or more instances of the identifier `<ellipsis>`, while there is no logical reason why such a difference has to be made. This implies that a pattern variable of the first kind can be followed by zero or more instances of the identifier `<ellipsis>` in the template, while a pattern variable of the second kind can only be followed by as many instances of the identifier `<ellipsis>` as the subpattern in which it occurs is followed.

For example, due to this restriction, transformers as specified by the `syntax-rules`-pattern language of the R7RS cannot naturally be lifted to transformers of lists that apply the original transformer to each element of the list (compare with the `map`-procedure of Scheme): If `((_ . <pattern>) <template>)` is a valid syntax rule of the `syntax-rules`-pattern language, the "lifted" rule `((_ <pattern> ...) (<template> ...))` is generally not a valid syntax rule.

With the extension of the pattern language as described in this SRFI — which, again, is already supported by at least Chibi-Scheme and Kawa — the "lifted" rule is always valid. A valid example (which evaluates to `(((bar 1) (bar 2)) ((baz 3) (baz 4)))`) is given by:

```
(let-syntax
    ((foo
      (syntax-rules ()
```

```
        ((foo (a b ...) ...) '(((a b) ...) ...))))
    (foo (bar 1 2) (baz 3 4)))
```

Note: There is at least one logical explanation for the presence of the second restriction in the R7RS: Allowing pattern variables with non-zero ellipsis count to have extra ellipses in the template has an ambiguous meaning without further specification (which is given below), in contrast to allowing this for zero-ellipsis variables, which is unambiguous. The ambiguity is in which of the ellipses in the template should iterate and which should replicate. In other words, which ones should be considered the ellipses that correspond to the ellipses in the pattern, and which ones should be considered the extra ellipses. The R7RS doesn't specify it because it doesn't have to since in in the R7RS, for any variable, either all of the ellipses in the template iterate (because the number of ellipses in the pattern and template match), or they all replicate (because there were no ellipses for this variable in the pattern).

Specification

A Scheme system that supports this SRFI should provide the library `(srfi 149)` so that Scheme code can detect (using `cond-expand`) whether the extensions proposed by this SRFI are supported by the Scheme system. The only export of `(srfi 149)` shall be `syntax-rules` with the same binding as `syntax-rules` from `(scheme base)`. Should this SRFI at some point formally become part of a new Scheme report (*e.g. R7RS-large*), the library `(srfi 149)` becomes obsolete and feature detection should be based on the feature identifier that defines the Scheme report in question.

Syntax

In section 7.1.5 of the R7RS, the rule for `<template element>` is changed into the following rule:

```
<template element>
  ⟶ <template>
  ⟶ <template element> <ellipsis>
```

Semantics

The rules in 4.3.2 of the R7RS are extended as follows:

- Pattern variables that occur in subpatterns followed by zero or more instances of the identifier <ellipsis> are allowed only in subtemplates that are followed by as many or more instances of `<ellipsis>`.
- If a pattern variable in a subtemplate is followed by more instances of `<ellipsis>` than the subpattern in which it occurs is followed, the input elements by which it is replaced in the output are repeated for the innermost excess instances of `<ellipsis>`. *(Note: This resolves the ambiguity mentioned above.)*
- It is an error if a subtemplate immediately followed by an instance of `<ellipsis>` does not contain at least one pattern variable such that the subpattern in which it occurs is followed by as many instances of `<ellipsis>`.

- If a template element is immediately followed by more than one instance of the identifier `<ellipsis>`, the semantics are analogous to when the instances of `<ellipsis>` are not in immediate succession but belonging to nested subtemplates.

Implementation

While it is technically not impossible to implement the `syntax-rules` of this SRFI as a custom macro transformer (see SRFI 147) in a portable way on an R7RS system, the benefits of such an implementation are negligible when weighted against the complexity of programming such a portable solution (and against its inherent slowness).

Therefore, this SRFI uses Chibi-Scheme's implementation of `syntax-rules` to provide a sample implementation. The advantage of Chibi-Scheme's implementation (besides the fact that this R7RS system is widely known and used) is that it is a portable solution as long as `er-macro-transformer` is available on the host system.

