Title

Basic Syntax-rules Extensions

Author

Taylor Campbell

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-46@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2003-10-06
- Draft: 2003-10-06--2004-01-15
- Revised: 2004-09-12
- Revised: 2004-11-07
- Final: 2005-02-28

Abstract

This SRFI proposes two extensions to the R5RS[1] `syntax-rules` pattern language: the first allows `syntax-rules` macros to generate macros, where the macro-generated macros use ellipsis that is not used by the macro-generating macros; the second allows for 'tail patterns.'

Rationale

Macros that generate macros are fairly common and very useful in a variety of circumstances, e.g. in CPS macros[2] for local continuation macros. R5RS currently provides no mechanism for generating literal ellipsis in a `syntax-rules` clause's template — all ellipsis in the template is processed by the macro. Macros that generate macros are thereby restricted, since the generated macros are unable to make use of ellipsis. This is a severe restriction that can be removed by a simple extension to the `syntax-rules` syntax.

Additionally, it is often very convenient to be able to match a finite sequence of elements not only *before* any ellipsis but also *after* any ellipsis. Such 'tail patterns' are unsupported by R5RS in its specification of `syntax-rules`; thus, this SRFI proposes the simple addition of tail patterns to `syntax-rules`.

Specification

`Syntax-rules` syntax is extended so that there is an extra possible token before the literal identifier list:

```
(syntax-rules [<ellipsis-identifier>] (<literal-identifier> ...)
  (<pattern> <template>)
  ...)
```

`Ellipsis-identifier` specifies the token used for ellipsis. It defaults to the usual R5RS[1] ellipsis identifier, `...`, but it can be specified to be any identifier, such as `:::`. This identifier's specification is considered to be a binding whose scope is the rules of the transformer. The macro system implementation must make the hygienic arrangements described in R5RS's section 4.3 to preserve the lexical scope of these bindings.

The `syntax-rules` pattern language is also extended to allow 'tail patterns.' The following clauses are added to `<pattern>`:

```
 (<pattern> ... <ellipsis> <pattern> ...)
#(<pattern> ... <ellipsis> <pattern> ...)
```

And the following clauses are added to the semantics of `syntax-rules`' pattern matching:

- *P* is of the form $(P_1\ \ldots\ P_{x-1}\ P_x$ `<ellipsis>` $P_{x+1}\ \ldots\ P_y)$ where `<ellipsis>` is the identifier `...` and *F* is a proper list of *M* forms such that *M* >= *Y*, the first *X-1* of which match P_1 through P_{x-1}, respectively, the forms F_x through $F_{m-(y-x)}$, where F_i is the I^{th} element of the proper list *F*, all match P_x, and the forms $F_{m-(y-x)}$ through F_m match the patterns P_{x+1} through P_y.
- *P* is of the form $\#(P_1\ \ldots\ P_{x-1}\ P_x$ `<ellipsis>` $P_{x+1}\ \ldots\ P_y)$ where `<ellipsis>` is the identifier `...` and *F* is a vector of *M* forms such that *M* >= *Y*, the first *X-1* of which match P_1 through P_{x-1}, respectively, the forms F_x through $F_{m-(y-x)}$, where F_i is the I^{th} element of the vector *F*, all match P_x and the forms $F_{m-(y-x)}$ through F_m all match P_{x+1} through P_y, respectively.