The following is the relevant part of the source file init-7.scm of Chibi-Scheme (please note Chibi-Scheme's copyright information and licensing terms):

```
(define (syntax-rules-transformer expr rename compare)
  (let ((ellipsis-specified? (identifier? (cadr expr)))
        (count 0)
        (_er-macro-transformer (rename 'er-macro-transformer))
        (_lambda (rename 'lambda))      (_let (rename 'let))
        (_begin (rename 'begin))        (_if (rename 'if))
        (_and (rename 'and))            (_or (rename 'or))
        (_eq? (rename 'eq?))            (_equal? (rename 'equal?))
        (_car (rename 'car))            (_cdr (rename 'cdr))
        (_cons (rename 'cons))          (_pair? (rename 'pair?))
        (_null? (rename 'null?))        (_expr (rename 'expr))
        (_rename (rename 'rename))      (_compare (rename 'compare))
        (_quote (rename 'syntax-quote)) (_apply (rename 'apply))
        (_append (rename 'append))      (_map (rename 'map))
        (_vector? (rename 'vector?))    (_list? (rename 'list?))
        (_len (rename'len))             (_length (rename 'length*))
        (_- (rename '-))   (_>= (rename '>=))   (_error (rename 'error))
        (_ls (rename 'ls)) (_res (rename 'res)) (_i (rename 'i))
        (_reverse (rename 'reverse))
        (_vector->list (rename 'vector->list))
        (_list->vector (rename 'list->vector))
        (_cons3 (rename 'cons-source))
        (_underscore (rename '_)))
    (define ellipsis (if ellipsis-specified? (cadr expr) (rename '...)))
    (define lits (if ellipsis-specified? (car (cddr expr)) (cadr expr)))
    (define forms (if ellipsis-specified? (cdr (cddr expr)) (cddr expr)))
    (define (next-symbol s)
      (set! count (+ count 1))
      (rename (string->symbol (string-append s (%number->string count)))))
    (define (expand-pattern pat tmpl)
      (let lp ((p (cdr pat))
               (x (list _cdr _expr))
               (dim 0)
```

```
          (vars '())
          (k (lambda (vars)
               (list _cons (expand-template tmpl vars) #f))))
        (let ((v (next-symbol "v.")))
          (list
           _let (list (list v x))
           (cond
            ((identifier? p)
             (if (any (lambda (l) (compare p l)) lits)
                 (list _and
                       (list _compare v (list _rename (list _quote p)))
                       (k vars))
                 (if (compare p _underscore)
                     (k vars)
                     (list _let (list (list p v)) (k (cons (cons p dim) vars))))))
            ((ellipsis? p)
             (cond
              ((not (null? (cdr (cdr p))))
               (cond
                ((any (lambda (x) (and (identifier? x) (compare x ellipsis)))
                      (cddr p))
                 (error "multiple ellipses" p))
                (else
                 (let ((len (length* (cdr (cdr p))))
                       (_lp (next-symbol "lp.")))
                   `(,_let ((,_len (,_length ,v)))
                      (,_and (,_>= ,_len ,len)
                             (,_let ,_lp ((,_ls ,v)
                                          (,_i (,_- ,_len ,len))
                                          (,_res (,_quote ())))
                                    (,_if (,_>= 0 ,_i)
                                        ,(lp `(,(cddr p)
                                               (,(car p) ,(car (cdr p))))
                                             `(,_cons ,_ls
                                                      (,_cons (,_reverse ,_res)
                                                              (,_quote ()))))
                                        dim
                                        vars
                                        k)
                                    (,_lp (,_cdr ,_ls)
                                          (,_- ,_i 1)
                                          (,_cons3 (,_car ,_ls)
                                                   ,_res
                                                   ,_ls)))))))))
              ((identifier? (car p))
               (list _and (list _list? v)
                     (list _let (list (list (car p) v))
                           (k (cons (cons (car p) (+ 1 dim)) vars)))))
              (else
               (let* ((w (next-symbol "w."))
                      (_lp (next-symbol "lp."))
                      (new-vars (all-vars (car p) (+ dim 1)))
                      (ls-vars (map (lambda (x)
                                      (next-symbol
                                       (string-append
```

```
                                    (symbol->string
                                     (identifier->symbol (car x)))
                                    "-ls")))
                                new-vars))
                        (once
                         (lp (car p) (list _car w) (+ dim 1) '()
                             (lambda (_)
                               (cons
                                _lp
                                (cons
                                 (list _cdr w)
                                 (map (lambda (x l)
                                        (list _cons (car x) l))
                                      new-vars
                                      ls-vars)))))))
                    (list
                     _let
                     _lp (cons (list w v)
                               (map (lambda (x) (list x (list _quote '()))) ls-vars))
                     (list _if (list _null? w)
                           (list _let (map (lambda (x l)
                                             (list (car x) (list _reverse l)))
                                           new-vars
                                           ls-vars)
                                 (k (append new-vars vars)))
                           (list _and (list _pair? w) once)))))))
                ((pair? p)
                 (list _and (list _pair? v)
                       (lp (car p)
                           (list _car v)
                           dim
                           vars
                           (lambda (vars)
                             (lp (cdr p) (list _cdr v) dim vars k)))))
                ((vector? p)
                 (list _and
                       (list _vector? v)
                       (lp (vector->list p) (list _vector->list v) dim vars k)))
                ((null? p) (list _and (list _null? v) (k vars)))
                (else (list _and (list _equal? v p) (k vars))))))))
(define (ellipsis-escape? x) (and (pair? x) (compare ellipsis (car x))))
(define (ellipsis? x)
  (and (pair? x) (pair? (cdr x)) (compare ellipsis (cadr x))))
(define (ellipsis-depth x)
  (if (ellipsis? x)
      (+ 1 (ellipsis-depth (cdr x)))
      0))
(define (ellipsis-tail x)
  (if (ellipsis? x)
      (ellipsis-tail (cdr x))
      (cdr x)))
(define (all-vars x dim)
  (let lp ((x x) (dim dim) (vars '()))
    (cond ((identifier? x)
           (if (any (lambda (lit) (compare x lit)) lits)
```

```
                  vars
                  (cons (cons x dim) vars)))
            ((ellipsis? x) (lp (car x) (+ dim 1) (lp (cddr x) dim vars)))
            ((pair? x) (lp (car x) dim (lp (cdr x) dim vars)))
            ((vector? x) (lp (vector->list x) dim vars))
            (else vars))))
    (define (free-vars x vars dim)
      (let lp ((x x) (free '()))
        (cond
         ((identifier? x)
          (if (and (not (memq x free))
                   (cond ((assq x vars) => (lambda (cell) (>= (cdr cell) dim)))
                         (else #f)))
              (cons x free)
              free))
         ((pair? x) (lp (car x) (lp (cdr x) free)))
         ((vector? x) (lp (vector->list x) free))
         (else free))))
    (define (expand-template tmpl vars)
      (let lp ((t tmpl) (dim 0))
        (cond
         ((identifier? t)
          (cond
           ((find (lambda (v) (eq? t (car v))) vars)
            => (lambda (cell)
                 (if (<= (cdr cell) dim)
                     t
                     (error "too few ...'s"))))
           (else
            (list _rename (list _quote t)))))
         ((pair? t)
          (cond
           ((ellipsis-escape? t)
            (list _quote
                  (if (pair? (cdr t))
                      (if (pair? (cddr t)) (cddr t) (cadr t))
                      (cdr t))))
           ((ellipsis? t)
            (let* ((depth (ellipsis-depth t))
                   (ell-dim (+ dim depth))
                   (ell-vars (free-vars (car t) vars ell-dim)))
              (cond
               ((null? ell-vars)
                (error "too many ...'s"))
               ((and (null? (cdr (cdr t))) (identifier? (car t)))
                ;; shortcut for (var ...)
                (lp (car t) ell-dim))
               (else
                (let* ((once (lp (car t) ell-dim))
                       (nest (if (and (null? (cdr ell-vars))
                                      (identifier? once)
                                      (eq? once (car vars)))
                                 once ;; shortcut
                                 (cons _map
                                       (cons (list _lambda ell-vars once)
```

```
                                                ell-vars))))
                          (many (do ((d depth (- d 1))
                                     (many nest
                                           (list _apply _append many)))
                                    ((= d 1) many))))
                        (if (null? (ellipsis-tail t))
                            many ;; shortcut
                            (list _append many (lp (ellipsis-tail t) dim)))))))))
                  (else (list _cons3 (lp (car t) dim) (lp (cdr t) dim) (list _quote t)))))
              ((vector? t) (list _list->vector (lp (vector->list t) dim)))
              ((null? t) (list _quote '()))
              (else t))))
         (list
          _er-macro-transformer
          (list _lambda (list _expr _rename _compare)
                (list
                 _car
                 (cons
                  _or
                  (append
                   (map
                    (lambda (clause) (expand-pattern (car clause) (cadr clause)))
                    forms)
                   (list
                    (list _cons
                          (list _error "no expansion for"
                                (list (rename 'strip-syntactic-closures) _expr))
                          #f)))))))))

(define-syntax syntax-rules/aux
  (er-macro-transformer syntax-rules-transformer))

(define-syntax syntax-rules
  (er-macro-transformer
   (lambda (expr rename compare)
     (if (identifier? (cadr expr))
         (list (rename 'let) (list (list (cadr expr) #t))
               (cons (rename 'syntax-rules/aux) (cdr expr)))
         (syntax-rules-transformer expr rename compare)))))
```

The implementation of the library `(srfi 149)` on a supporting Scheme system is as simple as possible:

```
(define-library (srfi 149)
  (export syntax-rules)
  (import (scheme base)))
```

Acknowledgements

The title of this SRFI is a tribute to the title of SRFI 46.

I would like to thank Alex Shinn for having made the free software Chibi-Scheme, from which I shamelessly stole the implementation of `syntax-rules`. Of course, this does not imply that he

necessarily endorses this SRFI.

I would also like to thank Al Petrofsky who pointed out that the need of resolving the ambiguity when lifting the second restriction should be made explicit and not just implicit in this SRFI. I used some of his wording on the mailing list while amending this SRFI.

Copyright

Editor: Arthur A. Gleckler