Examples

```
;;; Examples of the user-specified ellipsis token extension

;;; Utility macro for CPS macros
```

```
(define-syntax apply-syntactic-continuation
  (syntax-rules ()
    ((apply-syntactic-continuation (?k ?env ...) . ?args)
     (?k ?env ... . ?args))))

;;; Generates a list of temporaries, for example to implement LETREC
;;; (see below), and 'returns' it by CPS.
(define-syntax generate-temporaries
  (syntax-rules ()
    ((generate-temporaries ?origs ?k)
     (letrec-syntax
         ((aux (syntax-rules ::: ()
                 ;; We use a trick here: pass the continuation again
                 ;; to AUX in case it contains ellipsis.  If we stuck
                 ;; it right into AUX's template, AUX would process the
                 ;; ellipsis in ?K as ellipsis for something in the AUX
                 ;; macro.
                 ((aux ?temps () ?k*)
                  (apply-syntactic-continuation ?k* ?temps))
                 ;; Be careful about the ellipsis!
                 ((aux (?temp :::) (?x ?more :::) ?k*)
                  (aux (?temp ::: new-temp)
                       (?more :::)
                       ?k*)))))
       (aux () ?origs ?k)))))

;;; Instead of having lots of auxiliary clauses in LETREC, like in the
;;; R5RS sample implementation, we use GENERATE-TEMPORARIES.  Instead
;;; of 'returning,' like an ordinary function, we create a continuation
;;;  for GENERATE-TEMPORARIES with LET-SYNTAX.  Since this continuation
;;; uses ellipsis, we must use the ellipsis token extension.
(define-syntax letrec
  (syntax-rules ()
    ((letrec ((?var ?init) ...) ?body1 ?body2 ...)
     (let-syntax
         ((k (syntax-rules ::: ()
               ;; Use the same trick as with the continuations in
               ;; GENERATE-TEMPORARIES.  Be careful about the ellipsis!
               ((k ((?var* ?init*) :::)
                   (?body1* ?body2* :::)
                   ;; Here are the actual arguments to the continuation
                   ;; -- the previous bits of the pattern were just the
                   ;; 'environment' of the continuation --:
                   (?temp :::))
                (let ((?var* (if #f #f)) ; Get an 'unspecific' value.
                      :::)
                  (let ((?temp ?init*) :::)
                    (set! ?var* ?temp) :::
                    (let () ?body1* ?body2* :::)))))))
       (generate-temporaries (?var ...)
         ;; Pass K the environment.  GENERATE-TEMPORARIES will add the
         ;; temporary variable list argument.
         (k ((?var ?init) ...) (?body1 ?body2 ...)))))))

;;; The next example uses two other macros that we don't define here:
;;; SYNTACTIC-SYMBOL? and UNION.  (SYNTACTIC-SYMBOL? <x> <sk> <fk>)
;;; expands to SK if X is a symbol or FK otherwise.  (UNION <s1> <s2>
;;; <k>) applies K with APPLY-SYNTACTIC-CONTINUATION to the union of
;;; the syntactic lists S1 and S2.  Both of SYNTACTIC-SYMBOL? and UNION
;;; are possible to implement here, but we sha'n't bother with them, as
;;; we wish only to demonstrate an example of macros generating macro-
;;; generating macros, and they provide no such examples.

;;; ALL-SYMBOLS digs out all the symbols in a syntax.
(define-syntax all-symbols
  (syntax-rules ()
    ((all-symbols (?x . ?y) ?k)
     (let-syntax
         ((k (syntax-rules :::0 ()
               ((k ?y* ?k*  (?symbol :::0))
                (let-syntax
                    ((k* (syntax-rules :::1 ()
                           ;; Doubly nested ellipsis: we use another
                           ;; distinct ellipsis token.
                           ((k* ?k** (?symbol* :::1))
                            (union (?symbol  :::0)
                                   (?symbol* :::1)
                                   ?k**)))))
                  (all-symbols ?y* (k* ?k*)))))))
       (all-symbols ?x (k ?y ?k))))

    ((all-symbols #(?x ...) ?k)
     (all-symbols (?x ...) ?k))

    ((all-symbols ?x ?k)
     (syntax-symbol? ?x
       (apply-syntactic-continuation ?k (?x))
       (apply-syntactic-continuation ?k ())))))

(all-symbols (foo 4 bar #(#t (baz (#f quux)) zot) (mumble #(frotz)))
             (quote)) ; => (frotz mumble zot quux baz bar foo)

;;; This example demonstrates the hygienic renaming of the ellipsis
;;; identifiers.

(let-syntax
    ((f (syntax-rules ()
          ((f ?e)
           (let-syntax
               ((g (syntax-rules ::: ()
                     ((g (??x ?e) (??y :::))
                      '((??x) ?e (??y) :::)))))
             (g (1 2) (3 4)))))))
  (f :::))
    ; => ((1) 2 (3) (4)), if hygienic rules of ellipsis identifiers are
    ;      correctly implemented, not ((1) (2) (3) (4))

;;; --------------------
;;; Examples of tail patterns

;;; This example of the tail pattern extension is a crippled version of
;;; R5RS's BEGIN special form.  (It is crippled because it does not
;;; support internal definitions or commands within its body returning
;;; fewer or more than one value.)

(define-syntax fake-begin
  (syntax-rules ()
    ((fake-begin ?body ... ?tail)
     (let* ((ignored ?body) ...) ?tail))))

;;; For example,
;;;   (FAKE-BEGIN
;;;     (DISPLAY "Hello,")
;;;     (WRITE-CHAR #\SPACE)
;;;     (DISPLAY "world!")
;;;     (NEWLINE))
;;; would expand to
;;;   (LET* ((IGNORED (DISPLAY "Hello,"))
;;;          (IGNORED (WRITE-CHAR #\SPACE))
;;;          (IGNORED (DISPLAY "world!")))
;;;     (NEWLINE))

(let-syntax
    ((foo (syntax-rules ()
            ((foo ?x ?y ... ?z)
             (list ?x (list ?y ...) ?z)))))
  (foo 1 2 3 4 5))
    ; => (1 (2 3 4) 5)
```

Implementation

There are two example macro expanders here provided that implement the proposed extensions, Alexpander & EIOD. Alexpander is a complete, sophisticated expander for the `syntax-rules` macro system; EIOD is an implementation of R5RS's `eval` that obviously requires a macro expander internally. Both were written by Al* Petrofsky; see their source for copyright information & licensing terms. Alexpander is available at <http://srfi.schemers.org/srfi-46/alexpander.scm> and EIOD is available at <http://srfi.schemers.org/srfi-46/eiod.scm>.

Acknowledgements

Al* Petrofsky provided much crucial input about the fundamental design of this SRFI's extensions on the mailing list that strongly influenced the final result; I thank him greatly for that input.

References

1. Richard Kelsey, William Clinger, and Jonathon Rees (editors).
   The Revised^5 Report on the Algorithmic Language Scheme
   Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998, and ACM SIGPLAN Notices, Vol. 33, No. 9, October, 1998.
   http://www.schemers.org/Documents/Standards/R5RS/
2. Erik Hilsdale and Daniel P. Friedman.
   Writing Macros in Continuation-Passing Style
   Scheme and Functional Programming 2000, September, 2000.
   http://www.ccs.neu.edu/home/matthias/Scheme2000/hilsdale.ps

Copyright

Copyright (C) Taylor Campbell (2005). All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Editor: David Rush

Last modified: Sun May 17 12:34:31 MST 2009